UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

CASSIANO JAEGER STRADOLINI

# Migração de sistemas monolíticos para microsserviços: Estudo de caso de migração de um módulo de pagamentos de e-Commerce

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof. Dr. Pimenta Marcelo

Porto Alegre
2022

*“Any fool can write code that a computer can understand. Good programmers write code that humans can understand"*

–MARTIN J. FOWLER

## AGRADECIMENTOS

Agradeço aos meus pais, Eneida Jaeger Stradolini e Marco Antonio Fontoura Stradolini, por darem todo o suporte que eu precisei durante os 26 anos de vida que tenho. Gostaria de agradecer também aos meus amigos que me motivaram a crescer e buscar novos desafios. Por fim, também gostaria de agradecer as pessoas que me ajudaram diretamente na conclusão deste trabalho: Caetano Jaeger Stradolini, Rafael Perfeito, Vinicius Ambrosi e Marlom Oliveira.

## RESUMO

Este trabalho propõe um estudo de caso de migração de um módulo de pagamento de um sistema de arquitetura monolítica para uma arquitetura de microsserviços implantada em uma provedora de nuvem pública. O estudo faz uma investigação profunda e compara diversas tecnologias, abordagens e técnicas para identificar o passo-a-passo necessário para este tipo migração, assim como as melhores escolhas feitas baseadas no contexto do sistema a ser migrado. Dadas as discussões sobre a melhor abordagem, foi desenvolvida uma proposta de migração. Essa proposta foi implementada como uma prova de conceito para validar a sua viabilidade. Após isso, a proposta foi revisada e analisada por especialistas da área de tecnologia, cujas avaliações corroboraram com as decisões de arquitetura tomadas pelo autor do trabalho. São evidenciados os benefícios e desafios que essa proposta introduziu na arquitetura atual, bem como as principais lições aprendidas.



**Palavras-chave:** Microsserviços. Arquitetura monolítica. Arquitetura de sistemas.

**Migration from monolithic systems to micro services: Case study of migration of an e-Commerce payment module**

## ABSTRACT

This work proposes a case study of a payment module migration from a monolithic architecture system to a micro services architecture implemented in a private cloud provider. This study makes an detailed investigation and compares different technologies, approaches and techniques to identify the step-by-step necessary for this type of migration, as well as the best choices made based on the context of the system to be migrated. Given the discussions on the best approach, a system migration proposal was developed. This proposal was implemented as a proof of concept to validate the its feasibility. After that, the proposal was reviewed and analyzed by experts in the technology area, whose assessments corroborated the architectural decisions taken by the author of the work. The benefits and challenges that this proposal introduced in the current architecture are highlighted, as well as the main lessons learned.

**Keywords:** Micro Services. Monolithic Systems. System Architecture.

## LISTA DE FIGURAS

## LISTA DE TABELAS

## LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| API | Application Programming Interface |
| DDD | Domain-Driven Design |
| REST | Representional State Transfer |
| GCP | Google Cloud Platform |
| GKE | Google Kubernetes Engine |
| AWS | Amazon Web Services |
| EKS | Elastic Kubernetes Service |
| IPC | Inter-Process Communication |
| I/O | Input/Output |
| VM | Virtual Machine |

**SUMÁRIO**

# 1 INTRODUÇÃO

Esta seção irá abordar os temas introdutórios deste trabalho, sendo eles a motivação do desenvolvimento do mesmo, objetivos, convenções de texto utilizadas e, por fim, o tipo de estrutura em que o texto estará organizado.

## 1.1 Motivação

O autor deste trabalho trabalhou por mais de 5 anos no desenvolvimento de uma plataforma de e-Commerce em uma arquitetura monolítica. Ao longo desses anos o mesmo foi passando e identificando diversos desafios que tal arquitetura apresentava em sistemas de alta carga e acesso de usuários concorrentes, sendo alguns deles:

- Capacidade de escalar a aplicação para que a mesma suportasse uma maior quantidade de carga de trabalho em épocas de aumento de vendas, como por exemplo a *Black Friday*[1].
- Dificuldade em estender a base de código, por conta do alto acoplamento entre as classes da aplicação monolítica.
- Grande tempo de preparação de novos membros na equipe de desenvolvimento por conta da grande complexidade da base de código.
- O emprego de tecnologias novas era limitado por conta de muitas tecnologias antigas não poderem ser removidas da arquitetura.

O assunto de migrar tal arquitetura para uma arquitetura modular de microsserviços era bastante discutido e inclusive estava nos planos da empresa iniciar tal processo o mais rápido possível. Todavia, o autor saiu desta empresa e ingressou em outra em uma posição de desenvolvedor que iria trabalhar diretamente com a migração do sistema de e-Commerce da empresa para uma arquitetura de microsserviços na nuvem.

Percebeu-se que os desafios e dificuldades enfrentadas pela equipe de desenvolvimento dessa empresa eram muito parecidos com os que o autor enfrentou na antiga empresa em que trabalhava. Por conta disso o autor decidiu estudar a fundo o processo de migração de uma arquitetura monolítica para uma arquitetura de microsserviços.

Durante as conversas iniciais de orientação sobre o trabalho de conclusão foi le-

[1] A Black Friday refere-se ao dia seguinte ao Dia de Ação de Graças e é simbolicamente vista como o início da temporada crítica de compras de fim de ano. As lojas oferecem grandes descontos em eletrônicos, brinquedos e outros presentes (WIKIPEDIA, 2022).

vantada a ideia de se aprofundar um pouco mais nesse tópico e produzir um trabalho exatamente sobre o assunto que o autor estava pesquisando. Fazendo com que os resultados de tal estudo pudessem ser divididos com a comunidade acadêmica.

## 1.2 Objetivos

Esta seção irá abordar o objetivo principal deste trabalho, assim como os objetivos secundários propostos para o mesmo.

### 1.2.1 Objetivo principal

Este trabalho de conclusão possui dois objetivos principais. O primeiro é mostrar e exemplificar um processo de migração de uma arquitetura monolítica para uma arquitetura de microsserviços na nuvem, focado em resolver os problemas, como por exemplo dificuldade de escalabilidade e manutenção, vistos na arquitetura legada. Evidenciando quais são estes problemas e desafios e como a arquitetura proposta pode resolver, ou pelo menos minimizar, cada um deles.

O segundo objetivo é, juntamente com essa proposta de migração, explicar e detalhar todo o processo de decisão das tecnologias e ferramentas escolhidas e por quais motivos tais foram aceitas. Várias alternativas serão debatidas no trabalho e as mais adequadas serão escolhidas para a arquitetura. Todo o processo de escolha será documentado e explicado para que se entenda a lógica por trás das escolhas, auxiliando assim em outras propostas de migrações de arquiteturas.

Os dois objetivos caminham juntos para que este trabalho seja um material de referência para futuras migrações, onde as mesmas possam estender as análises aqui contidas e enriquecer este tópico.

### 1.2.2 Objetivo secundário

Implementar a arquitetura proposta a fim de validar as discussões contidas no trabalho, assim como corroborar com as análises de especialistas de tecnologia que irão avaliar a proposta de migração.

## 1.3 Convenções utilizadas

Ao longo do trabalho, são apresentadas diversas figuras de diagramas de arquitetura. A não ser caso indicado explicitamente diferente, a direção das setas nos desenhos indica a direção da dependência, ou direção da referência.

## 1.4 Estrutura do texto

A estrutura deste trabalho segue a seguinte organização:

- O capítulo 2 introduz o referencial teórico necessário para o bom entendimento do trabalho produzido, fazendo referências a literatura e documentações oficiais de tecnologias ou ferramentas. Algumas das bases teóricas relacionadas são: Kubernetes, conceito de bancos, arquitetura monolítica, arquitetura de microsserviços, DDD e estratégias de migração vistos na literatura e na indústria.
- O capítulo 3 expõe a situação atual do sistema legado, seus componentes, problemas e desafios.
- O capítulo 4 propõe uma proposta de migração para a arquitetura de microsserviços através de grande investigação sobre diversos tópicos, sendo alguns deles: Qual linguagem de programação usar? Que tipo de banco de dados? Que gerenciadores de contêineres são utilizados? Cada tópico analisado terá discussões sobre as possíveis propostas e todo o processo de decisão de qual proposta foi escolhida.
- O capítulo 5 detalha como foi implementado a migração proposta através de imagens, códigos e texto, de forma a analisar e entender as diferenças e melhorias da nova arquitetura proposta em relação à antiga.
- O capítulo 6 conterá uma reflexão sobre os feedbacks coletados de especialistas de tecnologia, através de um questionário, sobre a arquitetura proposta. Suas respostas serão discutidas pelo autor a fim de entender se a arquitetura proposta consegue atingir seus objetivos.
- O capítulo 7 apresenta a conclusão do trabalho, assim como as limitações enfrentadas durante o desenvolvimento e também os possíveis trabalhos futuros baseados nas pendências do trabalho atual e ideias que foram aparecendo durante o desenvolvimento deste trabalho.
- Por fim, os anexos incluem imagens do questionário apresentado aos respondentes

da pesquisa de opinião.

# 2 FUNDAMENTOS E TRABALHOS RELACIONADOS

Esta seção tratará dos conceitos e fundamentos necessários para o entendimento do trabalho desenvolvido. Após a descrição de tais fundamentos, serão analisados trabalhos relacionados ao tema de microsserviços e migrações de arquiteturas monolíticas e por fim a motivação por trás deste trabalho.

## 2.1 Modularidade

Fowler (2015) define modularização como uma divisão do sistema que permite que lhe sejam feitas modificações sem a necessidade de entendê-lo como um todo, mas apenas conhecendo bem um subconjunto. Ele também define um componente como um caso particular de módulo: uma unidade de software que seja independentemente substituível e atualizável. Os módulos de um sistema terão duas características complementares: coesão e acoplamento. Normalmente, para se obter os bons ganhos que a modularidade promete, se procura projetar cada módulo com a maior coesão e menor acoplamento possível.

### 2.1.1 Acoplamento

Acoplamento pode ser definido como o grau de interdependência entre dois ou mais módulos de um sistema de software (MYERS, 1975). Um baixo ou fraco acoplamento é definido pela baixa existência dessas relações.

O Alto acoplamento, no âmbito de desenvolvimento de software, gera problemas de entendimento e reuso do módulo, visto que sua alta complexidade aliada a uma alta interdependência dificulta a análise e entendimento do mesmo (MEYER, 1997).

### 2.1.2 Coesão

Coesão define o grau em os elementos dentro de um módulo pertencem juntos (YOURDON; PRESS; CONSTANTINE, 1999). Newman (2019) cita como a melhor definição de coesão a seguinte frase: “código que é alterado junto, deve estar junto”.

Quando a coesão de um módulo é dita como alta, significa que as suas classes, mé-

todos e variáveis dependem entre si e se relacionam logicamente como um todo (MARTIN, 2008). Em contrapartida, uma baixa coesão nos módulos acaba gerando problemas de entendimento, manutenção e reusabilidade do código (NEWMAN, 2019).

## 2.2 Escalabilidade

Não é clara a definição formal sobre escalabilidade (HILL, 1990), entretanto ela pode ser definida como um atributo que descreve a habilidade de um sistema, processo ou organização de crescer e gerenciar uma alta na demanda de recursos (TECHOPEDIA, 2017). Métodos de escalabilidade podem ser definidos em dois grandes grupos: escalonamento vertical e horizontal (MICHAEL et al., 2007).

A escalabilidade vertical, ou também conhecido como scale-out, em sistemas de informação, significa a capacidade de se adicionar recursos (ou removê-los) de um único nodo, normalmente envolvendo a adição de CPU, memória RAM ou memória de armazenamento em disco (EL-REWINI; ABD-EL-BARR, 2005).

A escalabilidade horizontal define a capacidade de se adicionar mais nodos (ou removê-los) de um sistema, como por exemplo adicionar novas máquinas ou instâncias de uma aplicação trabalhando paralelamente (EL-REWINI; ABD-EL-BARR, 2005). Um exemplo desse tipo de escalabilidade é o uso de sistemas "mercadorias"em pesquisas de biotecnologia, onde o uso de supercomputadores é necessário.

## 2.3 Disponibilidade

O conceito de disponibilidade relaciona a probabilidade de um sistema estar operacional em um determinado momento, ou seja, a quantidade de tempo que um dispositivo está realmente operando como a porcentagem do tempo total que deveria estar operando (LAROS et al., 2012).

Os recursos de disponibilidade permitem que o sistema permaneça operacional mesmo quando ocorrem falhas. Um sistema altamente disponível possui a habilidade de desabilitar a parte do sistema com defeito e continuar operando com capacidade reduzida. Em contraste, um sistema menos capaz pode falhar e se tornar totalmente inoperante. A disponibilidade normalmente é dada como uma porcentagem do tempo que se espera que um sistema esteja disponível, usualmente um valor de referência em relação ao tempo que

o sistema fica disponível durante o período de um ano (HAWKINS, 2000).

Configurar uma arquitetura de alta disponibilidade requer o endereçamento de dois grandes pontos: O gerenciamento de recursos redundantes e confiáveis e não permitir um ponto único de falha que possa tornar nossa aplicação inoperante (NEWMAN, 2015). A proteção dos dados nesse tópico é o ponto crucial, pois com os dados protegidos não importa o que aconteça com a sua infraestrutura ou qualquer outra coisa, é possível reconstruí-los.

## 2.4 Arquitetura monolítica

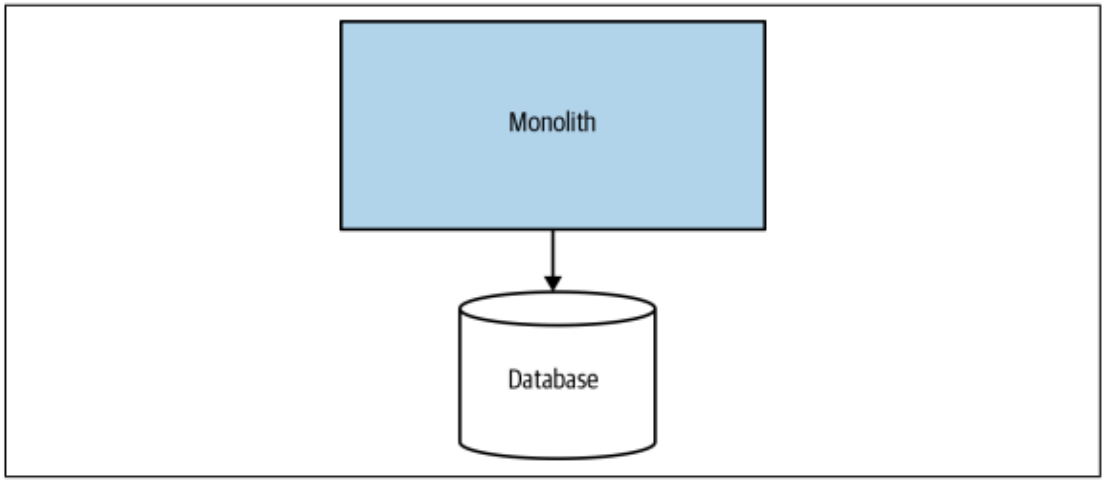
Um software dito monolítico é caracterizado por uma unidade de sistema completa que se apresenta e é implantado como um único processo que executa em um computador, mesmo que faça uso de bibliotecas compartilhadas (arquivos separados). Qualquer alteração feita no sistema requer que uma nova implantação da aplicação inteira seja executado (NEWMAN, 2019).

Newman (2019) define dois tipos principais de sistemas monolíticos: Os sistemas monolíticos de apenas um processo (uma tradução livre de *single process monoliths*) e sistemas monolíticos distribuídos. O sistema monolítico de apenas um processo é o tipo mais comum de monólito onde toda a base de código é executada como um único processo no sistema operacional. Existe uma variação deste sistema que é chamado de monólito modular, onde a base de código é dividida e modulada, porém a execução final ainda é feita através de um único processo. Essa variação, muitas vezes, é uma boa escolha para a maioria das organizações, pois adiciona um bom grau de modularidade, o que facilita o desenvolvimento de software, como visto na seção 2.1. Apesar disso, um dos problemas dessa variação é o fato de que muitas vezes, esses módulos acessam o mesmo banco de dados que não possui o mesmo nível de decomposição em relação ao código, gerando desafios significativos se for preciso substituir o monólito por outro sistema.

Em relação ao monólito distribuído, Newman (2019) o define como um conjunto de serviços separados que precisam, por algum motivo, serem implantados de forma conjunta. Na experiência de Newman, esse tipo de sistema monolítico possui todos os problemas de sistemas distribuídos e de sistemas monolíticos de apenas um processo.

O sistema monolítico, independente do seu tipo, é comumente mais vulnerável aos perigos específicos de acoplamento na implementação e implantação do sistema. Quanto mais pessoas estiverem trabalhando no mesmo local, mais vezes elas estarão no caminho

Figura 2.1: Diagrama de uma aplicação monolítica




Fonte: (NEWMAN, 2019)

de uma das outras. Diferentes times e pessoas querendo implantar mudanças em produção em tempos diferentes, mudanças de no mesmo pedaço de código por times diferentes e muitas vezes dificuldade de saber quem tem o domínio da base de dados ou quem toma as decisões referentes a mesma (NEWMAN, 2019).

Apesar dos pontos acima, a arquitetura monolítica não possui apenas desvantagens por si só. Pode sim se tratar de uma ótima escolha arquitetural, oferecendo diversos benefícios (NEWMAN, 2019; RICHARDSON, 2018).

## 2.5 Arquitetura de microsserviços

O termo microsserviço está atualmente em grande uso na comunidade de arquitetura de software. Mesmo sendo precaução a reação natural ao se olhar esse tipo de situação, a terminologia descreve um estilo de sistema que tem sido cada vez mais atraente (FOWLER; LEWIS, 2014). Muitas organizações descobriram que a adoção de uma arquitetura de microsserviços bem desenvolvida permite a entrega de sistemas mais rapidamente e também de abraçar novas tecnologias com maior facilidade (NEWMAN, 2015). microsserviços permite uma significativa maior liberdade para reagir e tomar diferentes decisões, facilitando a adaptação em situações de mudança inevitáveis (NEWMAN, 2015).

Newman (2019) afirma que microsserviços são serviços implantados independentemente que são modelados em volta de um domínio de negócio. Eles se comunicam entre si via transmissão de rede. A arquitetura é basicamente uma implementação de uma

abordagem de arquitetura de software chamada de *SOA*[1] usada para construir sistemas flexíveis e que possam ser implantados de forma independente (PAUTASSO et al., 2017).

A arquitetura de microsserviços é uma ótima opção em desenvolvimento de sistemas grandes e complexos, onde muitas vezes possuem uma escala global de atuação (FOWLER; LEWIS, 2014). É comum o pensamento de que microsserviços são uma nova bala de prata da arquitetura de software, porém a arquitetura possui uma série de efeitos colaterais e cria desafios que não existem na arquitetura monolítica e devem ser endereçados corretamente (NEWMAN, 2019).

Newman (2015) define uma série de características que diferenciam a arquitetura de microsserviços das demais, sendo elas:

- Pequenos serviços que fazem apenas uma coisa muito bem;
- Os serviços devem ser autônomos, sem estarem acoplados ou que modifiquem outros serviços durante novas implantações. Suas comunicações são feitas através de *API's*, a fim de diminuir o acoplamento entre os serviços.

Além disso, Newman (2015) também define alguns benefícios importantes da arquitetura de microsserviços:

- Heterogeneidade tecnológica: Com um sistema composto por diversos microsserviços, é possível escolher diferentes tecnologias para trabalhar em cada um deles;
- Resiliência: Um conceito chave na engenharia de resiliência é o *bulkhead*[2]
- Escalabilidade: Com um serviço grande e monolítico, temos que dimensionar tudo junto. Se uma pequena parte do sistema geral é limitado em desempenho, é preciso lidar com o dimensionamento de tudo como uma única peça. Com microsserviços menores menores, é possível dimensionar apenas os serviços que precisam de ajuste de escala, permitindo-nos executar outras partes do sistema em um hardware menor e menos poderoso.
- Facilidade de implantação: É possível fazer uma alteração em um único serviço e implantá-lo de forma independente do resto do sistema. Isso permite implantar o serviço mais rapidamente. Se um problema ocorrer, ele pode ser isolado rapi-

[1] *Service Oriented Architecture* - Arquitetura orientada a serviço - é um padrão de design de software onde múltiplos serviços colaboram para prover um conjunto de capacidades/funcionalidades (NEWMAN, 2015)

[2] O padrão *Bulkhead* é um tipo de design de aplicativo que é tolerante a falhas. Em uma arquitetura *Bulkhead*, os elementos de um aplicativo são isolados em *pools* para que, se um falhar, os outros continuem funcionando. Se um componente de um sistema falha, mas essa falha não ocorre em cascata, você pode isolar o problema e o resto do sistema pode continuar funcionando. (NEWMAN, 2015)

damente, facilitando uma rápida reversão. Facilitando a disponibilidade de novas funcionalidades aos clientes de forma mais rápida.

- Alinhamento organizacional: microsserviços ajudam a melhorar o alinhamento da arquitetura com a organização, minimizando a quantidade de pessoas trabalhando em uma base de dados para balancear o tamanho correto das equipes em relação a tamanho e produtividade.
- Composição: microsserviços possibilita que as funcionalidades criadas sejam consumidas de diferentes maneiras para diferentes propósitos de forma a aumentar a reusabilidade dos sistemas.
- Otimização de substituibilidade: Com pequenos microsserviços individuais, o custo de substituí- los ou deletá-los é muito mais fácil de ser gerenciado.

### 2.5.1 Padrões de projeto

Um padrão é uma solução reutilizável para um problema que ocorre em um contexto específico. É um ideia que tem suas origens na arquitetura do mundo real e que provou ser útil em arquitetura design de software. O conceito de padrão foi criado por Christopher Alexander, um arquiteto do mundo real (RICHARDSON, 2018). Os escritos de Christopher Alexander inspiraram a comunidade de software a adotar o conceito de padrões e linguagens de padrões. Na seção a seguir será revisado alguns dos padrões da literatura de microsserviços que serão utilizados na execução deste trabalho.

#### *2.5.1.1 Banco de dados por serviço*

Um dos principais padrões de projetos de microsserviços é o de um banco de dados por serviço. Manter vários serviços utilizando um mesmo banco de dados significa manter um monólito, o banco, na arquitetura que deveria ser desacoplada e flexível (REGAL, 2021). Utilizar bancos de dados diferentes para cada microsserviço elimina problemas de acoplamento, visto que mudanças em um banco de dados não afeta as outras aplicações. .Existem diversas formas de implementar esse padrão de projeto, e obviamente cada uma possui suas vantagens e desvantagens (REGAL, 2021). Um dos principais desafios vistos nesse tipo de padrão é como gerenciar transações que precisam da colaboração de diversos serviços (REGAL, 2021).

*2.5.1.2 Comunicação entre serviços*

Richardson (2018) afirma que microsserviços, muitas vezes, devem colaborar entre si. Por conta da autonomia de implantação e suas execuções acontecendo em diversas máquinas, eles precisam interagir utilizando *IPC*. Existem muitas tecnologias *IPC* diferentes para escolher. Os serviços podem usar mecanismos de comunicação baseados em solicitação/resposta síncrona, como HTTP, baseado em REST, gRPC ou SOAP. Alternativamente, eles podem usar comunicação assíncrona, baseada em mensagens, mecanismos de comunicação como AMQP ou STOMP. Há também uma variedade de formatos de mensagens. Os serviços podem usar formatos baseados em texto legível por humanos, como JSON e XML.

Existe uma variedade de estilo de comunicação cliente/servidor e eles podem ser divididos em duas dimensões:

- A primeira é se a comunicação é um-para-um (cada requisição do cliente é processada por um único serviço) ou um-para-muitos (cada requisição do cliente é processada por vários serviços)
- A segunda dimensão define se a comunicação é síncrona (cliente espera por um tempo fixo a resposta do serviço e pode se bloquear enquanto isso) ou assíncrona (o cliente não se bloqueia e a resposta não necessariamente será recebida imediatamente).

Também existem diferentes tipos de de interações um-para-um e um-para-muitos, sendo elas:

- Um-para-um:
    - Requisição e resposta. O cliente faz uma requisição ao serviço e espera uma resposta. Geralmente gera acoplamento entre serviços por conta da dependência.
    - Requisição e resposta assíncrona: O cliente faz uma requisição ao serviço, porém não é bloqueado para esperar a resposta.
- Um-para-muitos:
    - *Publish/Subscribe*: Cliente publica uma mensagem que é consumida por zero ou um mais serviços.
    - *Publish/async response*: Cliente publica uma mensagem de requisição e espera um tempo específico por respostas.

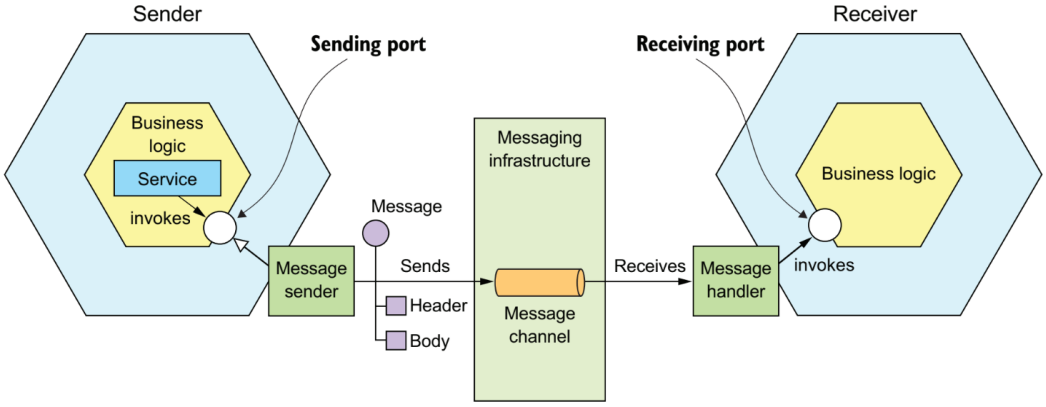
Tabela 2.1: Tipos de comunicação inter-processos

| | one-to-one | one-to-many |
|---|---|---|
| Synchronous | Request/Response | — |
| Asynchronous | Asynchronous Request/Response<br>One-way Notifications | Publish/Subscribe<br>Publish/Async Responses |

Source: (RICHARDSON, 2018)

Um tipo de implementação de comunicação entre serviços assincronamente é através do uso de um sistema de mensagens. Um sistema baseado em mensagens normalmente usa um agente de mensagens, chamado de *message broker*, que atua como um intermediário entre os serviços. Um *message broker* auxilia na comunicação entre serviços já que desacopla a comunicação entre os serviços e ainda por cima implementa recursos persistência de mensagens em caso de mensagens não entregues (GARTNER, 2022).

Figura 2.2: Arquitetura padrão de um *message broker*




Fonte: (RICHARDSON, 2018)

### *2.5.1.3 Observabilidade*

Possibilitar o diagnóstico do sistema enquanto o mesmo está em operação é extremamente importante em arquiteturas distribuídas, como as de microsserviços (NEWMAN, 2019). Algumas das técnicas citadas por Richardson (2018) para observabilidade são *Distributed Tracing* e *Log Aggregation*. O primeiro padrão se refere a uma identificação única das requisições, comumente chamado de trace id (ou id de correlação), que entram na arquitetura. Esse id deve ser mencionado por cada serviço que o receba e deve encaminhá-lo para cada outro serviço que ele se comunica durante aquela transação da requisição O segundo padrão descreve o uso de um repositório central de logs da aplicação que servirá como fonte de verdade para revisão dos mesmos durante *troubleshooting*

ou análise. Com a aplicação dos dois padrões descritos acima conseguimos um bom grau de nitidez sobre a situação das aplicações e suas comunicações, já que todos os logs estão centralizados e as comunicações entre os serviços estão identificadas com seus *ids* de correlação.

## 2.6 Domain Driven Design

DDD é apresentado por Evans (2003) como diversos conceitos sobre a separação do sistema em contextos limitados modelados na realidade do negócio que se necessita atender. Ele descreveu o seu sucesso de projetos de software em práticas, técnicas e princípios. A modelagem de domínios (com entidades, relacionamentos, restrições, interações e atributos) realizada juntamente com desenvolvedores, arquitetos, analistas de negócio e usuários é fundamental, porém não suficiente. É preciso garantir uma linguagem ubíqua para que especialistas técnicas e de domínio se entendam coerentemente e da mesma forma.

Por conta do objetivo do DDD, é possível afirmar que ele vai de encontro com a arquitetura de microsserviços, já que ambos tratam de definir limites para o negócio de forma a ser pensado independentemente do restante, facilitando o entendimento de todo o sistema por conta de serviços com objetivos (*boundaries*) bem delimitados.

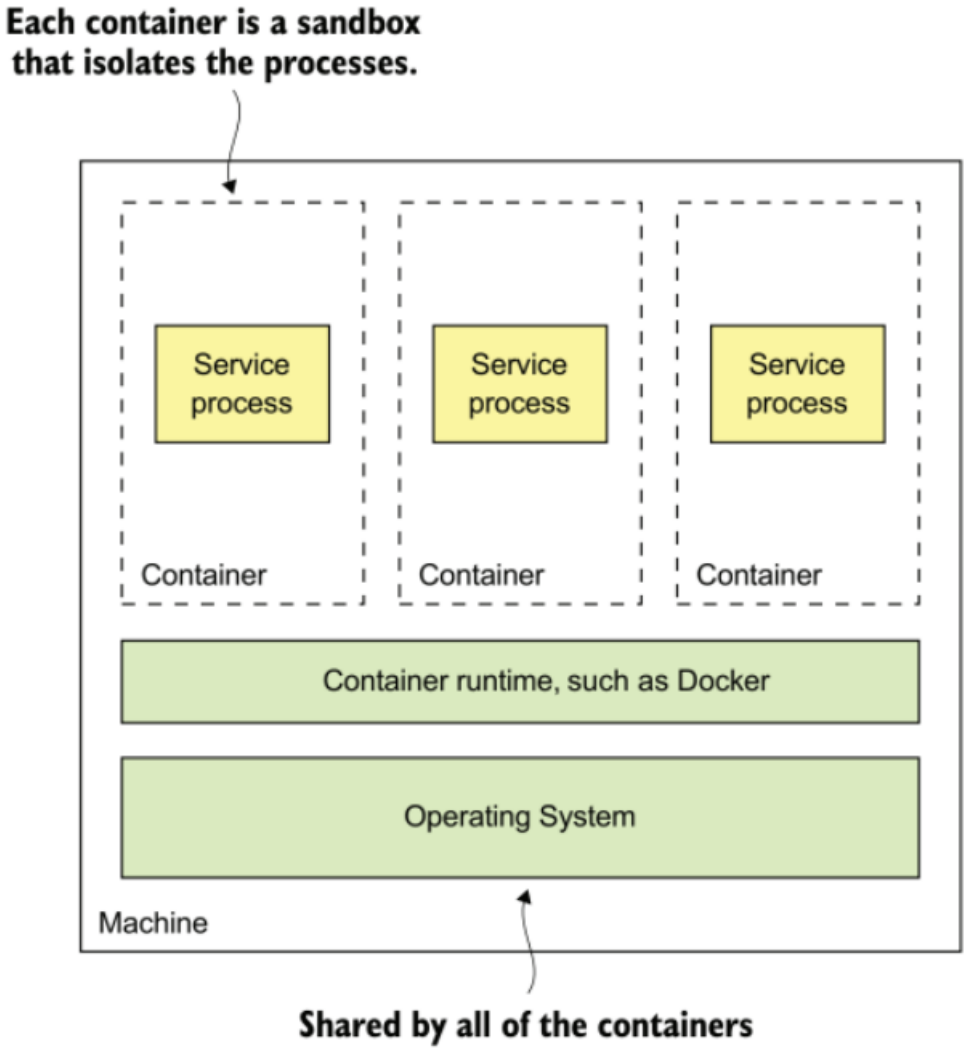
## 2.7 Contêineres

Richardson (2018) descreve contêineres como um mecanismo de implantação moderno e leve que se baseia na virtualização a nível de sistema operacional. Um contêiner pode consistir de inúmeros processos rodando no seu ambiente, onde cada contêiner isola os seus processos entre eles.

Um processo rodando dentro de um contêiner funciona como se ele tivesse sua própria máquina, onde ele possui um endereço IP próprio que elimina problemas de conflitos de portas de rede. O exemplo mais popular de contêineres é o *Docker*, porém existem outros, como por exemplo *Solaris*.

Contêineres também possibilitam especificar o uso de recursos de hardware, como CPU, memória e, dependendo da implementação do contêiner, recursos de I/O. O *runtime* do contêiner gerencia esses limites de modo que os contêineres "roubem"recursos da má-

Figura 2.3: Exemplo de contêineres executando em uma máquina




Fonte: (RICHARDSON, 2018)

quina hospedeira (RICHARDSON, 2018).

Fazer a implantação de serviços em contêineres agrega diversos benefícios. Sendo alguns deles:

- Encapsulamento da pilha de tecnologias utilizada no contêiner;
- As instâncias do contêiner são isoladas;
- Recursos do contêiner são controlados e limitados;
- Fáceis e leves de construir e implantar.

Claramente o uso de contêineres possui algumas desvantagens, principalmente por conta da necessidade de gerenciamento das instâncias e máquinas envolvidas nas operações. O gerenciamento de infraestrutura é um tópico bem importante e geralmente é um dos fatores que fazem equipes com menos capacidade de gerenciamento de infraestrutura

fujam dessa tecnologia (RICHARDSON, 2018).

## 2.8 Kubernetes

De acordo com Richardson (2018), Kubernetes é um framework de orquestração de contêineres. Esse framework trata cada máquina do cluster rodando a implementação de contêineres como uma pool de recursos. É preciso apenas descrever quantas instâncias do serviço são necessárias e o framework lida com o resto das configurações necessárias. Um Framework como Kubernetes possui três funções principais:

- Gerenciamento de recursos: Trata o cluster de máquinas como uma pool de recursos. Tornando a coleção de máquinas em uma única máquina;
- Gerenciamento de serviços: Garante que o número desejado de instâncias saudáveis esteja sendo executado em todas as vezes. Ele equilibra a carga das solicitações entre eles. A estrutura de orquestração executa atualizações contínuas de serviços e permite reverter para uma versão antiga sem comprometer os outros serviços;
- Agendamento: Seleciona qual máquina irá executar o contêiner.

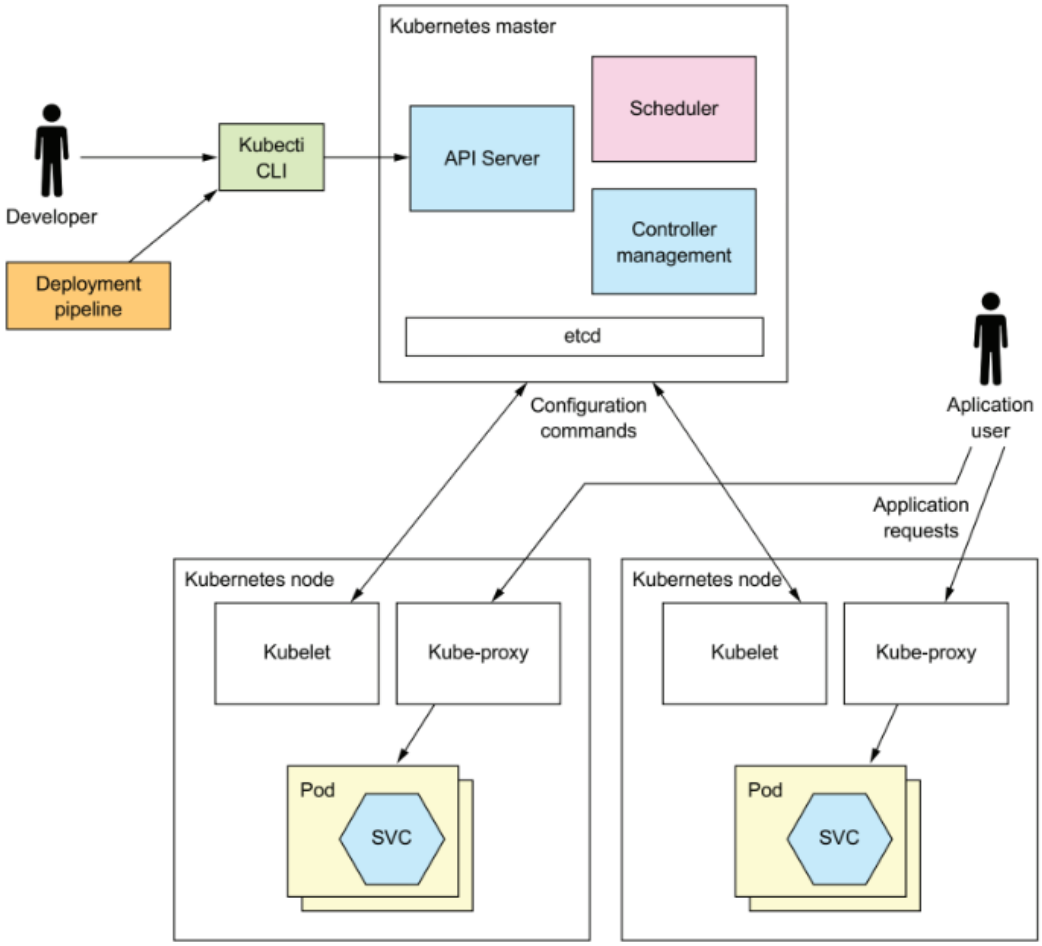
Máquinas em uma arquitetura de um cluster de Kubernetes podem ser ou *master* ou *nodes*. Uma máquina *master* é responsável por gerenciar o cluster. Uma máquina *node* executa os contêineres através do conceito de *pods*, que será visto adiante. Uma máquina *master* os seguintes componentes:

- *API Server*: API Rest usada para gerenciamento e implantação dos serviços nos clusters.
- *Etcd*: Um banco de dados NoSQL que armazena os dados do cluster.
- *Scheduler*: Seleciona um *node* para executar um *pod*.
- *Controller Manager*: Garante que o cluster esteja no estado desejado, como por exemplo a controladora *replication*, que gerencia corretamente o número de instâncias do serviço no cluster estão rodando através de comandos de iniciar ou remover instâncias.

Uma máquina *node* executa os seguintes componentes:

- *Kubelet*: Cria e gerencia os *pods* no cluster.
- *Kube-proxy*: Gerencia a rede, incluindo o balanceamento de carga entre os *pods*.
- *Pods*: Os serviços da aplicação.

Figura 2.4: Diagrama da estrutura de gerenciamento do Kubernetes




Fonte: (RICHARDSON, 2018)

Richardson (2018) também descreve os conceitos principais da arquitetura de implantação de serviços no Kubernetes:

- *Pod*: É a unidade básica de implantação. Consiste de um ou mais contêineres que dividem o mesmo IP e volumes de persistência. É uma estrutura efêmera e não deve possuir estado.
- *Deployment*: É uma especificação declarativa do *pod*. Ele é uma controladora que garante que o número desejado de instâncias do serviço estejam rodando a todo tempo. Cada *deployment* define um microsserviço. Usualmente é definido na linguagem de serialização de dados *yaml*.
- *Service*: Gerencia uma rede estática/estável para algum *deployment*. Um serviço tem um endereço IP e um DNS que aponta para o endereço do serviço e faz balanceamento de carga de tráfego TCP/UDP para os pods que ele está relacionado. O endereço IP e o DNS são acessíveis apenas dentro do cluster do Kubernetes. É

possível configurá-los para serem acessíveis por conexões externas.

- *ConfigMap*: Coleção de pares que definem configurações para um ou mais contêineres. Existe um tipo especial de *ConfigMap* que é usado para guardar informações sensíveis, como por exemplo senhas ou tokens de acesso, chamado de *Secret*.

Mais informações sobre Kubernetes e tutoriais trabalhar com o framework podem ser encontrados no livro da bibliografia referenciada neste trabalho[3] ou na documentação oficial[4] .

## 2.9 Estratégias de migração adotadas na literatura

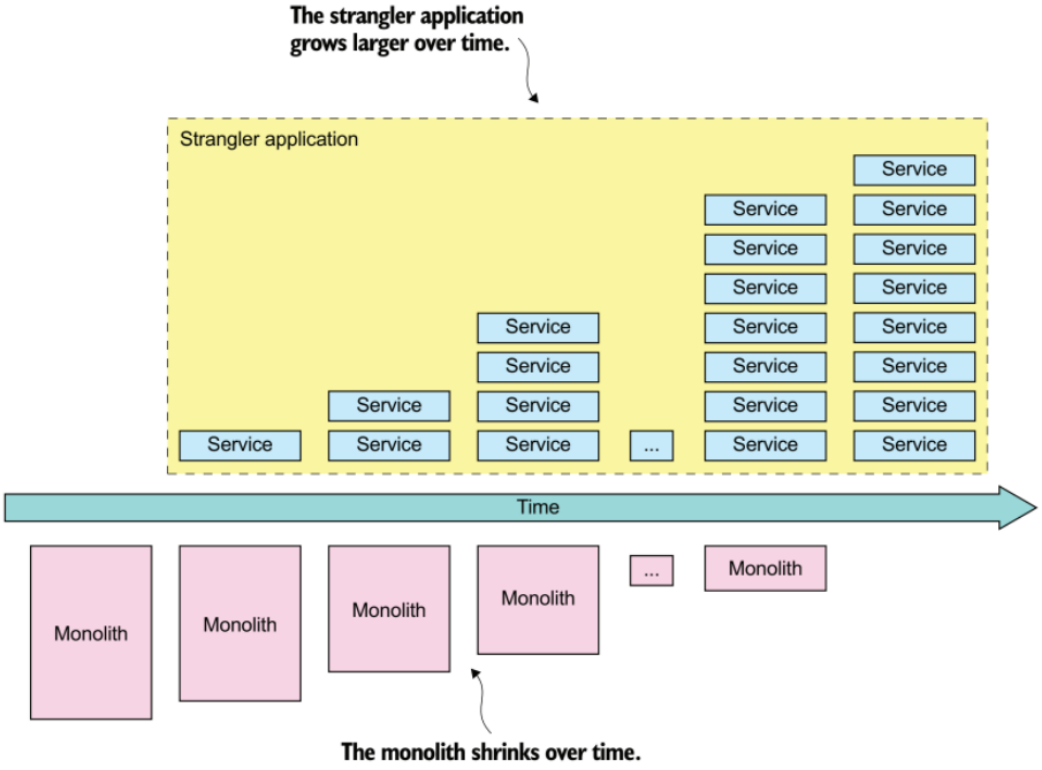
O processo de migração de uma arquitetura monolítica para uma arquitetura de microsserviços é uma forma de modernização de software. Essa modernização consiste basicamente no processo de converter uma aplicação legado para uma nova arquitetura mais moderna. Modernização de aplicações vem sendo feito a décadas e diversos ensinamentos podem ser tirados dessa experiência. Uma grande lição aprendida é que não se é recomendado fazer um *big bang rewrite*. Um *big bang rewrite* é o processo de desenvolver uma aplicação do zero, desconsiderando toda a base de código legado. Esse processo pode demorar meses ou anos, duplicando uma funcionalidade já existente, e apenas após isso é possível começar a adicionar novas funcionalidades ao sistema Richardson (2018).

Por conta disso, é recomendado fazer refatorações do monólito de maneira incremental. Dessa forma, os microsserviços trabalham em conjunto ao sistema legado até o momento em que todas as funcionalidades forem migradas e o sistema legado for descomissionado Richardson (2018).

Fowler (2004) se refere a esse tipo de migração proposta como *Strangler application pattern* (nome inglês derivado da planta videira estranguladora). Basicamente a analogia feita é de que essa planta (microsserviço) cresce em volta de uma árvore (aplicação legado) na intenção de chegar à luz solar acima da copa floresta (implementando funcionalidades do sistema legado). Após um tempo, a árvore ou morre por conta de sua idade avançada ou estrangulada (*strangled*) pela videira, deixando apenas uma videira em formato da árvore (microsserviços que em conjunto, substituem todo o sistema legado) (RICHARDSON, 2018). Newman (2019) descreve esse padrão de migração como tendo um início mais fácil e mais viável. Quanto mais externas ou secundárias as funcionalida-

[3] Richardson (2018)
[4] https://kubernetes.io/docs/home/

Figura 2.5: Exemplo de uma migração utilizando *Strangler Application Pattern*




Fonte: (RICHARDSON, 2018)

des a serem migradas, menor será a complexidade porque existirão menos acoplamentos. Por outro lado, quanto mais a migração se aproxima de funcionalidades do core do sistema/negócio, mais difícil tende a ser migração por conta de um alto nível de dependências, desse modo, análogo às raízes da árvore vítima da videira.

## 2.10 Trabalhos relacionados

Esta seção irá abordar e discutir os trabalhos relacionados ao trabalho em desenvolvimento.

### 2.10.1 Uma proposta de conversão de arquiteturas monolíticas para microsserviços visando redução de acoplamentos

O trabalho de Regal (2021) abordou a busca por uma forma ou sequência de passos para migrar uma arquitetura monolítica para outra orientada a microsserviços focando no uso de design patterns e topologia dos componentes envolvidos. As similaridades en-

tre o projeto mencionado e este Trabalho de Conclusão são a busca por desta sequência de passos para preparar e projetar uma migração de arquitetura monolítica para uma arquitetura de microsserviços. Ambos mencionam o uso do DDD como uma abordagem muito importante no processo de definição dos limites dos microsserviços. Apesar do primeiro focar nos design patterns e topologia dos componentes, o segundo, em contrapartida, busca entender mais profundamente os *trade offs* de tecnologia envolvidos neste processo.

### 2.10.2 Estratégia de migração de sistemas ditos monolíticos para arquitetura de microsserviços

Collioni (2018) faz uma proposta de migração muito mais enxuta e direta. Seu trabalho evidencia uma sequência direta de passos que podem ser seguidos, evidenciando pontos chaves, como por exemplo a conteinerização (através do *Docker*), processo de decisão dos domínios do microsserviço (DDD) e por fim o uso de um *message broker*. As semelhanças entre o presente trabalho e o trabalho de Collioni (2018) são a busca de uma sequência de passos a serem performados para fazer a migração da arquitetura monolítica para a arquitetura de microsserviços proposta. No entanto, o presente trabalho abre o leque de análises e faz discussões mais profundas sobre diversos aspectos que envolvem a migração. Desde a sequência de passo a passo até análises sobre a necessidade do uso de certas tecnologias ou padrões.

### 2.10.3 Como refatorar um monolítico em microsserviços

A Google (2021) desenvolveu um artigo bem completo sobre um passo-a-passo para planejar a migração de um sistema monolítico para uma arquitetura de microsserviços. O artigo se assemelha a esse trabalho em diversos aspectos, como por exemplo tópicos que precisam ser levantados e analisados para a migração, como que tipo de comunicação entre os serviços será usada, qual banco de dados e como extrair módulos (domínios) do monólito para desenvolver os microsserviços. O trabalho atual avança nas análises discutidas neste artigo e faz diversas outras, como por exemplo qual tipo de implantação será usada (*VM*, *serverless* ou Kubernetes), qual melhor linguagem de programação a ser utilizada e quais os motivos por trás de cada decisão tomada. O artigo

da Google (2021) é, de certa forma, uma generalização menos detalhada deste trabalho. Aqui serão feitas mais análises em relação a um estudo de caso específico e no artigo mencionado é detalhado um plano genérico de ação.

# 3 O SISTEMA A SER MIGRADO - SITUAÇÃO ATUAL

Será apresentado nesta seção os detalhes e características do sistema monolítico atual a ser migrado. Por fim teremos uma discussão sobre os itens analisados anteriormente e serão feitas algumas conclusões em relação ao sistema monolítico atual.

## 3.1 Apresentação do sistema presente

Este trabalho será baseado em um sistema real de e-Commerce do ramo de viagens norte americano no qual o autor faz parte como desenvolvedor. Por questões legais e de privacidade, dados sensíveis da empresa, clientes e informações que sejam confidenciais serão omitidas ou anonimizadas.

O ecossistema de aplicações de todo o sistema de e-Commerce é bem extenso e complexo. Diversas aplicações monolíticas se comunicam com uma única base de dados *on-premise*[1]. Este trabalho será focado no módulo de pagamentos da aplicação que se encontra em um monólito que é classificado como o motor de todo o e-Commerce. Existem aplicações satélites que se comunicam com este motor para diversos casos de uso, como por exemplo serviços de envio de e-mail e microsserviços que já foram concebidos através da quebra de pequenos módulos deste monólito.

### 3.1.1 Monólito principal

O monólito principal é uma aplicação *Java 7 EE*, desenvolvida e mantida a mais de 15 anos pela empresa e tem como foco principal ser o motor das regras de negócios de todo o e-Commerce. Esse motor possui diversas integrações com sistemas externos e aplicações satélites. A aplicação ainda recebe incrementos de funcionalidades ainda hoje e está em processo de deprecação.

Possui uma *API REST* para diversas integrações que o sistema possui com outras aplicações. Aplicações que fornecem serviços como apresentação de dados, métricas de clientes e serviços de envio de e-mail são alguns exemplos. A aplicação possui um sistema de autenticação *BASIC AUTH*, o que significa que o cliente da aplicação deve passar no *header* da requisição o ID e a senha concatenados pelo caractere dois pontos (:)

[1] Software que é instalado e executado em computadores nas instalações da pessoa ou organização que usa o software.

e codificados em base64 .

O módulo de pagamentos é um pacote dentro da aplicação contendo todas as classes referentes a gerenciamento de pagamento, transação e persistência de dados financeiros dos clientes. Esse módulo se comunica com outros dois módulos da aplicação: Checkout e Informação de Usuário. Seu funcionamento é essencial para o negócio, devido à responsabilidade de gerenciar todos os pagamentos que ocorrem no e-Commerce. Em vista disso, o módulo também possui uma integração *REST* com um gateway de pagamentos para realizar efetivamente as transações financeiras do e-Commerce como requisição de pagamento, reembolso e cancelamento. Preocupações de segurança e privacidade são tópicos muito importantes de serem endereçados neste módulo.

### 3.1.2 Banco de dados

O monólito se conecta a um banco de dados relacional legado que é mantido há mais de 20 anos dentro da infraestrutura de tecnologia da empresa. Este banco de dados funciona como uma base de dados única para todas as aplicações da empresa. Ele é uma base de dados Oracle que é usada como a única fonte de verdade dos dados de todas as aplicações. Por ser muito antigo, ele não recebe mais atualizações e atualmente se encontra na versão 12g - 2014 (a versão mais atual do banco de dados até o desenvolvimento deste trabalho é a 21c - 2022).

O banco de dados é organizado em *schemas* para cada grupo de aplicações. Existem mais de 30 criados, sendo que cada um possui em média mais de 100 tabelas. Existe um time de analistas de banco de dados que gerenciam as mudanças e fazem os scripts de atualização do banco de todas as aplicações.

## 3.2 Deficiências da aplicação

Processos de desenvolvimento não organizados e ultrapassados, uma arquitetura não escalável e falta de planejamento futuro das equipes de desenvolvimento foram catalisadores de problemas na aplicação e no banco de dados ao longo dos anos. As sub seções abaixo irão entrar em detalhes específicos de diversos pontos de vista técnicos, descrevendo e exemplificando diversos tipos de problemas existentes.

### 3.2.1 Padrões arquiteturais

Por se tratar de uma aplicação muito antiga, diversos tipos de padrões de arquitetura e desenvolvimento foram empregados ao longo dos anos. Ao fazer análise da antiga arquitetura e ter acesso a alguns dos códigos fontes, é possível notar alguns problemas relacionados a decisões precipitadas de *frameworks*, falta de consistência de acesso ao banco e o não uso de ferramentas de testes automatizados.

Como parte principal deste trabalho, um grande exemplo de decisão de arquitetura que não evoluiu bem com o decorrer dos anos é o monólito core analisado. Por ser considerado o motor de todo o e-Commerce, muitas funcionalidades foram desenvolvidas nessa base de código. Atualmente o monólito conta com mais de 5000 arquivos e mais de 2.5 milhões de linhas de código. Todo esse tamanho não é proporcional quando se refere a códigos de teste dentro da aplicação. A cobertura de testes de 10% é muito inferior à porcentagem de cobertura recomendada por especialistas de desenvolvimento de software (FOWLER, 2012b).

Diversas aplicações e módulos do monólito core acessam o banco de dados de maneiras diferentes e não padronizadas. Por ser muito antigo e sem um controle de qualidade de desenvolvimento adequado, é possível diagnosticar diversas violações de boas práticas de criação de entidades e relacionamentos, como falta de normalização das entidades. Algumas tabelas possuem modelagens de entidades e relacionamentos não usuais por conta de requisitos antigos que na época seria necessário grande esforço de refatoração para serem corretamente implementados. Além disso tudo, algumas tabelas são acessadas de maneira manual por diversas aplicações, sem haver uma maneira padronizada de acesso e controle de dados. Isso gera grandes inconsistências que precisam ser endereçadas de tempos em tempos pelos times analistas de banco de dados do cliente.

### 3.2.2 Coesão e Acoplamento

Embora os sistemas de uma empresa devam ser atualizados continuamente para refletir as práticas de negócios em evolução, a modificação repetida tem um efeito cumulativo na complexidade do sistema (COMELLA-DORDA et al., 2000). Um desses efeitos mencionados é o aumento expressivo no número de defeitos de software ao longo do tempo (KIRBAS et al., 2014). Também é comum nesses sistemas seus módulos ou métodos possuírem muitas responsabilidades e dependências fortes entre si. Isso causa

diversas complicações que precisam ser endereçadas ao longo do tempo (FEATHERS, 2004).

A aplicação em análise possui mais de 15 anos de existência e sua base de código é bastante amarrada e pouco coesa. Diversos times de desenvolvimento trabalharam nesta base, e muitas vezes simultaneamente. Como visto anteriormente, o processo de qualidade de código não era muito organizado, cada time de desenvolvimento tinha seus padrões e práticas de qualidade. Alguns, no entanto, não tinham essa cultura madura. Portanto existem diversos métodos ou classes que possuem muitas responsabilidades e dependências fortes entre si, tornando sua manutenção complexa e custosa, como veremos na seção abaixo.

### 3.2.3 Manutenção

A manutenção neste tipo de aplicação é bastante desafiadora e exige muito cuidado ao fazer manutenções no código. A grande quantidade de funcionalidades agrupadas, falta de cultura de teste e uma arquitetura não voltada para extensões geram dificuldade na hora de garantir que as modificações feitas na aplicação não produzam nenhum tipo de efeito colateral indesejado. O desafio de mudar código legado é preservar seu comportamento original e para isso é preciso de um feedback automático, ou seja, possuir uma suíte de testes automatizados na aplicação (FEATHERS, 2004). Porém isso nos leva a um dilema paradoxal: Para criar testes é preciso mudar o código, porém para mudar o código, é preciso criar testes. A solução para isso é tomar extremo cuidado ao fazer as modificações necessárias para testar o código e seguir uma receita: Identificar os pontos de mudanças, quebrar dependências, escrever os testes de proteção, fazer as mudanças necessárias para manutenção e por último refatorar o código.

Trabalhar numa base de código desta forma é bastante custoso e propenso a erros. É comum casos de manutenções em bases legadas gerarem erros completamente desconexos com o que foi mudado, devido ao grande acoplamento existente no código. Ao longo prazo esse cenário pode se tornar insustentável se não existir um processo muito bem definido de manutenção.

### 3.2.4 Performance e Escalabilidade

Por conta da infraestrutura *on premise* utilizada, o cliente necessita fazer toda a provisão de recursos físicos e virtuais para que a aplicação rode sem problemas de performance ou estabilidade. Por conta disso, a escalabilidade da aplicação está fortemente relacionada com a quantidade de servidores que a empresa possui. Caso a aplicação tenha algum problema de performance e estabilidade devido a um súbito aumento de acessos, é necessário que exista hardware disponível para que novas instâncias da aplicação sejam iniciadas para desafogar o tráfego. Também não existe nenhuma maneira automática e reativa de escalar a aplicação, pois é necessário fazer a implantação manual de novas instâncias. Mais detalhes e informações sobre como as implantações são gerenciadas serão descritas na seção 3.2.6.

Por ser um monólito, não existe possibilidade de fazer escalabilidade por módulos. Caso a aplicação esteja recebendo muitos acessos de busca de viagens, é necessário subir toda uma nova instância da aplicação. Ocupando processamento e espaço de memória com módulos que não são necessariamente o gargalo do tráfego. Ademais, por conta do grande tamanho da aplicação, é preciso de servidores potentes apenas para iniciar a aplicação monólito.

Outro ponto muito importante relacionado a esse tópico é a distribuição geográfica dos serviços oferecidos pela empresa. Sua área de atuação é continental, por conta da extensão do continente norte americano, desde a costa leste até a costa oeste. É um desafio manter tempos de latência curtos para todos os clientes por conta das grandes distâncias entre eles e os servidores que proveem o serviço do e-Commerce.

### 3.2.5 Disponibilidade

Como visto na seção de 3.2.4, existe muita dificuldade de escalar a aplicação. Não existe uma escalabilidade automática baseada em métricas e é preciso ter hardware disponível para novas instâncias da aplicação rodarem. Por esse motivo a disponibilidade da aplicação fica comprometida, visto que é preciso um trabalho manual de análise de uso e funcionalidade das aplicações. Caso alguma instância possua alguma falha, é necessário fazer o *reset* deste servidor e reiniciar a aplicação.

Apesar de existir algum nível de redundância na aplicação, ainda temos um *SPOF* (*single point of failure*). A existência de um *SPOF* grande é muito desencorajada em sis-

temas de alta disponibilidade por conta das consequências de falhas nesse sistema (DOOLEY, 2001). Caso algum problema que interrompa a conexão com esses servidores ocorra, como por exemplo falhas de rede ou internas da aplicação, todo o sistema é comprometido devido a não existência de um sistema de redundância em diferentes regiões ou servidores.

Por nossa aplicação analisada ser extremamente crítica para o negócio do e-Commerce, o tópico de disponibilidade é muito sensível e um dos grandes motivadores de migrar essa estrutura de uma arquitetura monolítica *on premise* para uma arquitetura de microsserviços na nuvem.

### 3.2.6 Implantação

Por conta da infraestrutura de servidores *on premise* é necessário um processo de implantação customizado para a realidade e estrutura disponível. O processo atual de implantação da aplicação consiste na geração dos arquivos executáveis através de uma pipeline automática gerenciada pelo Github. Após gerado os artefatos, é preciso realizar a implantação do mesmo nos servidores através de uma ferramenta legado.

A ferramenta usada para a implantação é muito antiga e inflexível. Para ser utilizada, a aplicação precisa de diversos arquivos de configuração específicos que aumentam a complexidade de desenvolvimento da aplicação e geram custos de manutenção. Além disso, toda a sua infraestrutura está localizada nos próprios servidores da aplicação. Caso algum problema ocorra com os servidores fica impossibilitada a execução de novas implantações.

Em caso de falhas nas implantações, o processo de *rollback* é completamente manual e depende de uma equipe dedicada interna do cliente para executá-lo. Além de adicionar uma complexidade extra, também aumenta os custos financeiros de manter uma equipe e infraestrutura dedicada para isso.

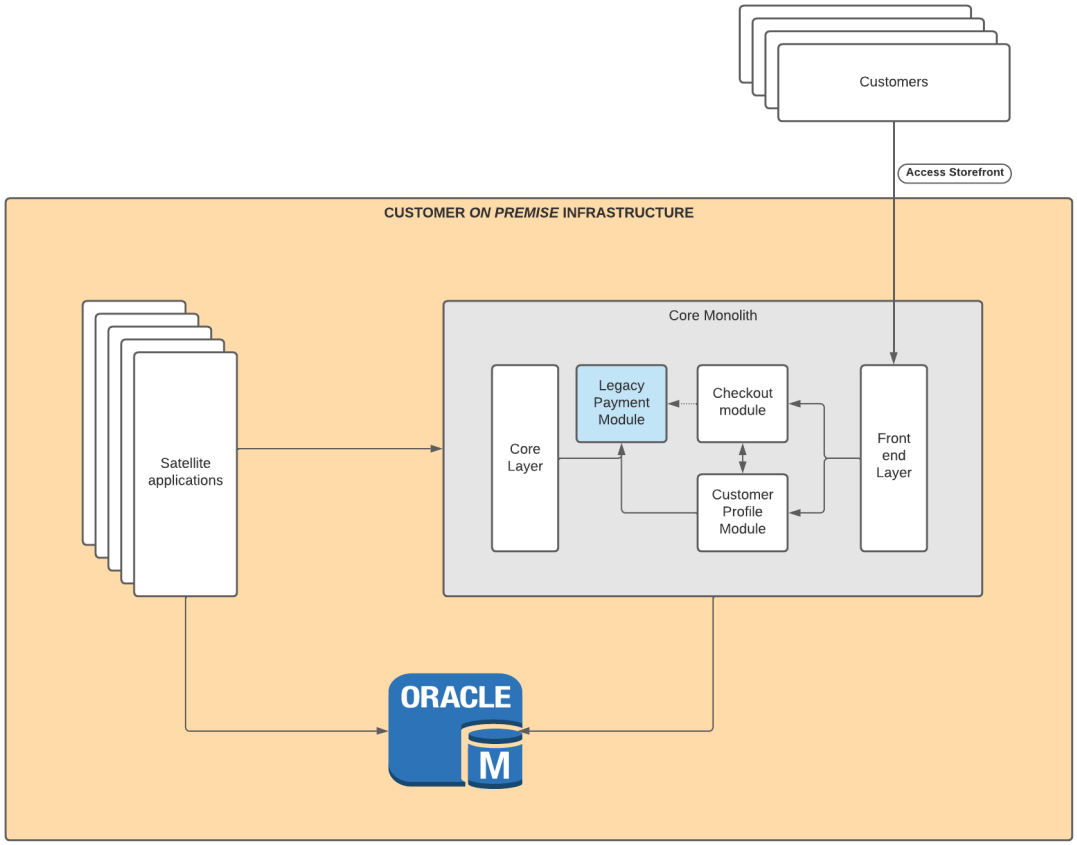
## 3.3 Discussão

É claro como a arquitetura monolítica e acoplada geram problemas de manutenção e qualidade dos desenvolvimentos. Um grande esforço de gerenciamento de pessoal e processos é necessário quando as implantações são executadas. Além do mais, a própria

qualidade da aplicação decai à medida que mais funcionalidades são agregadas.

Analisando as seções acima e suas argumentações, é possível afirmar que todo o processo atual de desenvolvimento e arquitetura gera muitos custos e uma grande complexidade de gerenciamento para o cliente. Existem diversos pontos de falha que podem ser resolvidos, ou pelo menos mitigados, através do uso de uma arquitetura de microsserviços em cima de uma infraestrutura de nuvem pública.

Figura 3.1: Diagrama de componentes da arquitetura atual




Fonte: Autor

# 4 PROPOSTA DE MIGRAÇÃO DO SISTEMA MONOLÍTICO

Por conta do grande tamanho da aplicação monólito e levando em consideração o tempo de desenvolvimento deste trabalho, foi escolhido fazer a migração de apenas um módulo específico. Foi escolhido o módulo de pagamentos, devido a sua grande importância para o negócio. Esse módulo servirá como exemplo base para as futuras migrações que poderão ser feitas no monólito. Veremos nas seções a seguir as propostas de migração do módulo de pagamentos e além da caracterização das propostas, será analisado também os motivos e o *trade off* envolvido em cada um dos pontos.

## 4.1 Expansão em monólito ou microsserviços

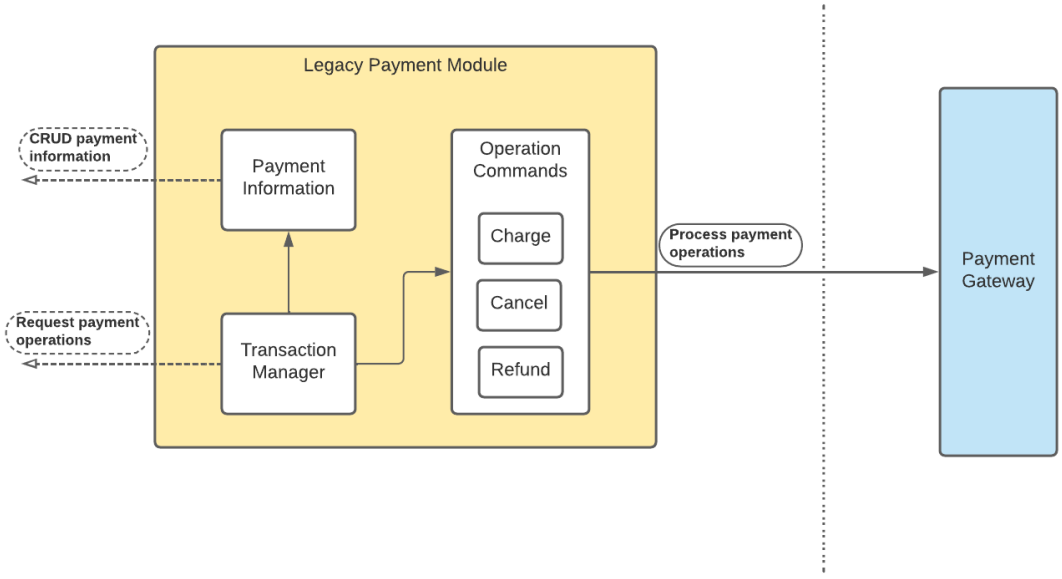
O mundo de desenvolvimento de software debate constantemente uma das questões que mais permeiam as discussões de arquitetura de softwares: Quando usar a arquitetura de microsserviços? Quando expandir uma arquitetura monolítica com microsserviços? Newman (2015) define que se deve possuir uma boa e justificável razão para desenvolver algo em microsserviços. Como visto nas seções anteriores, uma arquitetura de microsserviços apresenta muitos desafios de orquestração e controle dos mesmos. Esses custos devem ser menores do que os benefícios de se ter toda essa infraestrutura, não só de plataforma e serviços, como de times e conhecimento específico também. Por conta da arquitetura extremamente engessada e antiga, continuar expandindo a arquitetura atual pode se tornar um problema no futuro, visto a complexidade que o sistema atual já possui. A ideia de migrar para microsserviços também facilita a adoção de novas tecnologias, visto que as novas aplicações não estarão limitadas às tecnologias já empregadas no sistema monolítico.

Baseado nos dados acima, a decisão de expandir a aplicação de e-Commerce em uma arquitetura de microsserviços aparenta ser o melhor caminho a ser seguido. Visto o aumento expressivo de clientes esperados para os próximos meses, é necessário um código limpo, escalável e seguro. Além do mais, o uso de novas tecnologias acelera o processo de *go to market* de novas funcionalidades.

## 4.2 Domínio do microsserviço

Parte importantíssima do desenvolvimento de novos microsserviços é possuir seus domínios e bordas bem definidas. Como visto na seção 2.6, *domain driven design* funciona muito bem com microsserviços, dado a natureza de divisão de responsabilidades em pequenos serviços bem definidos e independentes. Usando a decomposição via *Strangler application pattern* vista na seção 2.9, onde a decomposição começa nas funcionalidades mais secundárias e externas, haverá uma menor complexidade no processo de migração das mesmas devido a um menor acoplamento de código e comportamento em relação a aplicação como um todo. Analisando a arquitetura atual da aplicação, é possível notar que existe um domínio específico para a realização das transações de pagamento do e-Commerce. Esse domínio comporta uma série de módulos que gerenciam as transações realizadas no momento do pagamento e também o controle das informações de pagamento dos usuários. Esse módulo será extraído da aplicação monolítica e será desenvolvido em cima de uma arquitetura de microsserviços. Nossa implementação a ser migrada, apesar de importante na aplicação, se encontra no final do processo de operação do negócio. Criando pouca dependência desta parte com o todo da aplicação e facilitando o processo de extração da lógica de negócio.

Figura 4.1: Diagrama de componentes do módulo de pagamentos do sistema monolítico




Fonte: Autor

### 4.3 Estimando dimensionamento de infraestrutura

Após a definição dos domínios dos microsserviços, a próxima etapa é fundamentalmente importante e servirá como base para todas as nossas próximas análises: Dimensionamento da infraestrutura. É a partir desta análise que sairão definições e informações sobre performance da aplicação, necessidade de hardware e ferramentas que auxiliarão nossa aplicação a lidar com a quantidade de dados esperada.

Uma métrica muito básica e amplamente usada é a de quantidade de requisições por segundo que a nossa aplicação terá que lidar. Como visto na descrição atual do sistema, o nosso e-Commerce possui uma média estimada de 400 milhões de usuários por mês, podendo aumentar para 500 milhões em épocas festivas ou de promoções. Com base nesse valor é possível estimar, fazendo algumas suposições, a quantidade de compras efetuadas por dia e, consequentemente, por segundo. Por ser um trabalho de estimativa e impreciso, é preciso levar em conta esta imprecisão ao fazer os cálculos de dimensionamento.

Se tomarmos como suposição que a taxa de conversão, ou seja, a taxa de usuários que efetivamente finalizam alguma compra no e-Commerce seja de 10%, teremos uma média de 40.000.00 de compras finalizadas por mês. Para obtermos a média por segundo de transações esperadas, seria possível fazer uma simples conta dividindo o total de compras finalizadas pela quantidade total de segundos em um mês. Ainda que seja uma boa estimativa, ela ainda é muito rasa e não leva outras variáveis em consideração, como por exemplo dias e horas de pico.

A maioria das transações de um e-Commerce ocorrem em algum horário de pico. Ao adicionarmos essa variável à equação, nossas estimativas já começam a se tornar mais precisas e realistas. Vamos assumir que a maior parte das compras sejam feitas no início do mês por conta do recebimento dos holerites. E também que o melhor dia de compra seja no domingo. Além disso, o horário de pico é às 20h por conta da grande quantidade de pessoas em suas casas. Se assumirmos que 50% das compras são efetuadas nesse momento, teremos a seguinte equação:

$$\alpha = \frac{40.000.000 * 0,5}{3.600} \tag{4.1}$$

Temos como resposta o valor arredondado de 5.600 de requisições de pagamento efetuadas por segundo. Este número pode parecer baixo, e faz sentido, pois não adicionamos à nossa equação o grau de incerteza e os acessos às informações de pagamento que

o nosso microsserviço também expõe.

Existem outras requisições feitas ao nosso microsserviço que não foram levadas em consideração ao fazer nossas estimativas iniciais. Quando é feita uma requisição de transação de pagamento, é preciso primeiramente ler/gravar dados de pagamento. Assim podemos inferir que para cada transação é necessário 3 requisições pelo menos. Baseado nessa premissa, nosso número estimado de requisições por segundo salta de 5.600 para 17.000.

Mesmo com todas essas análises, ainda faltam inúmeros fatores que também precisam ser estimados, como por exemplo a quantidade de usuários que estão atualizando dados de cartão no momento de pico ou até ataques de negação de serviço que nosso microsserviço possa receber. Uma forma de lidar com essas informações de natureza incerta é adicionar um fator de incerteza ao nosso cálculo. Se assumirmos que é possível que nossa demanda seja 5 vezes maior do que o esperado, nossas estimativas já atingem 85.000 requisições por segundo.

Apesar de todas essas suposições não nos garantirem uma certeza na carga processada pelo nosso microsserviço, elas nos ajudam a estressar diversos cenários em que a aplicação precise suportar. Além disso, é de grande auxílio para projetar as próximas etapas analisadas, como banco de dados na seção 4.5, implantação na seção 4.7 e até mesmo, como veremos na seção 4.4, as linguagens de programação adequadas.

## 4.4 Linguagem de programação

Assim como a definição das bordas e domínio do microsserviço são parte importantíssima do planejamento e geram muitos debates, a escolha da linguagem de programação utilizada no desenvolvimento também é alvo de diversas discussões. A regra de ouro é que não existe bala de prata, e as linguagens de programação não estão fora disso. Quando se está escolhendo uma linguagem de programação para microsserviços, é importante se atentar a alguns pontos como cultura de automação, observabilidade, descentralização de componentes e capacidade de esconder detalhes de implementação (GARAGE, 2019). Esses pontos ajudam a manter um microsserviço enxuto, de fácil manutenção e escalável, além de garantir o bom funcionamento do serviço através de métricas e informações que o serviço produz. Além disso, é interessante ter em mente aspectos de performance da linguagem escolhida, como consumo de memória, CPU e velocidade de processamento de requisições.

Para iniciar nossa análise, foram escolhidas três linguagens de programação que são muito utilizadas em desenvolvimento de microsserviços atualmente: Python, Java e Javascript (UPADHYAY, 2021). Serão analisados os pontos descritos anteriormente para embasar o processo de escolha da linguagem, assim como explicitar seus prós e contras em relação ao caso de uso do nosso microsserviço.

### 4.4.1 Python

Se pensarmos em velocidade de prototipação e versatilidade, Python tem vantagens se comparado com outras linguagens e *frameworks*. Sua facilidade de aprendizado, pouca verbosidade, facilidade de adicionar mudanças e testar rapidamente, devido a sua natureza interpretada, o tornam uma das grandes escolhas dos desenvolvedores para seus microsserviços. Um dos seus pontos fortes é a facilidade de manutenção do código, devido a facilidade de escrita e leitura do código. Outros pontos interessantes são seu grande suporte a outras tecnologias, como suporte a linguagens legadas como ASP e PHP e relevantes *frameworks* web rápidos e de baixo custo computacional como o Flask.

Entretanto, Python possui alguns pontos negativos relacionados à performance e testagem. Por ser uma linguagem interpretada, a linguagem possui menos velocidade de execução se comparada a linguagens compiladas com C, C++ ou Go. Além disso, sua natureza de tipagem dinâmica, em conjunto com a interpretação, dificultam a prevenção de erros. Aumentando o tempo de testagem necessário para garantir um código seguro e correto. Mais informações sobre Python podem ser encontradas na sua documentação oficial[1].

### 4.4.2 Go

Go se destaca com seu grande suporte a concorrência e *web services* nativamente. Adicionando uma sintaxe simples e poderosa, produtos escritos em Go suportam grande carga de trabalho e são fáceis de serem entendidos por outros desenvolvedores. A linguagem é escolhida por grande parte dos desenvolvedores para novos micro com alta carga de execução. Além disso, seu sistema de pacotes viabilizam a implementação rápida de produtos, evitando a criação e manutenção de *frameworks*. Por fim, a linguagem foi

[1] https://docs.python.org/3/

concebida exclusivamente para arquitetar sistemas grandes e complexos.

Apesar de ser uma linguagem poderosa e simples, Go possui alguns pontos negativos. Por ser uma linguagem nova e ainda em desenvolvimento, por vezes é difícil fazer uso de seus *frameworks*. Também é interessante pontuar que Go possui ponteiros em sua sintaxe, o que pode ser um problema na mão de desenvolvedores não muito experientes com gerenciamento de ponteiros e referência de memória. Mais informações sobre Go podem ser encontradas na sua documentação oficial[2].

### 4.4.3 Java

Java é uma das linguagens mais populares do mundo e ocupa a terceira colocação como a linguagem mais popular (TIOBE, 2022). Com seus *frameworks* de microsserviços robustos, grande comunidade, maturidade de desenvolvimento da linguagem e grande quantidade de desenvolvedores que a conhecem, Java figura como uma grande escolha para microsserviços. Spring Boot é dito como um dos frameworks de web services mais completos dentre todas as linguagens. Com grande suporte a tecnologias e ferramentas do mundo em nuvem, como Sleuth para *tracing id*, Actuator para API's de *health check*, cache, *frameworks* de acesso à banco de dados dentre diversos outros mecanismos valiosos para aplicações em microsserviços, como *circuit breakers* e métricas.

Java não é uma bala de prata e não é a escolha certa para todos os projetos. Apesar de seu uso muito flexível, a verbosidade da linguagem muitas vezes incomoda desenvolvedores mais acostumados a linguagens simples e de fácil interpretação, como Python. Além do mais, é uma linguagem pesada se comparada às outras e muitas vezes é dispensada de microsserviços ou desenvolvimentos mais pontuais ou exploratórios. Mais informações sobre Java podem ser encontradas na sua documentação oficial[3].

### 4.4.4 Discussão sobre a linguagem mais apropriada

Como descrito na seção de descrição atual do sistema, a necessidade de alta escalabilidade e performance é imprescindível na nossa aplicação. Por se tratar de uma parte vital ao negócio do cliente, o microsserviço deve ser resiliente e apresentar diversos recursos de métricas e observabilidade. Além disso, a facilidade de sustentar a aplicação

[2]https://go.dev/doc/
[3]https://docs.oracle.com/en/java/

também deve ser algo a ser considerado.

Python se encaixa bem quando o assunto é facilidade de escrita e manutenção, porém peca bastante em performance. Go, por outro lado, é bastante performático e poderoso para aplicações críticas, ainda mantendo uma certa facilidade de manutenção e escrita. Apesar disto, por ser uma linguagem ainda nova, existe certa dificuldade de se encontrar ou formar desenvolvedores de Go. Java consegue prover estas três necessidades de forma muito satisfatória. Com sua performance *multi-threading*, aproveitando de dezenas de *frameworks* consolidados para microsserviços, Java é uma ótima escolha para sistemas críticos. Seu suporte a observabilidade e métricas muito consolidado é um diferencial. Além disso, a facilidade de encontrar programadores experientes em Java pesa na escolha, ainda mais se considerarmos o atual momento de grandes movimentações e aquecimento do mercado de desenvolvimento de sistemas. Com base nessas informações o nosso microsserviço será escrito em Java, utilizando o framework Spring Boot para montar o *web service*.

## 4.5 Banco de dados

Durante a fase de arquitetura de um novo microsserviço, é comum se deparar com a pergunta: Qual banco de dados será utilizado pela aplicação. A arquitetura de microsserviços nos possibilita escolher diversas tecnologias de persistência de dados, essa liberdade de escolha é chamada de persistência poliglota (FOWLER, 2012a). Muitas vezes, equipes de desenvolvimento resumem esta escolha à decidir entre banco de dados relacionais e banco de dados não relacionais. Se dá menos importância, no entanto, a outras características muito importantes para um bom desempenho da aplicação. É muito importante levar em consideração outros aspectos atrelados ao banco de dados, como por exemplo réplicas e sua distribuição geográfica, sistema de recuperação, escalabilidade vertical/horizontal (CORBETT et al., 2013). Os critérios relevantes para a nossa análise se concentram em disponibilidade, consistência e particionamento, sendo estas três características conhecidas como o teorema CAP (BREWER, 2000).

Apesar de diferentes tipos de banco de dados, como bancos tabulares, em grafo, chave-valor e etc, a nossa análise irá se centrar entre banco de dados relacionais (SQL) e não relacionais (NoSQL) devido ao seu maior uso comercial em microsserviços, principalmente relacionados a dados de pagamento e transacionais como os do nosso microsserviço a ser construído (STATISTA, 2022).

### 4.5.1 Análise comparativa

A tabela a seguir terá descrições das características de cada banco e como isso é relevante, ou não, para a proposta de migração.

#### *4.5.1.1 Teorema CAP*

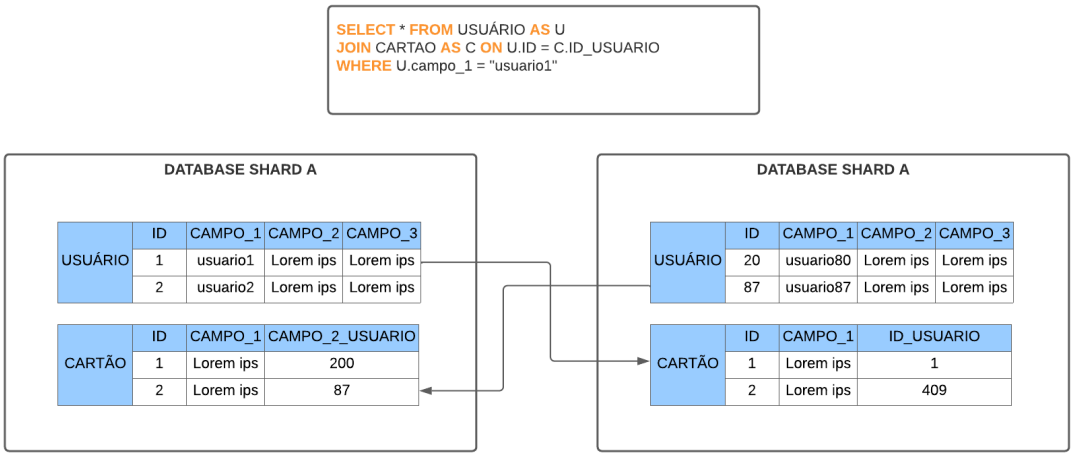
O teorema de Brewer constata que é impossível garantir disponibilidade e consistência ao mesmo tempo em um sistema distribuído tolerante a partições (BREWER, 2000). Os modelos de transação de banco de dados ACID e BASE se diferem na maneira em que eles lidam com esse problema.

O conjunto de propriedades de transações de banco de dados ACID tem como objetivo garantir a consistência dos dados da aplicação. Em contrapartida, o modelo BASE se foca em garantir alta disponibilidade. Banco de dados relacionais implementa nativamente o modelo de transação ACID, enquanto bancos de dados não relacionais implementam o BASE. Existem bancos de dados não relacionais que garantem certo grau de ACID, porém a sua abordagem de gerenciamento vai contra os princípios desse modelo, os tornando não muito recomendados em cenários de altíssima consistência de dados (REDMOND; WILSON, 2012).

#### *4.5.1.2 Escalabilidade*

Bancos de dados relacionais naturalmente se beneficiam de escalabilidade vertical, por conta das informações vistas na seção 2.2. Apesar disso, esse modelo, por natureza de seu gerenciamento interno, é muito mais difícil de escalar horizontalmente. Essa dificuldade está fortemente relacionada com as relações dos dados de chave estrangeira e chave primária. Os bancos relacionais não são projetados, nativamente, para trabalharem distribuídos. As operações de junção são caras em termos de processamento e ficam muito piores quando se é necessário gerenciar chaves estrangeiras e chaves primárias em bancos distribuídos (CHAUDHRY; YOUSAF, 2020). Não menos importante, o fator de atraso de tráfego de rede em ambientes distribuídos também pesa bastante. Um exemplo seria a leitura de informações de cartão de um usuário X. A tabela de usuário se encontra no *shard* A e a tabela de cartões se encontra no *shard* B. A cada leitura de informações de pagamento deste usuário, o *shard* A deverá adicionar uma cláusula JOIN que fará uma requisição de rede para o *shard* B e então fazer o cruzamento dos dados pedidos.

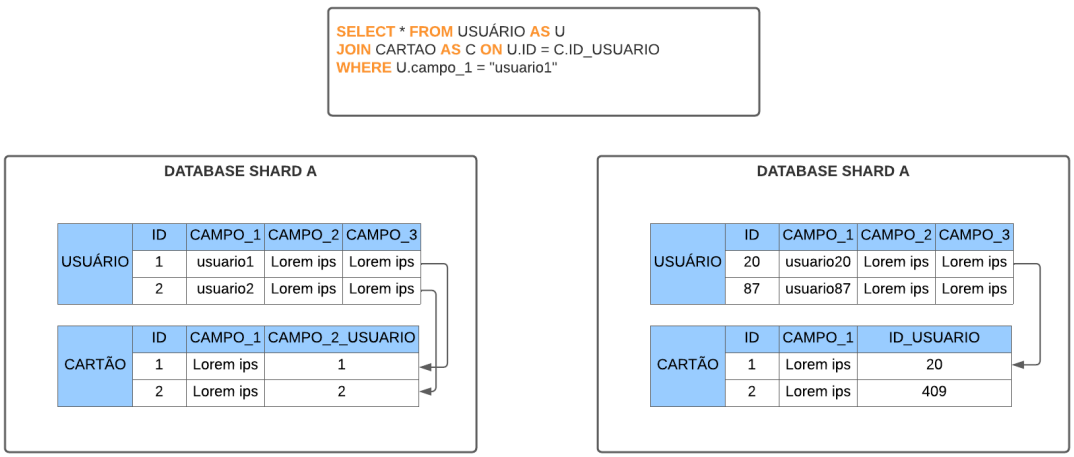
Figura 4.2: Tabelas em bancos distribuídos sem gerenciamento de *hotspots*




Fonte: Autor

Apesar deste problema, os provedores de nuvem possuem soluções de bancos de dados relacionais que resolvem parcialmente esse problema. A solução envolve persistir tabelas e dados relacionados na mesma instância do banco de dados distribuído. Isso diminuiria a necessidade de junções e gerenciamento das chaves primárias e estrangeiras sobre a camada de rede, diminuindo o fator tempo de latência de rede no processamento do banco (GOOGLE, 2022b).

Figura 4.3: Tabelas em bancos distribuídos com gerenciamento de *hotspots*




Fonte: Autor

Por conta da necessidade de orquestrar todas essas comunicações via rede, esses serviços de banco de dados são mais caros do que bancos relacionais não distribuídos ou bancos não relacionais distribuídos.

Os bancos de dados não relacionais, por sua vez, possuem grande facilidade de

escalabilidade vertical (naturalmente) e horizontal . Isso se deve ao fato de não existirem relações entre os dados, excluindo a necessidade de gerenciamento de chaves primárias e estrangeiras e facilitando a distribuição dos dados em diferentes bancos (FOWLER, 2012a). Isso simplifica muito o gerenciamento dos dados distribuídos, por consequência aumentando e muito a performance desses bancos nesse modelo de escalabilidade.

### 4.5.2 Discussão sobre a escolha do banco de dados

Nosso microsserviço é extremamente crítico para o e-Commerce, visto que lida com informações de pagamentos e transações dos clientes. Nossa aplicação precisará de réplicas de leitura, devido a um grande número de requisições que o sistema irá receber para devolver as informações de transação e pagamento. Como visto nas seções 3.2.4 e 4.3, a aplicação sofre de latência devido a sua área de atuação no país e grande quantidade de carga de requisições que precisa gerenciar, que cobre todos os estados dos Estados Unidos. Por conta disso, é necessário ter uma distribuição geográfica de pelo menos uma réplica em cada costa, para fornecer um serviço de menor latência para todos os clientes independente de sua localização no país.

Apesar da alta disponibilidade do banco de dados ser uma característica importante para a nossa aplicação, devido a as perdas de receitas em cenários de indisponibilidade, a falta de garantia de consistência dos dados é, obviamente, um fator impensável em aplicações financeiras. Falhas neste cenário seriam muito mais problemáticas do que alguns momentos de indisponibilidade do banco. A necessidade de um banco de dados consistente, nesse caso, possui muito mais peso do que uma facilidade de escalabilidade horizontal e de menores custos. Assim, a escolha de um banco de dados relacional se torna indispensável.

## 4.6 Message Brokers

A arquitetura de microsserviços não está imune a erros inesperados ou de comunicação entre serviços. É possível dizer que a quantidade deste tipo de falhas aumenta significativamente neste tipo de arquitetura devido a grande quantidade de serviços se comunicando entre si e da necessidade de orquestração deste grande volume de dados (NEWMAN, 2015). Operações financeiras e processamento de pagamentos devem ser

enviados uma única vez e falhas de comunicação podem gerar erros de processamento, como por exemplo perda ou duplicação dessas informações. Pensando no nosso caso de uso, digamos que nosso microsserviço, que recebe milhões de requisições por dia para processar os pagamentos, está passando por alguma instabilidade de rede, ou o provedor de pagamento está fora do ar.

O uso de *message brokers* pode nos auxiliar a mitigar erros de comunicação por conta da sua natureza tolerante a falhas e assíncrona (KORAB, 2017). Usar um *message broker* para lidar com os dados de transações financeiras garante que essas informações não serão perdidas nem duplicadas acidentalmente, além de fornecer prova de recebimento e permitir que os sistemas se comuniquem de forma confiável, mesmo quando as redes intermediárias estão inativas ou instáveis. Além disso, *message brokers* podem possuir criptografia ponta a ponta, tornando a comunicação de dados sensíveis mais segura. Além dos benefícios de segurança e consistência vistos nas seções acima, maior flexibilidade de gerenciamento das respostas evita problemas de gargalo entre aplicações, visto que o *message broker* é responsável por armazenar as mensagens a serem executadas pelos serviços, sem a necessidade de integração ponto a ponto entre aplicações (KORAB, 2017).

Dado os casos de uso analisados, é claro os benefícios que o uso de um serviço desses trará para a nossa arquitetura. Existirá um *message broker* que receberá mensagens produzidas pela aplicação principal que representam as requisições de pagamento dos usuários. Nosso microsserviço por sua vez consome estas mensagens, efetua os pagamentos via integração com algum *gateway* de pagamento e produz novas mensagens que representam o status da transação do usuário. O uso de um *message broker* do tipo *Publisher/Subscriber* no nosso cenário gera vários benefícios de resiliência e consistência dos dados. Além de facilitar o gerenciamento das mensagens por outras aplicações.

## 4.7 Gerenciamento de implantações

Como visto na seção 3.2.6, a implantação da aplicação atual é bastante manual e custosa. Existem diversas abordagens para tornar este processo mais ágil, automático e barato. Serão avaliadas algumas mais conhecidas e recomendadas pelo mercado e profissionais de tecnologia. Após isso, será analisado como fazer o gerenciamento dessa aplicação. Conceitos como escalabilidade, disponibilidade, distribuição, gerenciadores de tráfego e observabilidade serão levados em consideração na nossa análise para fazer a

melhor escolha de abordagem de implantação.

Como todos os processos avaliados possuem suporte para integração e entrega contínua, este tópico não será abordado em detalhes, pois será inferido que tudo relacionado a isso já está garantido para todas as partes.

### 4.7.1 Abordagens de implantação

Esta seção irá abordar as diferentes formas de se gerenciar a implantação de uma arquitetura de microsserviços em ambientes de nuvem.

#### *4.7.1.1 Máquinas virtuais em uma nuvem pública*

Os provedores de computação em nuvem dispõe de máquinas virtuais que podem ser utilizadas para fazer a implantação de aplicações nos ambientes virtualizados das máquinas virtuais. Esse tipo de implantação é bastante básico e são uma forma comum de hospedar vários ambientes em um único servidor, mas utilizam muito mais capacidade de processamento, pois reservam parte do hardware do servidor para seu uso, mesmo quando não está em uso (GOOGLE, 2022a). Atualmente não é a maneira mais recomendada para fazer a orquestração de microsserviços por conta de sua característica simplista, porém poderosa. Abordagens mais poderosas e menos custosas são fundamentais quando se tratando de microsserviços irão suportar grandes cargas de tráfego e precisarão de um escalonamento mais preciso e controlado.

#### *4.7.1.2 Serverless*

Segundo Richardson (2018), a abordagem de *Serverless* é bastante recomendada em cenários em que é preciso haver uma rápida entrada no mercado. Por ter toda a parte de infraestrutura gerenciada pelo provedor de nuvem, acaba sendo muito mais simples e rápido fazer a configuração e executar a aplicação nesses ambientes. Normalmente uma aplicação básica precisa apenas seguir pequenos padrões de configuração interna para estar pronta para rodar em um ambiente *Serverless*. Em casos de não haver uso da aplicação, a flexibilidade de custeamento da plataforma viabiliza minimizar os custos quando não há grande tráfego de requisições passando pela aplicação. Também é possível escalar a aplicação automaticamente baseado na carga que a aplicação está recebendo.

Apesar dos pontos acima, a falta de controle da infraestrutura também pode ser tor-

nar um problema em aplicações que necessitam de configurações mais complexas e precisas de infraestrutura (escalonamento, recursos), rede (controle de comunicações, *proxies*), segurança (encriptação de comunicações, controle de acesso a aplicações) e observabilidade (capacidade de visualizar os estados internos de todas as aplicações e suas relações) (REDHAT, 2017). A aplicação fica presa às ferramentas disponíveis do provedor de nuvem, tornando a aplicação acoplada ao provedor e dificultando uma possível migração para outro provedor. Também existem limitações de linguagens a serem usadas. Cada provedor provê um conjunto de linguagens que são suportadas em suas plataformas *serverless* (REDHAT, 2017). Além disso, sua natureza voltada a eventos não é recomendada para aplicações com tarefas de longa duração ou aplicações de tempo real com altas cargas de tráfego por muito tempo, devido a custos por utilização/tráfego (RICHARDSON, 2018).

#### *4.7.1.3 Kubernetes*

Como visto na seção 2.8, Kubernetes possui um foco muito grande em flexibilidade de configuração e criação de uma estrutura grande e complexa de microsserviços. Por conta disso, a criação de uma pipeline de CI/CD para diversas aplicações se torna mais fácil e de integração mais transparente.

Também é preciso reforçar sua capacidade de configuração de requerimentos de segurança como políticas de acesso às aplicações, controle de tráfego, encriptação de informações sensíveis e diversos *frameworks* de observabilidade que já possuem integrações nativas com Kubernetes e microsserviços rodando em seus clusters. Também é possível controlar a visibilidade das aplicações para o mundo externo através de *proxies* e gerenciadores de permissões, e também as próprias comunicações entre os microsserviços. Tudo isso sendo observado por diversos *frameworks* de métricas e observabilidade que geram informações valiosas para os administradores da infraestrutura e das aplicações (RICHARDSON, 2018; KUBERNETES, 2022b).

Em vista dos pontos anteriores, Kubernetes é muito recomendado para migrações de aplicações monolíticas para microsserviços, por conta de todo o gerenciamento refinado que é possível graças à grande flexibilidade de configuração. Em termos de custos, Kubernetes acaba sendo mais barato que a abordagem *serverless*, pois não possui custo por uso, mas sim por infraestrutura dedicada (IVANOV, 2021). Aplicações possuem carga uniforme, previsível e alta ao longo do tempo acabam sendo mais vantajosas em Kubernetes (RICHARDSON, 2018). Por fim, também é facilitada a criação e uso de banco de

dados e outros tipos de aplicações, como *message brokers* e *proxies*, devido a componentes pré existentes e o uso de gerenciadores de recursos (KUBERNETES, 2022b).

Assim como a abordagem *serverless*, Kubernetes também possui alguns desafios que precisam ser levados em consideração. Sua grande flexibilidade gera uma grande necessidade de profissionais capacitados para gerenciar tais recursos e infraestrutura. Seu uso é recomendado para arquiteturas mais robustas e que necessitam de configurações mais refinadas, devido à complexidade de configuração e manutenção.

### 4.7.2 Discussão sobre abordagens de implantação

O uso de máquinas virtuais é interessante em aplicações *on premise* que estão sendo migradas para o mundo de computação em nuvem. Sua natureza extremamente manual de configuração não encaixa bem com a arquitetura de microsserviços. Não é uma abordagem que escala bem com o desenvolvimento de novos microsserviços.

Diferentemente, *serverless* possui grande poder para trabalhar com microsserviços. Sua capacidade de fácil configuração e flexibilidade de custo e escalonamento são diferenciais. Entretanto, nossa aplicação é extremamente crítica para a execução do negócio do e-Commerce. É preciso um controle mais refinado da infraestrutura usada. Conceitos como segurança e observabilidade são extremamente importantes e precisarão de configurações específicas, como por exemplo encriptar dados sensíveis de clientes e pagamento. Além disso, o controle de acesso a essa aplicação precisa ser muito bem definido, pois por tratar com dados de pagamento, é muito importante que apenas as aplicações necessárias possuam acesso a esta aplicação. Neste tipo de aplicação é muito importante seguir a política de segurança de menor privilégio necessário e Kubernetes possui os meios para isso. É importante lembrar que, como visto na seção 4.3, existe uma grande necessidade de gerenciamento de aumento de carga nos serviços, coisa que Kubernetes também resolve muito bem (RICHARDSON, 2018).

A migração do monólito para a arquitetura de microsserviços não termina com a aplicação analisada neste trabalho. Outras aplicações podem e serão migradas futuramente e uma infraestrutura de orquestração de contêineres flexível e segura é extremamente importante. Por tanto, a escolha de usar Kubernetes como infraestrutura de implantação e gerenciamento do microsserviço é a mais acertada para o contexto analisado, pois implementa todos os pontos críticos analisados acima.

## 4.8 Malha de serviços

Em ambientes de múltiplos microsserviços, um dos grandes desafios é o gerenciamento da comunicação entre eles. Para resolver este problema foi desenvolvida uma ferramenta de controle de comunicação apelidado de malha de serviços (tradução livre de *service mesh*). Uma malha de serviços funciona muito bem em ambientes com vários microsserviços, devido a complexidade de orquestração das comunicações (RICHARDSON, 2018).

Entretanto, seu funcionamento não é tão eficiente ou até mesmo útil em ambientes com poucos microsserviços, sendo este exatamente o caso analisado neste trabalho. Como teremos apenas um microsserviços se comunicando com um banco de dados e um *message broker*, o uso de uma malha de serviços neste contexto adicionaria uma complexidade muito maior do que a necessária. Portanto não será adicionado à nossa arquitetura uma ferramenta de malha de serviços. Os tópicos das seções 4.9 e 4.10 serão implementados utilizando abordagens mais simples e rápidas.

## 4.9 Observabilidade

Como visto na seção 2.5.1.3, observabilidade é parte fundamental em um ambiente de arquitetura de microsserviços. Monitoramento das aplicações em tempo real e com capacidade de rastreamento de falhas é fundamental para que as mesmas sejam identificadas, analisadas e resolvidas com rapidez e precisão. Como visto na seção de malha de serviços, nossa aplicação não contará com a ferramenta para garantir a observabilidade da aplicação. Com isso em mente será preciso implementar soluções mais simples e individuais para cada requisito.

### 4.9.1 Logs

Existem diversas ferramentas de mercado que implementam o monitoramento e registro de logs de uma aplicação Java. Apesar disso, algumas se destacam por conta de uma integração simples e muitas funcionalidades disponíveis para uso com baixa necessidade de configuração. A ferramenta de logs Splunk foi considerada líder de mercado no segmento oito vezes seguidas e é a ferramenta de controle e gerenciamento de logs mais

utilizada atualmente (SPLUNK, 2021). Isto, e mais a familiaridade do autor do trabalho com a ferramenta, a tornou a candidata escolhida para a ferramenta de monitoramento da arquitetura proposta.

### 4.9.2 Métricas

Além de monitoramento de logs da aplicação, é preciso saber em detalhes como anda o funcionamento da aplicação internamente. Métricas variam desde dados de processamento da aplicação como consumo de memória e processados, tempo de resposta de requisições até mesmo métricas customizadas para a realidade específica da aplicação. Esses dados são essenciais para entender o funcionamento dos microsserviços e entender possíveis causas de erros ou falhas (RICHARDSON, 2018). Existem inúmeras soluções no mercado para monitoramento de métricas em aplicações Java. Muitas delas estão atreladas a provedores de computação em nuvem, open source e softwares proprietários. Por existirem muitas possibilidades a análise deste tipo de solução tomaria muito tempo. Por conta do tempo e espaço limitado deste trabalho, uma discussão e análise comparativa deverá ser postergada para futuros trabalhos. Na nossa arquitetura usaremos New Relic como ferramenta de monitoramento de métricas por conta de uma integração simples e uma boa quantidade de funcionalidades que não requerem configuração extra.

### 4.9.3 Alertas

A configuração de monitoramento de logs e métricas não é suficiente para se ter uma boa observabilidade da aplicação. É inviável pensar em uma análise manual desses dados por conta do grande volume de dados coletados por essas ferramentas diariamente. É necessário uma abordagem automática de análise para que situações fora do normal ou que identifiquem possíveis falhas sejam alertadas aos times de desenvolvimento e sejam endereçadas o mais rapidamente possível. Existem diversas ferramentas no mercado que executam tais tarefas. Splunk e New Relic possuem sistemas próprios de alertas baseado em métricas, buscas e análise de dados customizados. Esses sistemas possuem integração com ferramentas de mensagens instantâneas, e-mails, ligações automáticas e diversas outras formas de contato.

Nossa arquitetura irá usar os alertas já embutidos no Splunk e New Relic para

identificar possíveis cenários de erros.

## 4.10 Gerenciamento de tráfego

O controle de quais aplicações possuem acesso ao cluster de microsserviços é parte importante das medidas de segurança e resiliência da aplicação. Garantir que apenas aplicações conhecidas e confiáveis possam se comunicar com o microsserviço permite menos pontos de falhas e tentativas de invasões. Como visto na seção 4.8, não será utilizado nenhuma solução de gerenciamento de tráfego, como por exemplo malha de serviços. Como forma de implementar este requisito na aplicação analisada, será utilizado os recursos padrões de listas de permissão oferecidos pelo Kubernetes.

## 4.11 Gerenciando escalabilidade e disponibilidade do microsserviço

Nesta seção será descrito e analisado as abordagens de escalabilidade e disponibilidade que podem ser empregadas em arquiteturas de microsserviços.

### 4.11.1 Escalabilidade

Como visto na seção 2.2, existem duas maneiras primárias de escalonamento de um microsserviço: horizontal e vertical. Por não ser relevante para este trabalho como gerenciar o consumo dinâmico de recursos de hardware, iremos analisar apenas a implementação de escalabilidade horizontal.

Existem duas formas de se implementar escalabilidade horizontal, manual e automática (MICHAEL et al., 2007). A abordagem manual é utilizada na forma de escalabilidade preventiva do microsserviço. Em situações como datas festivas ou de grandes promoções, como a Black Friday norte-americana, é comum preparar a sua infraestrutura para receber um grande número de acessos. Também é comum já preparar os microsserviços para um possível grande número de requisições a serem atendidas.

Apesar disso, é uma abordagem cada vez menos usada por conta de tecnologias de escalonamento automático de microsserviços que muitas arquiteturas de gerenciamento de contêineres já possuem. Infraestruturas de Kubernetes de diversos provedores de nuvem possuem tecnologias de escalonamento automático de microsserviços baseado em

métricas, como por exemplo uso de hardware dedicado, quantidade de requisições por minuto, previsão de carga e diversas outras que caso ultrapassem uma margem, irão ativar o mecanismo de escalonamento horizontal. Como Kubernetes usa por padrão métricas de recursos de hardware, como consumo de CPU e memória (KUBERNETES, 2022a), iremos utilizar estas mesmas métricas em nossa aplicação. O valor de margem utilizado pelo Kubernetes para fazer o escalonamento horizontal automático é bastante relativo e requer um refinamento ao longo da vida do microsserviço. Para fins de análise deste trabalho será utilizado o valor arbitrário de 50% do uso de recursos de CPU e memória.

### 4.11.2 Disponibilidade

É visto na seção 2.3 sobre alguns desafios de aplicações de alta disponibilidade. A arquitetura sugerida por este trabalho consegue parcialmente cobrir estes pontos. O uso de banco de dados distribuídos nos dá maior segurança de que os dados, em casos extremos de falhas, estão replicados e protegidos em outros locais físicos. Apesar disto, ainda nos falta evitar um ponto único de falha em relação ao nosso microsserviço e torná-lo, por sua vez, redundante.

Com o uso do conceito de multi-clusters do Kubernetes (KUBERNETES, 2022b), a execução do mesmo serviço em clusters em várias regiões proporciona melhor tolerância a falhas. Se um serviço em um cluster não estiver disponível, a solicitação poderá falhar e ser veiculada de outros clusters. Este mecanismo resolve o problema de um único ponto de falha, tornando nossa arquitetura redundante e tolerante a falhas.

## 4.12 Provedores de nuvem

Quando se está migrando para uma arquitetura de microsserviços, é recomendado o uso de computação em nuvem Google (2021). Os benefícios são inúmeros e podemos enumerar alguns deles: Como elasticidade e escalabilidade, alta disponibilidade, provisionamento automático de recursos, monitoramento, segurança da informação e backup dos dados.

Sendo nesse caso escalabilidade e alta disponibilidade os pontos principais da migração da arquitetura monolítica para microsserviços propostos neste trabalho. Ao decidir qual provedor de nuvem escolher, deve se atentar a diversos fatores que garantam quali-

dade e confiança a longo prazo. Durante esta análise, foram considerado os seguintes pontos:

- Saúde Financeira - O provedor deve manter um registro da estabilidade e crescimento;
- SLA's - O provedor deve conseguir prometer um nível básico de serviço, respeitando as métricas oferecidas;
- Possui os serviços necessários para a nossa arquitetura;
- Custos - O provedor deve oferecer custos justos de seus serviços;
- Conhecimento técnico da equipe - A equipe de desenvolvimento deve possuir conhecimentos específicos técnicos para operar na plataforma escolhida.

Segundo dados de mercado pela Statista (2021), atualmente os três maiores provedores de nuvem no mundo são, em ordem de fatia de mercado, AWS (Amazon - 33%), Azure (Microsoft - 20%) e GCP (Google - 10%). Usaremos estes três provedores para fazer nossa análise, utilizando os quatro pontos descritos acima como balizas.

### 4.12.1 Saúde Financeira

Neste ponto os três provedores de nuvem estão bem colocados. Baseado em seus resultados recentes de faturamento. No último trimestre de 2021, a AWS teve um faturamento de 13.5 bilhões de dólares. No terceiro trimestre de 2021, Microsoft Azure teve um faturamento de 15.1 bilhões de dólares. No ano de 2021, GCP atingiu um faturamento total de 18.1 bilhões de dólares (HOLORI, 2021).

AWS lidera no mercado de computação em nuvem com uma grande margem, porém Azure e GCP estão cada vez conseguindo mais espaço. Todos os três provedores analisados estão crescendo muito acima de seus concorrentes diretos, totalizando mais de 60% do mercado global de provedores de nuvem.

### 4.12.2 SLA

Em termos de SLA, os três provedores de nuvem também são muito semelhantes em seus serviços ofertados. A AWS garante um tempo de disponibilidade contínua por

mês de 99.95%[4]. Azure garante 99.99%[5] de tempo de disponibilidade contínua em seus serviços. Ainda por cima disponibiliza créditos nos serviços que são dados aos clientes em casos de tempos de atividades menores que 99% e 95%. A plataforma GCP garante 99.95%[6] de disponibilidade de seus serviços. Caso esse tempo não seja atingido, existe uma política de 50% do valor pago no mês para o serviço em formato de créditos.

### 4.12.3 Custos

Em relação aos custos dos provedores de nuvem, devemos levar em consideração os serviços que serão necessários para que o microsserviço possa operar com sucesso. Sendo eles o cluster de Kubernetes gerenciados e o serviço de banco de dados SQL distribuído.

#### *4.12.3.1 Kubernetes*

De acordo com a análise feita pela consultoria Simform (2022), os custos relacionados ao gerenciamento de um cluster de Kubernetes são muito semelhantes entre os três provedores de nuvem analisados. Outra análise feita em 2020 mostra que a AKS (Azure Kubernetes Service) e a GKE (Google Kubernetes Engine) possuem quase o mesmo custo do cluster gerenciado de Kubernetes, sendo a AWS EKS (Elastic Kubernetes Service) 5% mais cara em relação aos dois (CHAPEL, 2020). Tendo em vista as análises acima, é possível concluir que os três provedores de nuvem possuem custos muito semelhantes de custos de seus clusters gerenciados de Kubernetes.

#### *4.12.3.2 Banco de dados*

A forma de precificar um serviço de banco de dados depende de diversos fatores. Sendo alguns deles: Quantidade total de dados persistida, a quantidade de dados processados por instância do banco por hora, a quantidade de dados em backup e por último a quantidade de banda de rede necessária. Para fins de simplificação, será considerado como base da análise apenas a quantidade total de dados persistida no banco de dados e espaço de backup. Será inferido um espaço de armazenamento de 32Gb e 5Gb para

[4]https://aws.amazon.com/pt/legal/service-level-agreements/
[5]https://azure.microsoft.com/en-us/support/legal/sla/
[6]https://cloud.google.com/terms/sla

backup.

Por conta da necessidade de tolerância a falhas e sharding do nosso banco, a análise será calculada utilizando os preços de persistência de dados em multi-região de cada um dos provedores de nuvem. Serão analisados os bancos de dados AWS RDS, Google Cloud Spanner e Azure SQL Database por serem as contra partes de cada provedor de nuvem de um banco de dados relacional com capacidades de processamento distribuído.

Baseado na calculadora da própria Azure[7], um banco de dados com as características acima teria um custo mensal de 4,012.23 dólares. Dadas as mesmas características, a calculadora de valores da GCP[8] estima um valor de 4,380.00 dólares por mês, um pouco mais acima do que a contra parte da Azure. Por fim, a calculadora de valores da AWS[9] estima um valor bruto de 5.120,76 dólares por mês, valor mais acima do que os bancos analisados anteriormente.

### 4.12.4 Discussão sobre provedores de nuvem

Ao analisar os pontos acima, é perceptível a grande semelhança dos serviços e características ofertados pelos três provedores de nuvem. Não existe nenhum ponto analisado que se destaque de algum dos provedores de nuvem, além de algumas diferenças na parte de custos.

O que mais pode pesar nessa decisão pode ser visto em situações de negociação de contratos e projetos. É comum haver parcerias de migração de ecossistemas inteiros de empresas para a infraestrutura dos provedores de nuvem. Empresas que possuem softwares on premise costumam adotar essa abordagem para migrar suas aplicações para a nuvem. Muitas vezes esses tipos de parcerias são o que definem qual provedor de nuvem será utilizado. Por conta do cliente da aplicação analisada já possuir um contrato de migração de outras aplicações com a Google, a escolha de permanecer no mesmo provedor de nuvem facilita a implementação da nossa arquitetura de microsserviços na infraestrutura de nuvem da GCP. Além do mais, a GCP oferece acesso gratuito durante 12 meses para alguns de seus serviços e 300 dólares em créditos para conhecer e avaliar a GCP por completo. Por este ser um trabalho de pesquisa acadêmica, que não contempla verba para financiar compra dos serviços dos provedores de nuvem, a possibilidade de utilizar

[7] https://azure.microsoft.com/en-us/pricing/calculator/?service=azure-sql-database
[8] https://cloud.google.com/products/calculator#id=29438cf6-37db-4725-8964-03386a0715a1
[9] https://calculator.aws/#/createCalculator/RDSMySQL

recursos pagos para testes é um grande diferencial.

# 5 DESIGN DA ARQUITETURA PROPOSTA E IMPLEMENTAÇÃO

Após a análise realizada sobre a situação atual do sistema e seus problemas, sobre decisões estratégicas tomadas pelo cliente e toda a fundamentação teórica a respeito de microsserviços, a arquitetura projetada para a aplicação possui grandes diferenças em tecnologias e abordagens em relação à arquitetura monolítica atual. Nas próximas subseções iremos discutir e analisar os detalhes da arquitetura proposta e também da implementação de parte da mesma para exemplificar e avaliar, em forma de uma prova de conceito, os designs propostos.

## 5.1 Design da arquitetura proposta

Será descrito nesta seção os detalhes e decisões de arquitetura para a nova arquitetura de microsserviços proposta, assim como quais ferramentas e aplicações serão utilizadas na arquitetura final.

### 5.1.1 Stack de tecnologia

Como visto na seção de linguagem, Java foi a stack de tecnologia escolhida para essa proposta de migração. Baseado na pesquisa de ecossistemas Java de 2021 encomendada pela Snyk (2021) em parceria da empresa Azul, onde 2000 desenvolvedores responderam diversas perguntas em relação à linguagem, *frameworks* e ferramentas, Spring Boot continua sendo o principal framework para desenvolvimento de microsserviços em Java. Dado os fatos acima, a implementação da proposta de arquitetura irá utilizar Spring Boot como framework de desenvolvimento web e microsserviços. A versão utilizada será a mais recente, Java 18.

### 5.1.2 Cluster de Kubernetes

Será desenhado um cluster de Kubernetes simples e sem muitos componentes extras. Por não haver a necessidade de service mesh, sua implementação acaba sendo menos complexa. Será criado um pod para o microsserviço, que ficará exposto ao mundo exterior para que seja consumido pelos serviços e aplicações através de um recurso de *Ingress*.

Mais detalhes sobre o *Ingress* serão descritos na seção abaixo. O motor que irá executar e gerenciar o cluster desenvolvido será o GKE, motor de kubernetes da GCP.

#### *5.1.2.1 Lista de permissões*

Não é recomendado que o serviço fique exposto a qualquer conexão de quaisquer aplicações se não for estritamente necessário. Com isso em mente, uma controladora *Ingress* que irá possuir um range de ips autorizados a fazer requisições diretamente. Apenas os serviços permitidos poderão fazer contato com o microsserviço desenvolvido. A controladora irá trabalhar como um verificador de permissão, onde irá permitir apenas o recebimento de requisições de aplicações confiáveis, trazendo mais segurança para a aplicação, pois diminui a possibilidade de agentes mal intencionados entrarem em contato com a aplicação.

#### *5.1.2.2 Balanceamento de carga*

Para o balanceamento de carga será utilizado o recurso padrão de serviços de *LoadBalancer*. Esse recurso é capaz de balancear e distribuir a carga dos *pods* criados de maneira automática.

#### *5.1.2.3 Escalonamento horizontal automático*

Para escalonamento horizontal, será utilizado o recurso de *Horizontal Pod Autoscaler*. O valor de quantidade de réplicas mínimas será de 1, o valor de quantidade de réplicas máximas será de 10 e a métrica analisada para disparar o escalonamento automático será de porcentagem de CPU utilizada acima de 50%.

### 5.1.3 Message Broker

Para serviço de mensageria foi escolhido o RabbitMQ. A escolha de usar o RabbitMQ como serviço de mensageria se baseia na questão de fácil implementação e integração com o ambiente de cluster do Kubernetes e com aplicações Spring Boot, por já ser uma. A autenticação para *publisher* e *consumer* são feitas através de segredos de usuário e senha.

### 5.1.4 Banco Dados

Baseado na escolha da GCP como provedora de nuvem e a necessidade de um banco distribuído que possua capacidade de *sharding*, o Google Cloud Spanner se torna a escolha trivial de sistema de banco de dados por possuir todas as características acima citadas. Além disso, o banco fornece um emulador que facilita o desenvolvimento e teste de aplicações localmente, pois evita a necessidade de fazer conexão com um banco real, evitando assim a geração de custos durante a fase de desenvolvimento. Ele terá configurações de multi-região e *sharding* dos dados baseado em um *UUIDs* gerados pelo próprio banco, evitando a criação de *hotspots* na base de dados distribuída (GOOGLE, 2022b).

### 5.1.5 Observabilidade

Splunk e New Relic possuem sistemas próprios de alertas baseado em métricas, queries e análise de dados customizados. Esses sistemas possuem integração com ferramentas de mensagens instantâneas, e-mails, ligações automáticas e diversas outras formas de contato. Nossa arquitetura irá usar esses alertas nativos das aplicações para identificar possíveis cenários de erros.

#### *5.1.5.1 Splunk*

O Splunk será responsável por possuir os alertas da nossa aplicação em relação a logs que indiquem algum tipo de falha. Eles serão configurados para serem disparados quando uma certa quantidade de erros forem coletados em uma certa quantidade de tempo. Este trabalho ficará restrito a apenas a logs de erros que indiquem falhas nas comunicações entre serviços e erros internos.

#### *5.1.5.2 New Relic*

Para o New Relic usaremos suas métricas default de uso de memória, processador e tempo de resposta de requisições para caso estes valores passem de limites pré estabelecidos como seguros na nossa aplicação. Por conta da natureza pessoal do trabalho, os alertas serão mandados para o e-mail pessoal do autor.

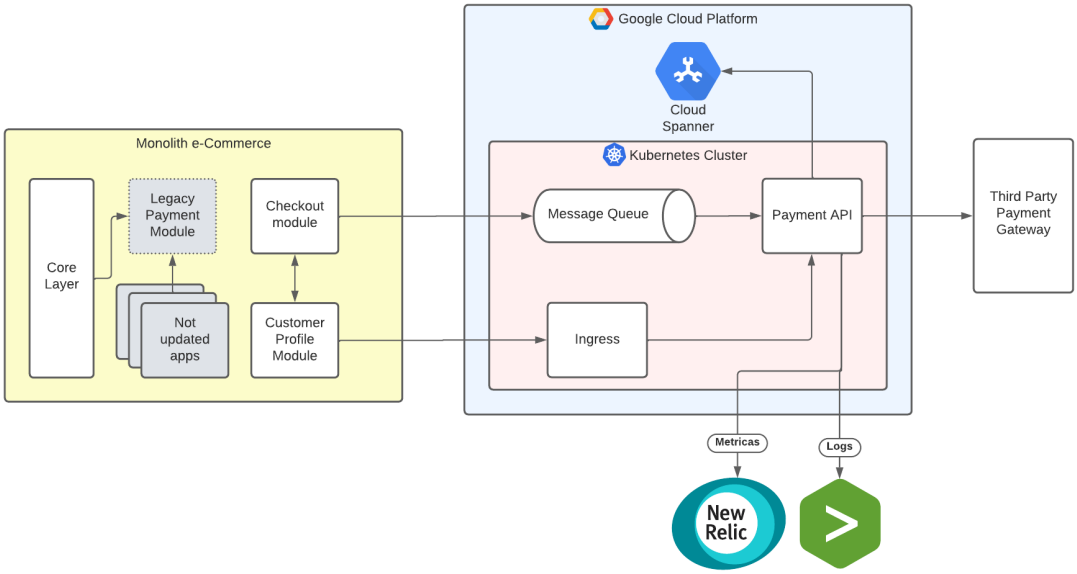
### 5.1.6 CI/CD

Apesar deste tópico não ter sido analisado no trabalho, por conta do uso de uma provedora de nuvem e um ambiente de cluster de Kubernetes, diversos serviços de CI/CD podem ser usados para integrar a base de código com o ambiente de deploy. É possível usar ferramentas como *Harness*, *CircleCI* ou até mesmo o *Cloud Building*, solução da Google para CI/CD nativo em nuvem.

### 5.1.7 Diagrama final

A imagem abaixo exemplifica a arquitetura final proposta por este trabalho. É importante lembrar que o *gateway* de pagamentos se mantém intacto, já que é uma aplicação *third-party* não gerenciada pela empresa que mantém o monólito.

Figura 5.1: Diagrama de componentes da arquitetura de microsserviços proposta




Fonte: Autor

## 5.2 Implementação da prova de conceito da arquitetura proposta

Como visto em seções anteriores, o tempo e custo de implementação de toda a arquitetura proposta não é compatível com o tempo e extensão do trabalho. Portanto, neste trabalho será implementado uma prova de conceito que irá conter as partes mais importan-

tes da arquitetura discutida e analisada a fim de poder validar os aspectos e características elencadas durante as análises da seção anterior.

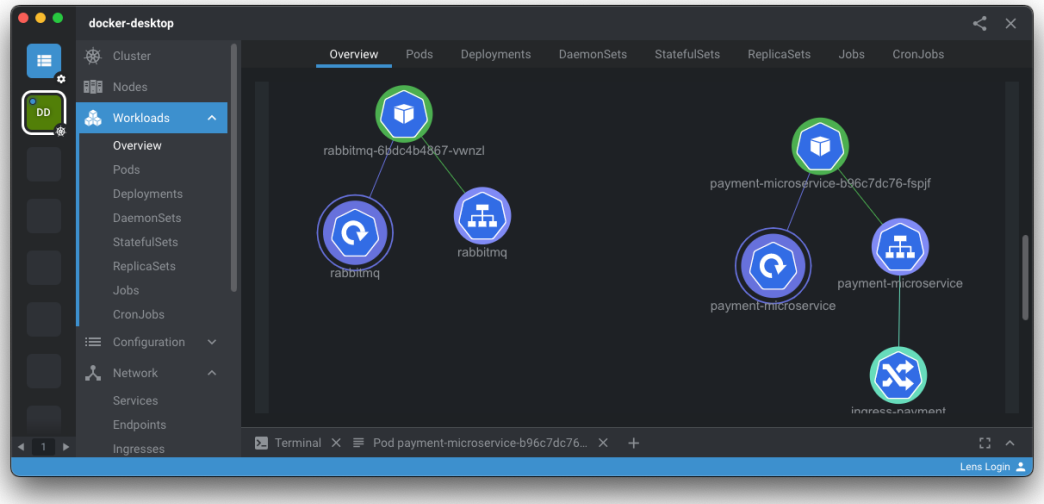
Foi implementado o cluster local de Kubernetes contendo o microsserviço, *message broker*, uma controladora *Ingress*, um controlador de escalonamento automático, uma integração com uma instância real do Google Cloud Spanner e a configuração do New Relic para gerenciamento de métricas da aplicação e do cluster. Além disso, foi modelado um código *stub*, para fins de exemplificar a migração da arquitetura, de uma parte do sistema legado que se comunica com o módulo migrado e a *API* de pagamento utilizada pela aplicação monolítica atual.

Por conta da natureza de implantação local dessa prova de conceito, as funcionalidades de *CI/CD* não foram implementadas. O Splunk será substituído pelo serviço Loggly, por conta de uma maior facilidade de configuração e tempo maior de gratuidade de uso do serviço.

### 5.2.1 Kubernetes

Por conta de restrições de tempo e problemas de provisionamento de serviços na própria GCP, o cluster não foi criado na infraestrutura da GCP. O cluster de Kubernetes foi criado localmente através do Kubernetes do *Docker for Desktop*. O cluster contém *pods* e serviços do RabbitMQ e do microsserviço de pagamentos, além da controladora *Ingress* do microsserviço.

Figura 5.2: Diagrama do cluster na ferramenta Lens




Fonte: Autor

Todos os componentes foram criados através de arquivos de manifestos de Kubernetes que estão versionados no GitLab[1].

### 5.2.2 microsserviços de pagamentos

O microsserviço de pagamentos foi desenvolvido em Spring Boot e implementa três tipos de casos de uso. Ele processa pagamentos, retorna todos os dados de pagamentos de um cliente específico e retorna informações de pagamento por id. O processamento de pagamentos é feito consumindo as mensagens publicadas pelo aplicativo legado e realizando chamadas REST para a API do Gateway de pagamentos desenvolvido para a prova de conceito. Durante o processamento e retorno dos dados de pagamento, o microsserviço se comunica com uma instância real do Google Cloud Spanner. Mais detalhes sobre o banco serão descritos na subseção de banco de dados desta seção. Mais detalhes de implementação do microsserviço podem ser vistos no repositório remoto do GitLab[2].

### 5.2.3 Aplicação monolítica legado

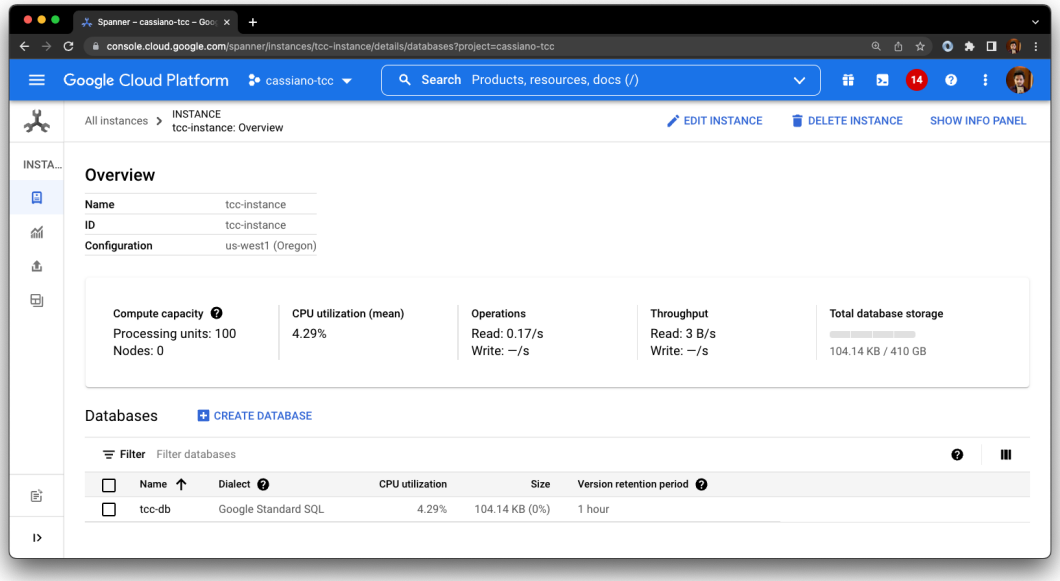
Por conta de questões legais e de segurança, não é possível mostrar código da aplicação original, portanto impossibilitando a modificação do código para exibição neste trabalho. Por conta disso, foi criada uma aplicação monolítica legado e que implementa alguns casos de uso presentes na aplicação original. A aplicação possui exemplos de códigos legados e também uma refatoração para publicar mensagens *message broker* RabbitMQ para serem consumidas pelo microsserviço de pagamento. A implementação da aplicação pode ser vista no Gitlab[3].

### 5.2.4 Banco de dados

O banco de dados utilizado é uma instância real do Google Cloud Spanner. Por motivos de custos, a instância não possui configuração de multi regiões, apenas uma instância na região *us-west1(Oregon)*, região geográfica do data-center que provisiona

[1] https://gitlab.com/cassianojaegertcc/kubernetes-manifests
[2] https://gitlab.com/cassianojaegertcc/payment-microservice
[3] https://gitlab.com/cassianojaegertcc/mock-legacy-app

Figura 5.3: Configuração do banco de dados Google Cloud Spanner




Fonte: Autor

o banco de dados. A imagem 5.3 exibe algumas informações referentes a configuração da instância e do banco de dados.

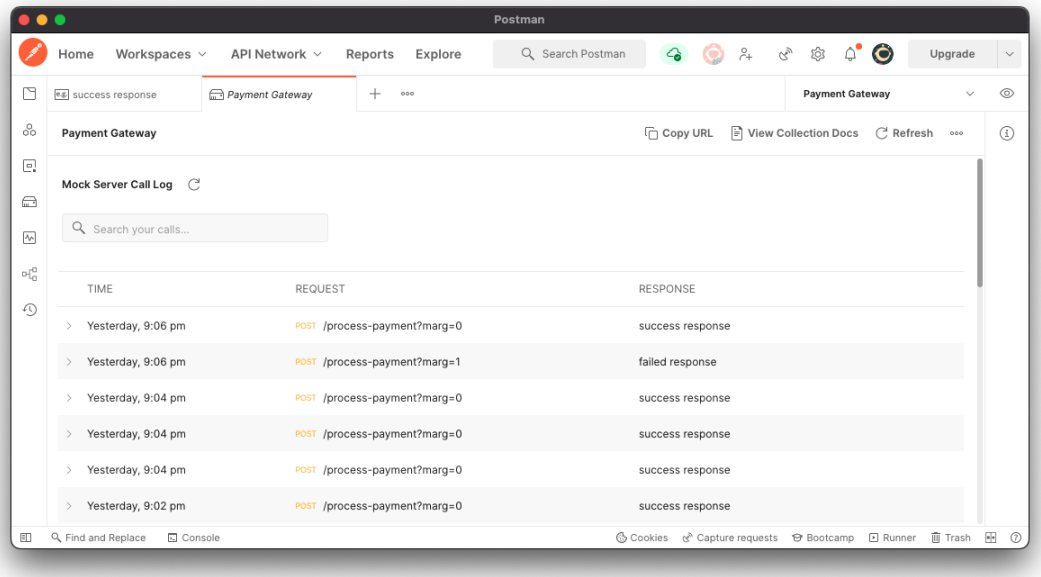
Sua configuração é a mais econômica possível, possuindo apenas 100 unidades de processamento com apenas uma região. Por conta disso a sua performance é um pouco prejudicada, porém essa configuração foi feita apenas para a prova de conceito. Aplicações produtivas irão possuir configuração multi região e maior quantidade de unidades de processamento. A configuração está disponível no console[4] da Google Cloud Platform.

### 5.2.5 Gateway de pagamento

Assim como a aplicação monolítica real, o *gateway* de pagamento utilizado também possui restrições de uso. Portanto foi criado um *mock server*, servidor falso que recebe requisições e devolve respostas pré formatadas, para representar o *gateway*. Esse servidor foi escrito utilizando o Postman, aplicação multiplataforma de construção e uso de *API's*. Seu funcionamento consiste em retornar respostas com valores aleatórios e pré-formatados para as requisições de pagamento enviadas pelo microsserviço de pagamentos. A imagem 5.4 mostra a configuração do *mock server*.

[4]https://console.cloud.google.com/spanner/instances/tcc-instance/details/databases?project=cassiano-tcc

Figura 5.4: Interface do Postman contendo a configuração do *mock server*




Fonte: Autor

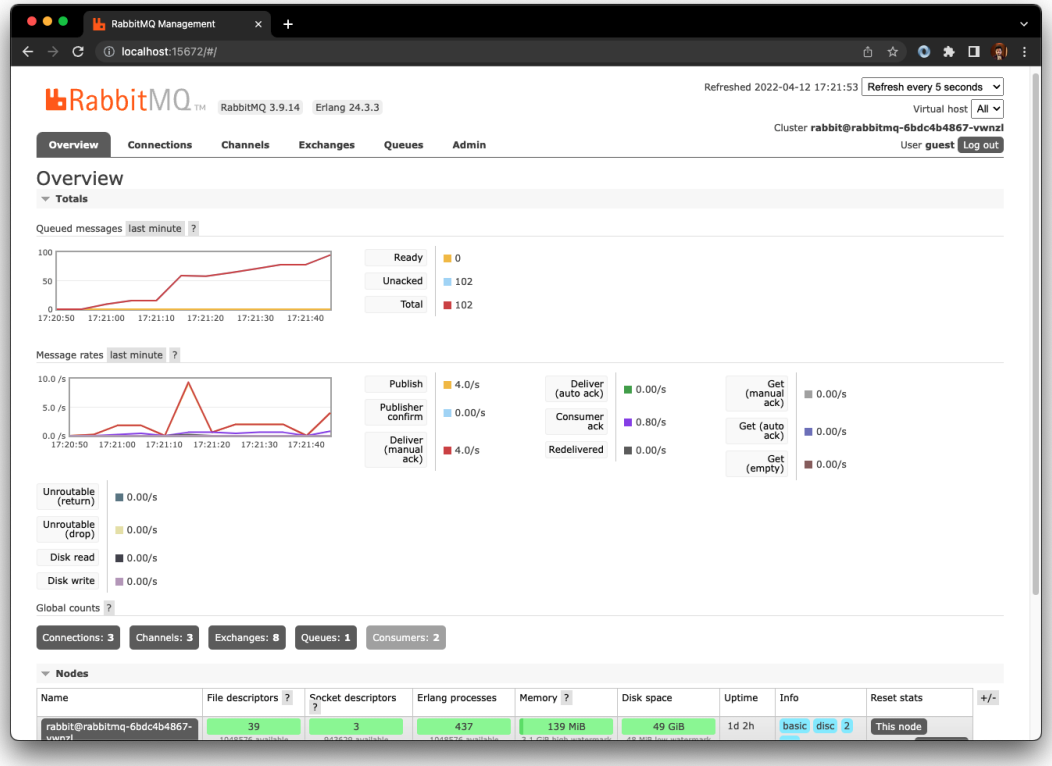
### 5.2.6 RabbitMQ

O *message broker* não possui nenhum tipo de alteração ou modificações de funcionamento. Ele foi adicionado ao cluster através do manifesto de *deployment* do Kubernetes, utilizando uma imagem padrão disponível nos registros de imagens do *Docker*. Uma *exchange* e uma fila foram criadas e as mensagens são publicadas e consumidas através delas. As duas aplicações se autenticam com o serviço através de credenciais de usuário e senha. A imagem 5.5 exibe o console de administrador da ferramenta, mostrando algumas configurações básicas.

## 5.3 Discussão

Nesta seção serão discutidos alguns pontos importantes que demonstram as diferenças e características da antiga arquitetura em relação a arquitetura proposta por este trabalho.

Figura 5.5: Console de administrador do RabbitMQ implantado no cluster




Fonte: Autor

### 5.3.1 Configuração do banco de dados e observabilidade

A configuração de um novo banco de dados para o microsserviço abriu oportunidades para melhorar a modelagem dos dados e aumentar o ganho de desempenho das operações no banco de dados. Por conta de ser um banco multi região com *sharding* de dados, novos desafios são encontrados ao modelar os dados e configurar o banco, porém por conta do uso de novas tecnologias em um ambiente preparado para escalonamento, isso se torna muito mais fácil.

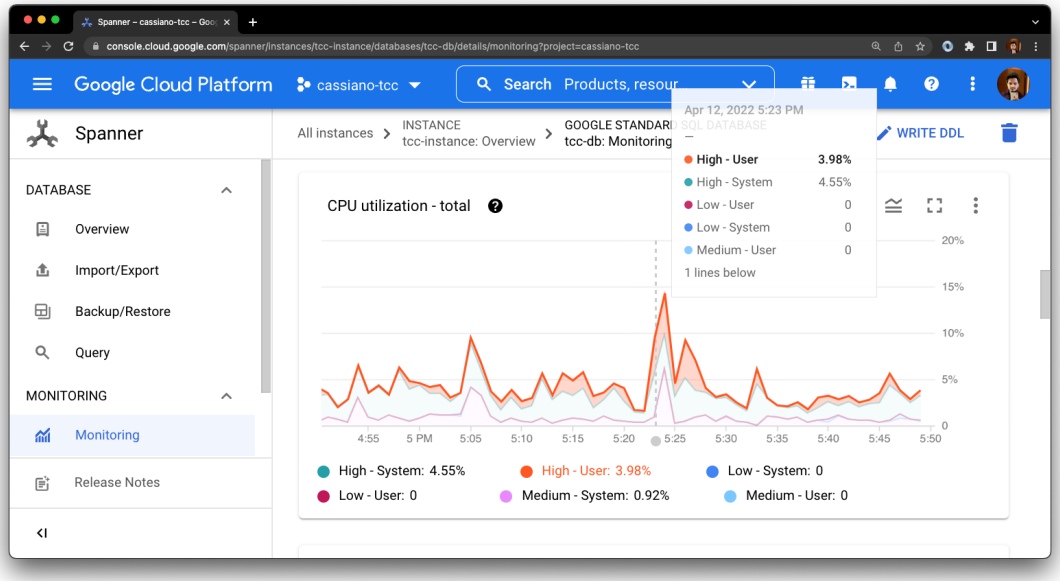
Fowler (2012a) apresenta a técnica de escalonamento horizontal de banco de dados, chamada de *sharding*, onde cada *sharding* é uma instância do banco de dados usado para dividir a carga. Ela "roteia"os dados da tabela baseada nos valores de identificação primário. O Google Cloud Spanner possui uma modelagem chamada de *Interleaved Tables* para melhorar a performance do banco de dados distribuído. Essa modelagem, usada principalmente em relação de 1:N, faz com que dados relacionados sejam persistidos no mesmo *shard*, evitando assim que buscas a dados relacionados precisem acessar outros *shards* em busca de dados.

O banco de dados do aplicativo legado é um super-banco de mais de 15 anos. A grande quantidade de dados no mesmo local e a complexidade de uma modelagem que foi evoluindo ao longo desses anos prejudicou muito a performance de buscas e persistência de dados no mesmo.

Ao utilizar um banco de dados específico para o microsserviço e ainda utilizar técnicas de aumento de performance e resiliência como *sharding* e *Interleaved Tables*, auxilia muito na não degradação do banco de dados ao longo do tempo, aumentando a vida útil do banco e garantindo uma performance satisfatória durante sua evolução ao longo dos anos.

Por ser um banco de dados muito antigo, o banco legado possui um gerenciamento e observabilidade muito reduzidos. Com o uso de um serviço de banco de dados em nuvem, como no caso do Google Cloud Spanner, a observabilidade e manutenção do schema são facilitados. O banco possui uma interface inteira de monitoramento e métricas em tempo real.

Figura 5.6: Console de métricas do Google Cloud Spanner




Fonte: Autor

É possível ver na imagem acima apenas uma das métricas observadas em tempo real. Existem diversas outras como *throughput*, latência de conexões, operações por segundo e etc. Essas métricas não eram facilmente obtidas com a ferramenta de gerenciamento de banco de dados utilizada no banco legado.

### 5.3.2 Assincronicidade e performance

Durante o fluxo de processamento de pagamento do sistema legado, diversas operações de banco de dados e *API's* eram feitas. Esses fluxos síncronos pesam muito na aplicação e degradam sua performance consideravelmente com o aumento de requisições feitas por segundo.

```
public void processPaymentRequest(PaymentDTO paymentRequest) {
  /**
   * Legacy Code developed to process payment request
   * Code masked and hidden for security purposes
   */
  /**
   * Synchronous operations
   * - API calls to payment gateway
   * - Database operations
   * A regular payment request would take 5s-10s to finish
    */
  /**
   * More 250+ lines of code to perform payment request
   */
 }
}
```

O trecho acima é uma representação simplificada e anonimizada do processo de requisição de pagamento na base de código legada. É possível identificar quebras de características de código limpo, como por exemplo o princípio de classes e métodos pequenos e o princípio de façam apenas uma coisa (MARTIN, 2008).

A migração de toda essa lógica para um microsserviço, utilizando um *message broker* para trocas seguras de comunicação assíncrona, garante mais segurança e clareza de código para ambas aplicações.

```
  public void sendPaymentRequest(PaymentDTO paymentDTO) throws
     JsonProcessingException {
  String paymentDtoJson =
     objectMapper.writeValueAsString(paymentDTO);
```

```
    Message message = MessageBuilder
        .withBody(paymentDtoJson.getBytes())
        .setContentType(MessageProperties.CONTENT_TYPE_JSON)
        .build();

    rabbitTemplate.convertAndSend(MockLegacyAppApplication.topicExchangeName,
        "payment.processing.test", message);
}
```

O uso do *message broker* facilita o desacoplamento da arquitetura, visto que o sistema legado não precisa saber quem irá processar a requisição e muito menos se importar se a mensagem chegou. É visível também como ficou facilitada a testagem do código, por conta da migração da lógica para outro sistema.

### 5.3.3 Escalonamento horizontal automático e balanceamento de carga

Na aplicação legado, é necessário fazer o provisionamento de máquinas físicas, caso não exista nenhuma disponível, e gerenciar manualmente a implantação de uma nova instância da aplicação para que se possa escalar horizontalmente. Além disso, ainda é necessário fazer a configuração do balanceamento de carga para essa nova instância de aplicação. Todo esse processo é extremamente custoso e propenso a erros.

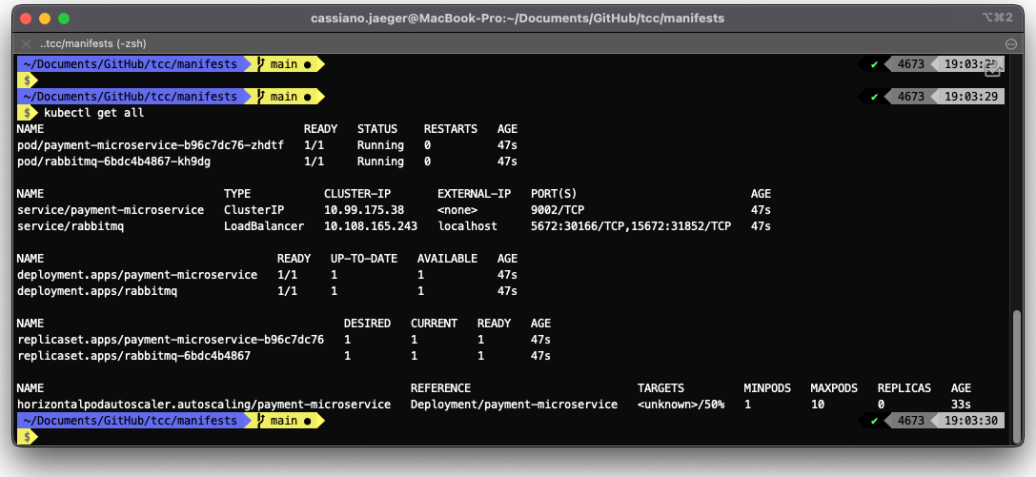
A nova arquitetura torna esse processo mais fácil e rápido. O provisionamento de máquinas e implantação de novas instâncias é feito de maneira automática pelo recurso de *Horizontal Pod Autoscaler* do Kubernetes.

É possível visualizar o recurso de escalonamento automático funcionando antes (5.7), durante (5.8) e depois (5.9) de um pico de processamento no cluster. Não é necessário nenhum tipo de intervenção manual. O servidor de métricas do Kubernetes monitora os recursos de forma automática, enquanto o recurso de *Horizontal Pod Autoscaler* gerencia a adição ou remoção de novos *pods*.

### 5.3.4 Métricas

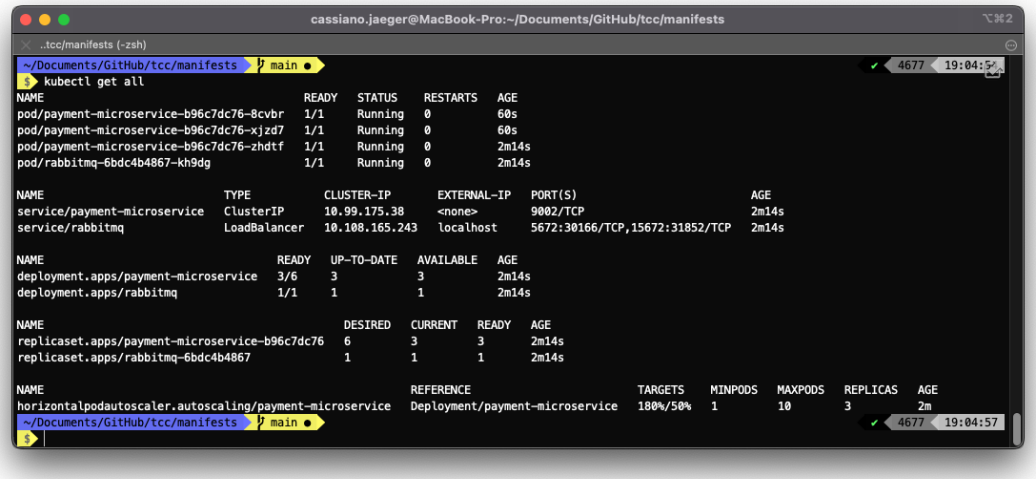
A aplicação legado não possui métricas de performance, uso e outras informações relevantes para a manutenção da operação do sistema. O New Relic foi configurado na

Figura 5.7: Imagem da CLI *kubectl* descrevendo as informações do cluster no início de sua operação



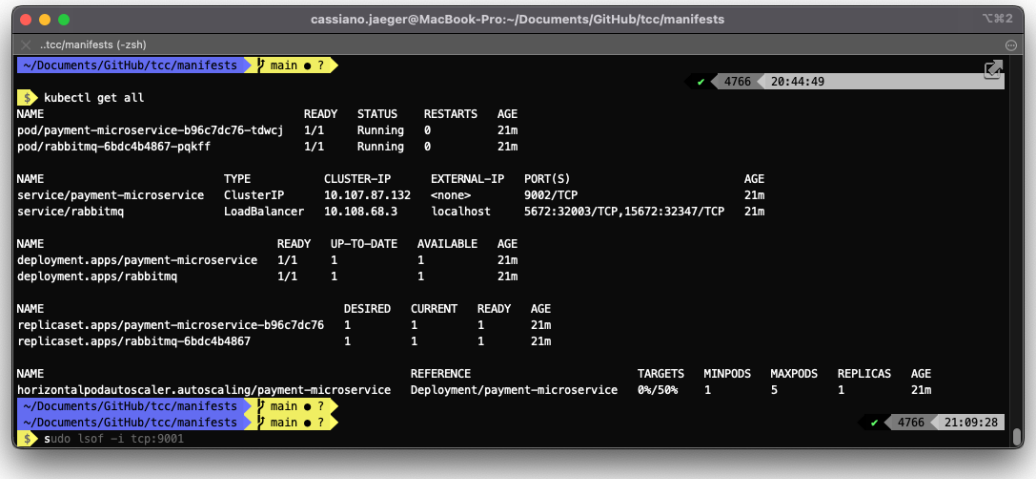

Fonte: Autor

Figura 5.8: Imagem da CLI *kubectl* durante o processo de *out-scaling* por conta de um aumento na carga recebida nos microsserviços




Fonte: Autor

Figura 5.9: Imagem da CLI *kubectl* após o processo de *out-scaling*, onde as instâncias são reduzidas por conta da diminuição da carga no microsserviço




Fonte: Autor

arquitetura proposta de microsserviços tanto para coleta de métricas do cluster de Kubernetes como para as métricas específicas da aplicação.

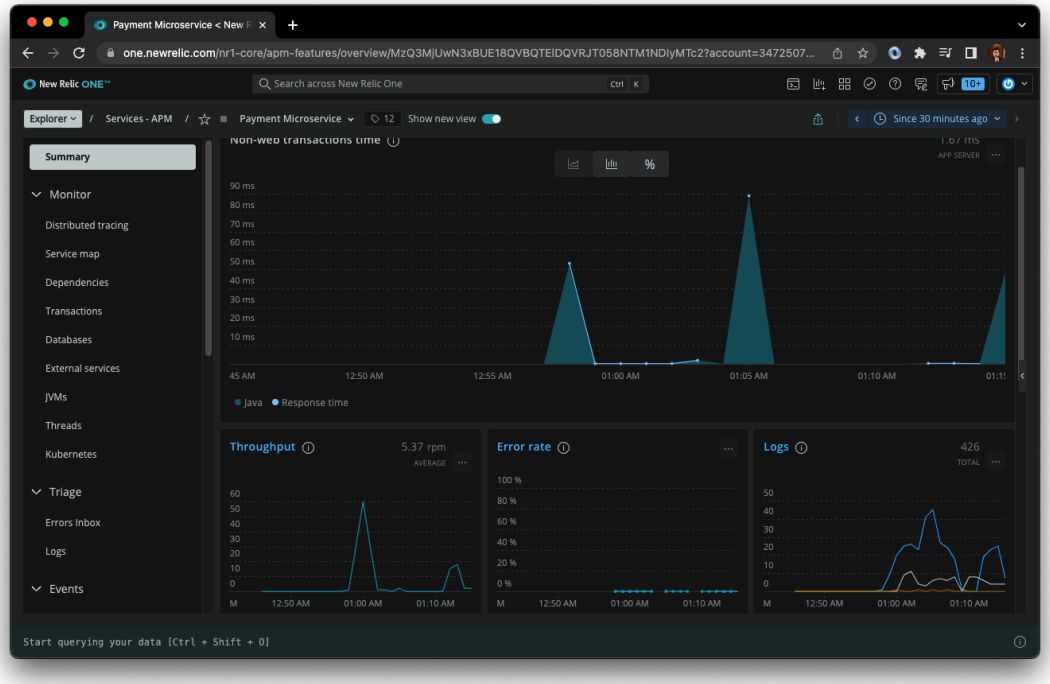
As métricas da aplicação são inúmeras e vão desde *throughput* de dados, tempo de resposta, quantidade de erros até informações sobre *threads* correntes, conexão com banco de dados e etc. Na imagem 5.10 é possível ter uma ideia de todas as métricas disponíveis no menu esquerdo.

Além de métricas detalhadas da aplicação, também é coletado informações do próprio cluster de Kubernetes. Por serem métricas muito focadas em infraestrutura, recursos e configurações de rede, detalhes e descrições serão omitidas. Essas métricas são de grande valia para entender e monitorar o estado de saúde das aplicações e do próprio cluster de Kubernetes, pois o nível de detalhamento é muito expressivo. Na imagem 5.11 é possível ver algumas dessas métricas coletadas da aplicação.

### 5.3.5 Logs da aplicação

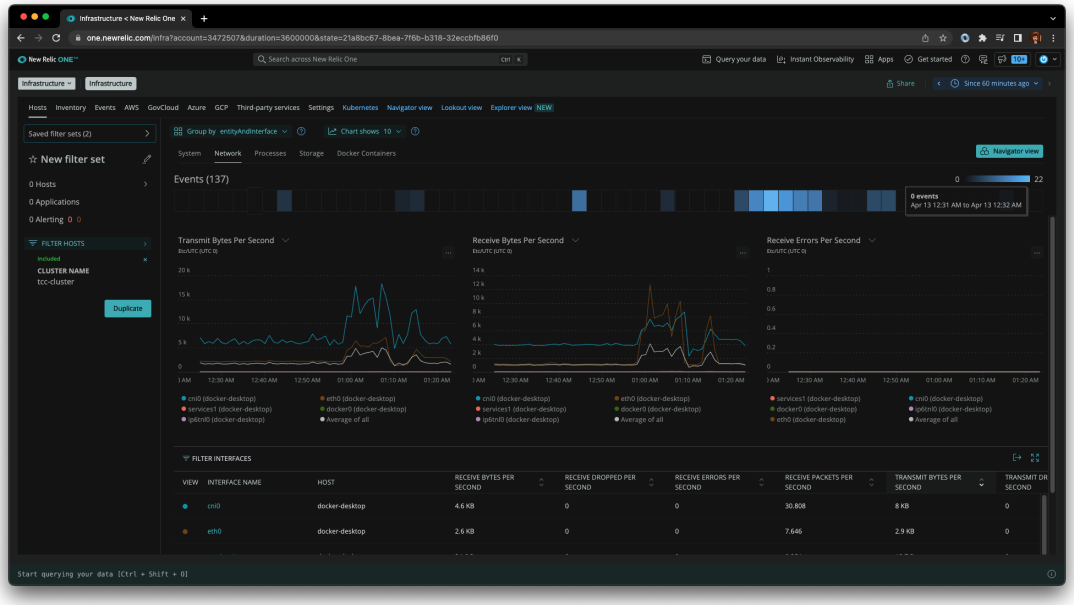
A aplicação legado não possui um agregador de logs. Para se ter acesso aos logs de da aplicação é necessário acessar uma aplicação interna da empresa. A ferramenta é bastante limitada e não fornece alertas ou maneiras de facilitar buscas e *debugging*. Por conta de custos e tempo de gratuidade no agregador de logs Splunk, foi optado por utilizar uma alternativa que possui mais tempo de gratuidade no serviço, auxiliando assim

Figura 5.10: Imagem da página inicial do New Relic



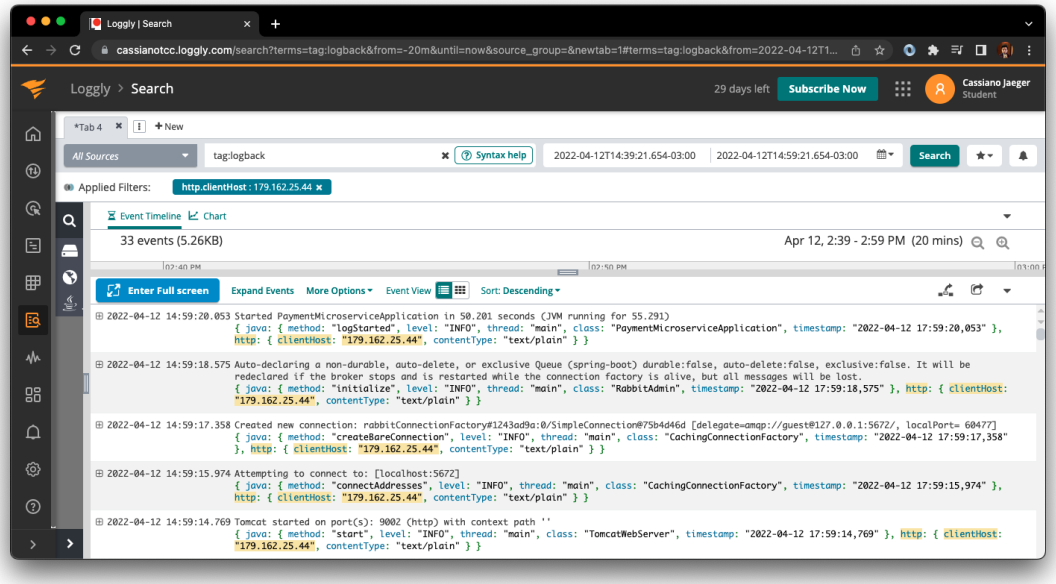

Fonte: Autor

Figura 5.11: Imagem da página de métricas do New Relic




Fonte: Autor

Figura 5.12: Imagem da página de logs da aplicação no Loggly




Fonte: Autor

no desenvolvimento deste trabalho. O microsserviço desenvolvido possui integração com a ferramenta de logging chamada Loggly. A ferramenta agrega os logs de todos os pods da aplicação e exibe de forma simplificada. Foi desenvolvido uma integração simples com a *ferramenta*, optando apenas pela coleta e exibição dos logs. A imagem 5.12 mostra como é a página inicial do agregador de logs utilizado na arquitetura.

# 6 DISCUSSÃO SOBRE A AVALIAÇÃO DOS ESPECIALISTAS

Grande parte do processo de desenvolvimento e arquitetura de software se baseia na cooperação e troca de feedbacks entre desenvolvedores e arquitetos. Como forma de confirmar as análises contidas neste trabalho, esta seção será responsável por definir critérios de avaliação da arquitetura proposta e expor as opiniões e sugestões de especialistas da área de tecnologia.

Os especialistas irão responder a um questionário contendo perguntas relacionadas a características e problemas do sistema monolítico e de como a arquitetura proposta neste trabalho resolve tais problemas. O questionário irá conter o desenho da arquitetura atual do sistema monolítico e informações relevantes sobre alguns critérios de avaliação que serão analisados. Os mesmos critérios serão analisados na arquitetura proposta. Dadas as respostas dos especialistas, será feita uma análise comparativa das mesmas e ponderações em relação aos resultados obtidos.

## 6.1 Critérios de avaliação

Os critérios de avaliação que serão utilizados para a arquitetura proposta são os mesmos que foram descritos na seção 3.2, sendo eles: Coesão e acoplamento, manutenção, performance e escalabilidade, disponibilidade e implantação. Primeiramente os especialistas irão responder algumas perguntas sobre seu perfil e experiência profissional. Após isso serão respondidas de duas a três perguntas discursivas que abordam cada um destes 5 pontos mencionados. Para mais detalhes sobre as perguntas, ver apêndice A.

## 6.2 Discussão da análise dos especialistas

Os perfis profissionais dos especialistas conseguiram trazer graus de diversidade nas respostas. Foram 4 pessoas, entre 26 e 31 anos, onde seus graus de escolaridade variam entre graduação incompleta até pós graduação completa. Seu tempo de experiência profissional varia entre médio e alto, onde os profissionais possuem entre 4 a 8 anos de experiência com desenvolvimento de software. A maioria está em uma posição júnior/-pleno, porém um dos especialistas é um desenvolvedor sênior em sua empresa. Todos já trabalharam com aplicações monolíticas e apenas um não trabalhou com microsserviços.

Os 4 especialistas concordaram entre si sobre problemas de coesão e acoplamento do sistema monolítico. O grande tamanho das classes e da base de dados, aliado a um grande número de dependências é a explicação utilizada por todos. O especialista mais sênior destacou que o maior problema é o acoplamento, visto que a coesão não parece ser necessariamente um problema baseado nas informações da pesquisa. Os 4 especialistas também concordam que a coesão e o acoplamento são melhor desenvolvidos na arquitetura de microsserviços propostas, já que com sistemas mais modulares existe maior facilidade de controlar dependências e coesão dentro do código. Mais uma vez o especialista mais sênior pontua que a coesão pode continuar a mesma, porém o acoplamento é diminuído por conta da divisão das responsabilidades das aplicações. Todavia existe uma desvantagem que é controlar e gerenciar a comunicação do sistema em relação ao antigo.

Novamente os especialistas concordam em relação aos problemas de manutenção que a arquitetura monolítica possui. É pontuado mais de uma vez sobre os problemas de isolar mudanças no código para que não existam efeitos colaterais indesejados, já que a cobertura de testes na aplicação é bem reduzida. Nenhum deles conseguiu abordar alguma vantagem de manutenção nesta aplicação. Os especialistas também foram unânimes na melhora da manutenção da aplicação, por conta do uso de novas tecnologias e processos de qualidade de software, bem como a legibilidade e reuso do código. Contudo, o especialista mais sênior levantou o ponto que a complexidade do sistema de pagamentos aumentou, visto que foi adicionado à arquitetura o *message broker* RabbitMQ. Para ele, a manutenção, em um primeiro momento, se tornou mais complexa. Porém o mesmo acredita que essas novas estruturas irão se pagar com o tempo.

Mais uma vez os especialistas concordam que a escalabilidade e performance do sistema monolítico apresentado é deficitária. Todos mencionam o gasto de recursos necessário para escalar horizontalmente a aplicação, visto que caso apenas um módulo esteja com pico de tráfego, uma aplicação inteira precisa ser instanciada, gerando gasto de recursos de hardware por módulos que não precisam ser escalonados. O especialista sênior ainda pontua que o problema da escalabilidade horizontal advém muito por uma questão monetária do que técnica, visto a grande alocação de recursos que muitas vezes não são usados de maneira eficiente. Os especialistas também pontuam o tempo gasto com provisionamento de hardware como algo a ser levado em consideração nestes ambientes. A escalabilidade vertical em sistemas monolíticos não foi um problema levantado, inclusive é dito por um dos especialistas que é a maneira comumente adotada nesse tipo de arquitetura. Sobre a escalabilidade na arquitetura proposta, os especialistas comentam sobre a

maior facilidade de escalar horizontal e verticalmente a aplicação, visto o uso de um cluster de Kubernetes que faz isso automaticamente utilizando a estrutura de *Auto Scaler*. A performance também é melhorada por conta de uma maior eficiência do uso dos recursos de hardware e software, já que é possível escalar separadamente o módulo de pagamento ou o *message broker*. Por fim, o módulo de pagamentos está isolado do sistema legado, evitando problemas de lentidão causados pelo sistema maior.

Em relação à disponibilidade, os especialistas levantam alguns pontos críticos da arquitetura monolítica analisada. Por se tratar de uma infraestrutura *on premise*, qualquer problema na infraestrutura pode afetar toda a operação do e-Commerce, e não apenas uma parte. Isso também é verdade para qualquer falha que ocorra dentro da aplicação, visto que ela pode interferir no funcionamento de toda a aplicação. Eles também pontuam o tempo de espera para a volta do serviço, devido ao grande tamanho e quantidade de dependências do sistema legado. Os especialistas comentam que a nova plataforma de execução auxilia muito na disponibilidade da aplicação, visto que o Kubernetes é capaz de reiniciar e rotear, para outras instâncias, as requisições de *pods* que não estejam funcionando corretamente e que possam comprometer o funcionamento da aplicação como um todo. Por ser um sistema em nuvem, erros de infraestrutura interna dificilmente acontecem, porém não são impossíveis. O desenvolvedor mais sênior salienta que esse tipo de melhora não advém da arquitetura de microsserviços por si só, mas sim do conjunto da arquitetura e da plataforma de execução em nuvem utilizando Kubernetes.

Por fim, o último tópico da análise era sobre a implantação da aplicação. Um dos especialistas analisa o fato de termos muitas etapas manuais e como isso é propenso a erros. Outros falaram que a falta de detalhes comprometeria seu julgamento e portanto se reservaram a não darem suas opiniões. Em relação a nova arquitetura de microsserviços, os especialistas comentam a maior facilidade e segurança da implantação automatizada através de pipelines como *CircleCI* e *Harness*. Gerando menos custos e mais controle das tarefas performadas. Entretanto, o desenvolvedor sênior levantou o ponto sobre a ausência de sistemas de homologação na descrição das arquiteturas e como isso pode ser prejudicial. Isso se deve ao fato de que a descrição do processo de implantação não estava completa, gerando dúvidas na avaliação do especialista.

Metade dos especialistas não adicionou nada nas seções de comentários extras, porém dois, mais seniores, deixaram comentários sobre a arquitetura monolítica e sobre a arquitetura de microsserviços. Um deles mencionou a importância de projetos de migração para microsserviços onde os pontos abordados por este trabalho sejam levados em

consideração, visto que, para ele, são pontos extremamente relevantes para todas as partes interessadas. Outro comentou que parte dos problemas que ele avaliou na arquitetura monolítica não advém apenas do seu tipo de arquitetura, e sim também de outros fatores que transformam qualquer arquitetura, inclusive microsserviços, em bombas relógios.

Por fim, o especialista mais sênior faz uma grande análise sobre a nova arquitetura proposta como um todo. Ele confirma diversas melhorias apontadas no trabalho, porém ressalta que outras não originam apenas da arquitetura de software proposta, mas sim de todo o contexto em volta, como a provedora de nuvem escolhida, plataforma de Kubernetes e etc. Ele finaliza reforçando seu ponto anterior de que parte dos problemas da arquitetura legada analisada não foram gerados pela arquitetura em si, mas sim por diversos outros fatores. As respostas de todos os quatro especialistas podem ser vistas na íntegra no apêndice B.

# 7 CONCLUSÃO

Este capítulo descreve a conclusão deste trabalho em 3 seções: resultados e contribuições, limitações e trabalhos futuros.

## 7.1 Resultados e contribuições

As discussões investigadas durante o trabalho demonstram a quantidade de informação, conhecimento e esforço necessário para que um processo de migração de uma arquitetura monolítica para uma arquitetura de microsserviços ocorra de maneira eficiente e organizada. Para isso foi feita uma investigação e comparação de diversas tecnologias, abordagens e técnicas para identificar o passo-a-passo necessário para este tipo migração, assim como as melhores escolhas feitas baseadas no contexto do sistema a ser migrado. Foram avaliados tipos e tecnologias de banco de dados, tecnologias de gerenciamento de contêineres, plataformas de observabilidade e assuntos mais abrangentes como, por exemplo, qual linguagem de programação ou provedor de nuvem trabalhar.

Foi construída uma abordagem de migração baseada nas discussões vistas no capítulo 4 e a mesma foi implementada com sucesso, visto no capítulo 5. Por fim, especialistas de tecnologia avaliaram a arquitetura proposta e, em termos gerais, afirmaram que ela resolve os problemas de acoplamento, escalabilidade e disponibilidade vistos na seção 6 e abre espaço para mais melhorias. A proposta de migração também adiciona resiliência ao se utilizar um *message broker* para lidar com as requisições de pagamento. Por fim, o trabalho atinge seu objetivo de ser um guia prático e completo sobre o processo de migração de arquiteturas monolíticas para arquiteturas de microsserviços em ambientes de nuvem, além de propor discussões interessantes que podem ser aprofundadas em futuros trabalhos.

## 7.2 Limitações

A decomposição da arquitetura monólito em uma arquitetura de microsserviços se baseia fortemente na decomposição de seus módulos através de técnicas e conceitos do DDD. Por conta da limitação de tempo e escopo deste trabalho, esse processo foi limitado em apenas definir as funcionalidades de pagamento que seriam migradas para

microsserviços.

Por conta de limitação de escopo e tempo, houveram pontos que não foram possíveis de serem implementados ou que não alcançaram um nível maior de qualidade, sendo alguns deles:

- A implementação da arquitetura de microsserviços na infraestrutura gerenciada de Kubernetes da GCP. Esse processo foi iniciado durante o trabalho, mas ao longo do desenvolvimento notou-se que era preciso muito mais estudo e conhecimentos de gerenciamento de infraestrutura que não estão cobertos no escopo deste trabalho;
- A migração dos dados do sistema legado para o novo banco de dados da arquitetura de microsserviços proposta. Este tópico é complexo e interessante o suficiente para ser um trabalho de pesquisa por si só;
- O design e planejamento da migração dos módulos e sistemas que se comunicam com o módulo de pagamento legado para se comunicarem com o novo módulo de pagamentos desenvolvido em microsserviços.

Por conta de questões de *compliance* e segurança, não foi possível implementar a nova arquitetura na infraestrutura do cliente, se limitando a desenvolver a arquitetura proposta e integrá-la com uma aplicação *stub* que simula o comportamento do sistema real.

## 7.3 Trabalhos futuros

Ainda que algumas limitações foram enfrentadas, foi possível pensar em possibilidades de melhoria deste trabalho, sendo elas:

- Por conta do trabalho ser focado em apenas um módulo do monólito, é possível abordar a migração dos outros módulos da aplicação monolítica. Seria possível abordar outros pontos mais profundamente, produzindo um trabalho mais denso e completo;
- Identificar e trabalhos nos desafios de autenticação dos clientes que se comunicam com o cluster através de um sistema OAuth2;
- Fazer uso de uma *service mesh* para orquestrar o controle de tráfego, segurança de comunicação e observabilidade do cluster;
- Implementar a arquitetura proposta em uma provedora de nuvem, de preferência a

Google Cloud Platform, pois foi a melhor proposta para a arquitetura analisada;

Outro ponto importante que pode ser analisado no futuro é o trabalho de migração dos dados do banco do sistema legado para o banco na provedora de nuvem e de integração com os clientes que usam os módulos antigos. Este ponto é passível de uma análise muito profunda e cheia de *trade-offs* de performance e consistência no design, porém é limitado por questões de *compliance* do cliente.

# REFERÊNCIAS

BREWER, E. A. Towards robust distributed systems. In: **Symposium on Principles of Distributed Computing (PODC)**. [s.n.], 2000. Available from Internet: <http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf>.

CHAPEL, J. Managed kubernetes pricing comparison: Eks vs. aks vs. gke. 2020. [Online; accessed 13-Apr-2022]. Available from Internet: <https://jaychapel.medium.com/managed-kubernetes-pricing-comparison-eks-vs-aks-vs-gke-dbf3f2c6290c>.

CHAUDHRY, N.; YOUSAF, M. M. Architectural assessment of nosql and newsql systems. **Distrib. Parallel Databases**, Kluwer Academic Publishers, USA, v. 38, n. 4, p. 881–926, dec 2020. ISSN 0926-8782. Available from Internet: <https://doi.org/10.1007/s10619-020-07310-1>.

COLLIONI, A. de F. Estratégia de migração de sistemas ditos monolíticos para arquitetura de microsserviços. UFRGS, 2018.

COMELLA-DORDA, S. et al. A survey of legacy system modernization approaches. p. 30, 04 2000.

CORBETT, J. C. et al. Spanner: Google's globally distributed database. **ACM Trans. Comput. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 31, n. 3, aug 2013. ISSN 0734-2071. Available from Internet: <https://doi.org/10.1145/2491245>.

DOOLEY, K. **Designing Large Scale Lans**. [S.l.]: O'Reilly Media, 2001. ISBN 978-0596001506.

EL-REWINI, H.; ABD-EL-BARR, M. **Advanced Computer Architecture and Parallel Processing (Wiley Series on Parallel and Distributed Computing)**. USA: Wiley-Interscience, 2005. ISBN 0471467405.

EVANS, E. **Domain-Driven Design: Tackling Complexity in the Heart of Software**. [S.l.]: Addison-Wesley Professional, 2003. ISBN 978-0321125217.

FEATHERS, M. **Working Effectively with Legacy Code**. USA: Prentice Hall PTR, 2004. ISBN 0131177052.

FOWLER, M. Softwarecomponent. 2015. [Online; accessed 23-Fev-2022]. Available from Internet: <https://martinfowler.com/bliki/SoftwareComponent.html>.

FOWLER, M. J. Stranglerfigapplication. 2004. [Online; accessed 20-Mar-2022]. Available from Internet: <https://martinfowler.com/bliki/StranglerFigApplication.html>.

FOWLER, M. J. **Nosql Distilled: A Brief Guide to the Emerging World of Polyglot Persistence**. [S.l.]: Addison-Wesley Professional, 2012. ISBN 978-0321826626.

FOWLER, M. J. Testcoverage. 2012. [Online; accessed 23-Mar-2022]. Available from Internet: <https://martinfowler.com/bliki/TestCoverage.html>.

FOWLER, M. J.; LEWIS, J. Microservices. 2014. [Online; accessed 01-Mar-2022]. Available from Internet: <https://martinfowler.com/articles/microservices.html>.

GARAGE, R. Moving to microservices: Top 5 languages to choose from. 2019. [Online; accessed 01-Apr-2022]. Available from Internet: <https://rubygarage.org/blog/top-languages-for-microservices>.

GARTNER. Ib (integration broker). 2022. [Online; accessed 10-Mar-2022]. Available from Internet: <https://www.gartner.com/en/information-technology/glossary/ib-integration-broker>.

GOOGLE. Como refatorar um monolítico em microsserviços. 2021. [Online; accessed 11-Apr-2022]. Available from Internet: <https://cloud.google.com/architecture/microservices-architecture-refactoring-monoliths>.

GOOGLE. O que é uma máquina virtual? 2022. [Online; accessed 07-Apr-2022]. Available from Internet: <https://cloud.google.com/learn/what-is-a-virtual-machine#:~:text=Virtual\%20machine\%20defined,from\%20software\%20called\%20a\%20hypervisor>.

GOOGLE. Práticas recomendadas de criação de esquema. 2022. [Online; accessed 05-Apr-2022]. Available from Internet: <https://cloud.google.com/spanner/docs/schema-design#primary-key-prevent-hotspots>.

HAWKINS, M. W. **High Availability: Design, Techniques and Processes**. Prentice Hall, 2000. ISBN 978-0137141197. Available from Internet: <https://www.amazon.com/High-Availability-Design-Techniques-Processes/dp/013714119X>.

HILL, M. D. What is scalability? **SIGARCH Comput. Archit. News**, Association for Computing Machinery, New York, NY, USA, v. 18, n. 4, p. 18–21, dec 1990. ISSN 0163-5964. Available from Internet: <https://doi.org/10.1145/121973.121975>.

HOLORI. Outstanding cloud market size growth: Aws vs azure vs gcp market share in 2021. 2021. [Online; accessed 12-Apr-2022]. Available from Internet: <https://holori.com/2021-cloud-market-size-and-aws-azure-gcp-market-share/#:~:text=Google\%20cloud\%20revenue\%20climbed\%20nearly,\%2418\%20billion\%20revenue\%20in\%202021>.

IVANOV, A. Kubernetes vs. serverless: When to use and how to choose? part 1. 2021. [Online; accessed 09-Apr-2022]. Available from Internet: <https://dysnix.com/blog/kubernetes-vs-serverless-part-1/>.

KIRBAS, S. et al. The effect of evolutionary coupling on software defects: An industrial case study on a legacy system. In: . New York, NY, USA: Association for Computing Machinery, 2014. (ESEM '14). ISBN 9781450327749. Available from Internet: <https://doi.org/10.1145/2652524.2652577>.

KORAB, J. **Understanding Message Brokers**. [S.l.]: O'Reilly Media, 2017. ISBN 9781491981535.

KUBERNETES. Horizontal pod autoscaling. 2022. [Online; accessed 10-Apr-2022]. Available from Internet: <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#how-does-a-horizontalpodautoscaler-work>.

KUBERNETES. Kubernetes documentation. 2022. [Online; accessed 09-Apr-2022]. Available from Internet: <https://kubernetes.io/docs/home/>.

LAROS, J. H. et al. **Energy-Efficient High Performance Computing: Measurement and Tuning**. [S.l.]: Springer Publishing Company, Incorporated, 2012. ISBN 1447144910.

MARTIN, R. C. **Clean Code: A Handbook of Agile Software Craftsmanship**. Prentice Hall PTR, 2008. ISBN 978-0132350884. Available from Internet: <https://www.amazon.com.br/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882>.

MEYER, B. **Object-Oriented Software Construction**: Second edition. Prentice Hall, 1997. ISBN 978-0136291558. Available from Internet: <https://www.amazon.com.br/Object-Oriented-Software-Construction-Book-CD-ROM/dp/0136291554>.

MICHAEL, M. et al. Scale-up x scale-out: A case study using nutch/lucene. In: **2007 IEEE International Parallel and Distributed Processing Symposium**. [S.l.: s.n.], 2007. p. 1–8.

MYERS, G. J. **Reliable software through composite design**. Petrocelli/Charter, 1975. ISBN 978-0884052845. Available from Internet: <https://www.amazon.com/Reliable-software-through-composite-design/dp/0884052842>.

NEWMAN, S. **Building Microservices: Designing Fine-Grained Systems**. O'Reilly Media, 2015. ISBN 978-1491950357. Available from Internet: <https://www.amazon.com.br/Building-Microservices-Sam-Newman/dp/1491950358>.

NEWMAN, S. **Monolith to Microservices**: Evolutionary patterns to transform your monolith. O'Reilly Media, 2019. ISBN 978-1492047841. Available from Internet: <https://www.amazon.com.br/Monolith-Microservices-Sam-Newman/dp/1492047848>.

PAUTASSO, C. et al. Microservices in practice, part 1: Reality check and service design. **IEEE Software**, v. 34, n. 1, p. 91–98, 2017.

REDHAT. What is serverless? 2017. [Online; accessed 08-Apr-2022]. Available from Internet: <https://www.redhat.com/en/topics/cloud-native-apps/what-is-serverless#pros-and-cons>.

REDMOND, E.; WILSON, J. R. **Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement**. [S.l.]: Pragmatic Bookshelf, 2012. ISBN 1934356921.

REGAL, G. L. Uma proposta de conversão de arquiteturas monolíticas para microsserviços visando redução de acoplamentos. UFRGS, 2021.

RICHARDSON, C. **Microservices Patterns: With Examples in Java**. Manning Publications, 2018. ISBN 978-1617294549. Available from Internet: <https://www.amazon.com.br/Microservice-Patterns-examples-Chris-Richardson/dp/1617294543>.

SIMFORM. Cloud pricing comparison 2022: Aws vs azure vs google cloud. 2022. [Online; accessed 13-Apr-2022]. Available from Internet: <https://www.simform.com/blog/compute-pricing-comparison-aws-azure-googlecloud/>.

SNYK. Jvm ecosystem report 2021. 2021. [Online; accessed 14-Apr-2022]. Available from Internet: <https://snyk.io/jvm-ecosystem-report-2021/>.

SPLUNK. Splunk named a leader in the 2021 gartner siem magic quadrant for the eighth time. 2021. [Online; accessed 09-Apr-2022]. Available from Internet: <https://www.splunk.com/en_us/blog/security/splunk-named-a-leader-in-the-2021-gartner-siem-magic-quadrant-for-the-eighth-time.html#:~:text=By\%20Splunk\%20July\%2006\%2C\%202021,history\%20of\%20the\%20SIEM\%20market>.

STATISTA. Cloud infrastructure services vendor market share worldwide from 4th quarter 2017 to 4th quarter 2021. 2021. [Online; accessed 11-Apr-2022]. Available from Internet: <https://www.statista.com/statistics/967365/worldwide-cloud-infrastructure-services-market-share-vendor/#:~:text=In\%20the\%20third\%20quarter\%20of,with\%20eight\%20percent\%20market\%20share>.

STATISTA. Ranking of the most popular database management systems worldwide, as of january 2022. 2022. [Online; accessed 02-Apr-2022]. Available from Internet: <https://www.statista.com/statistics/809750/worldwide-popularity-ranking-database-management-systems/>.

TECHOPEDIA. Scalability. 2017. [Online; accessed 20-Mar-2022]. Available from Internet: <https://www.techopedia.com/definition/9269/scalability>.

TIOBE. Tiobe index for april 2022. 2022. [Online; accessed 01-Apr-2022]. Available from Internet: <https://www.tiobe.com/tiobe-index/>.

UPADHYAY, A. How to choose a programming language for a project? 2021. [Online; accessed 01-Apr-2022]. Available from Internet: <https://www.geeksforgeeks.org/how-to-choose-a-programming-language-for-a-project/>.

WIKIPEDIA, C. Black friday (shopping). 2022. [Online; accessed 23-April-2022]. Available from Internet: <https://en.wikipedia.org/wiki/Black_Friday_(shopping)>.

YOURDON, E.; PRESS, Y.; CONSTANTINE, L. L. **Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design**. Pearson Education, 1999. ISBN 978-0138544713. Available from Internet: <https://www.amazon.com.br/Structured-Design-Fundamentals-Discipline-Computer/dp/0138544719>.

## APÊNDICE A — QUESTIONÁRIO DA AVALIAÇÃO POR ESPECIALISTAS DA ARQUITETURA DE MICROSSERVIÇOS

# Avaliação por especialistas de tecnologia - Arquitetura de micro serviços - TCC - Cassiano Jaeger Stradolini

Este formulário será utilizado como fonte de dados para a avaliação de uma arquitetura de micro serviços proposta por Cassiano Jaeger Stradolini como parte de seu projeto de conclusão de curso.

A primeira seção será focada em perguntas de perfil profissional, como maneira de entender as características e conhecimentos que o especialista possuí. A segunda será focada em perguntas relacionadas a arquitetura analisada atual e a proposta pelo trabalho.

Agradeço imensamente pelas respostas que você fornecer. Isso será muito importante para a conclusão e validação do projeto que foi desenvolvido.

OBS: Adicionarei todos que responderam aos agradecimentos <3 <3 <3

*Obrigatório

1. E-mail *

_____

Análise de perfil profissional

2. Qual a sua idade? *

_____

3. Quantos anos de experiência trabalhando com desenvolvimento? *

*Marcar apenas uma oval.*

- ◯ De 0 a 2 anos
- ◯ De 2 a 4 anos
- ◯ De 4 a 6 anos
- ◯ De 6 a 8 anos
- ◯ + de 8 anos

4. Grau de escolaridade *

*Marcar apenas uma oval.*

- ◯ Nível médio completo
- ◯ Nível superior incompleto
- ◯ Nível superior completo
- ◯ Pós-graduação incompleta
- ◯ Pós-graduação completa
- ◯ Outro: _____

5. Cargo na empresa em que trabalha *

*Marcar apenas uma oval.*

- ◯ Desenvolvedor - Junior/Pleno
- ◯ Desenvolvedor - Sênior
- ◯ Lider tecnico
- ◯ Arquiteto
- ◯ Gerente de desenvolvimento
- ◯ Outro: _____

6. Experiência com micro serviços *

*Marcar apenas uma oval.*

- ◯ Sim
- ◯ Não

7. Experiência com aplicações monolíticas *

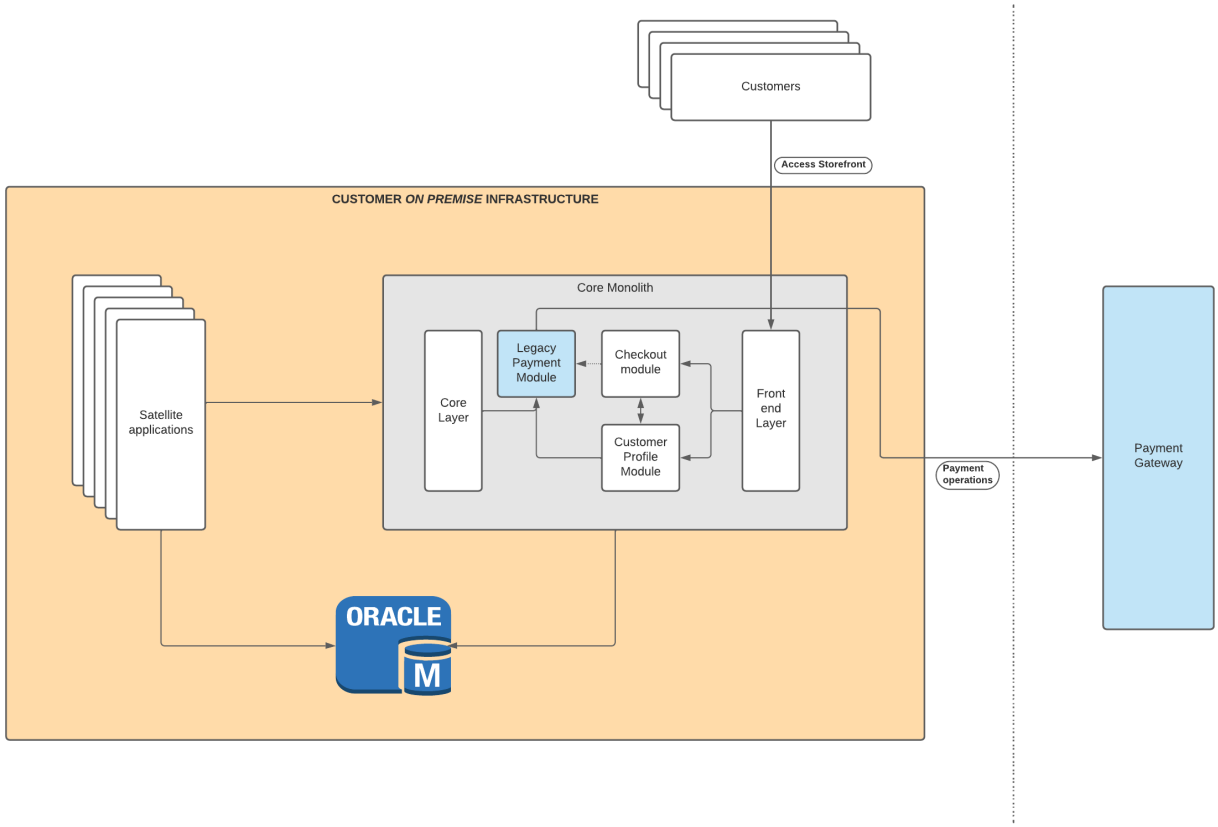
*Marcar apenas uma oval.*

- ◯ Sim
- ◯ Não

## Análise da arquitetura atual

A seção a seguir conterá informações sobre uma arquitetura monolítica e uma proposta de migração para uma arquitetura de micro serviços. Dadas essas informações, por favor responda as seguintes perguntas.

### Diagrama da arquitetura atual



## Descrição da arquitetura atual

O monólito principal é uma aplicação Java 7 EE, desenvolvida e mantida a mais de 15 anos pela empresa e tem como foco principal ser o motor das regras de negócios de todo o e-Commerce. Esse motor possui diversas integrações com sistemas externos e aplicações satélites. A aplicação ainda recebe incrementos de funcionalidades ainda hoje e está em processo de deprecação.

Possui uma API REST para diversas integrações que o sistema possui com outras aplicações. Aplicações que fornecem serviços como apresentação de dados, métricas de clientes e serviços de envio de e-mail são alguns exemplos. A aplicação possui um sistema de autenticação BASIC AUTH, o que significa que o cliente da aplicação deve passar no header da requisição o ID e a senha concatenados pelo caractere dois pontos (:) e codificados em base64.

A aplicação em análise possui mais de 15 anos de existência e sua base de código é bastante amarrada. Diversos times de desenvolvimento trabalharam nesta base, e muitas vezes simultaneamente. Como visto anteriormente, o processo de qualidade de código não era muito organizado, cada time de desenvolvimento tinha seus padrões e práticas de qualidade. Alguns, no entanto, não tinham essa cultura madura. Portanto existem diversos métodos ou classes que possuem muitas responsabilidades e dependências fortes entre si, tornando sua manutenção complexa e custosa.

A aplicação é hospedada em uma infraestrutura interna do cliente. Por conta da infraestrutura de servidores on premise, é necessário um processo de deploy customizado para a realidade e estrutura disponível. O processo atual de deployment da aplicação consiste na geração dos arquivos executáveis através de uma pipeline automática gerenciada pelo github. Após gerado os artefatos, é preciso realizar o deployment do mesmo nos servidores através de uma ferramenta legado.

## Dado a descrição acima e levando em consideração a sua experiência com aplicações desse tipo, responda as perguntas abaixo:

8. Qual a sua opinião sobre a coesão e acoplamento da arquitetura acima? *

9. Descreva suas percepções em relação a manutenção dessa aplicação? Quais a dificuldades e possíveis facilidades que esse tipo de arquitetura provém em relação a manutenibilidade da base de código? *

10. Como você avalia a performance e escalabilidade dessa arquitetura? Quais os principais pontos positivos e negativos desse tópico nessa arquitetura? *

11. Em relação a disponibilidade da aplicação. Quais pontos são positivos ou negativos dessa arquitetura? *

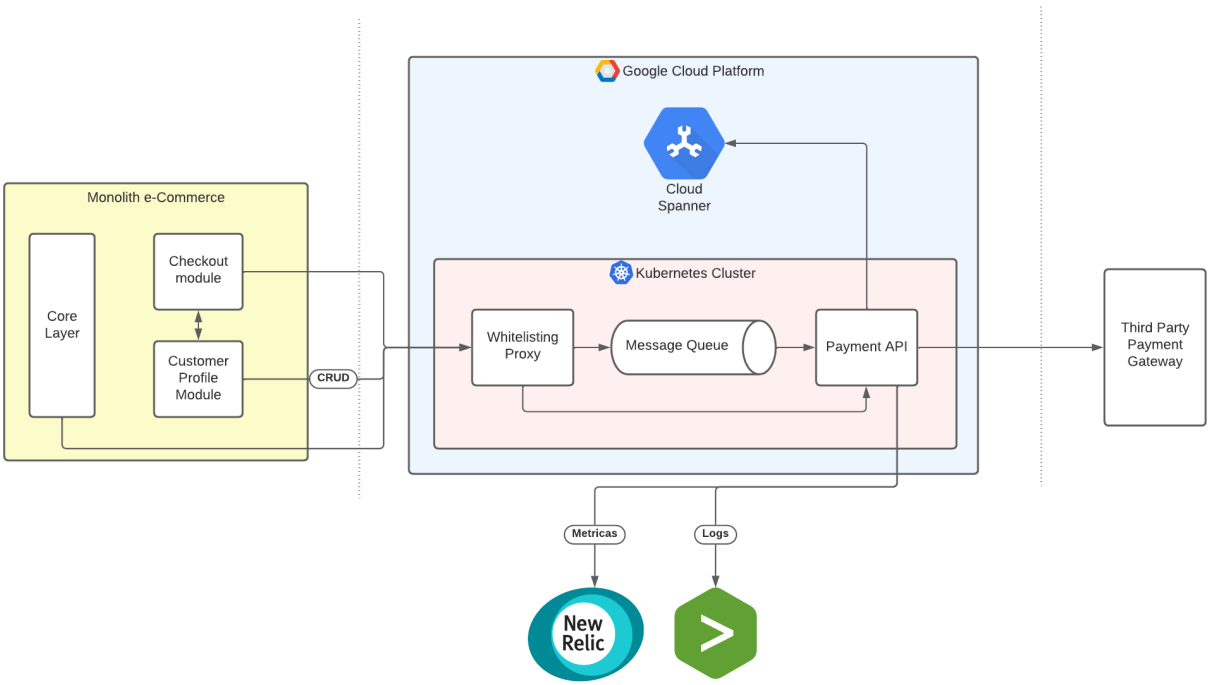
12. O que você acha desse tipo processo de deploy da aplicação analisada? *

13. Algum comentário extra? *

## Análise da arquitetura proposta

Esta seção terá como enfoque uma arquitetura de micro serviços proposta para fazer a migração do monólito. Esse trabalho abrange apenas o módulo de pagamento, indicado no diagrama abaixo como Legacy Payment Module (Caixa cinza descrita dentro do componente do monólito). Então todos os detalhes abaixo são referentes a esse módulo de pagamento migrado.

## Diagrama da arquitetura proposta



## Descrição da arquitetura proposta

A arquitetura proposta possui grandes diferenças em tecnologias e  abordagens em relação à arquitetura monolítica atual.

A implementação da proposta de arquitetura irá utilizar Java 18 + Spring Boot/Cloud como framework de desenvolvimento web e micro serviços.

Em relação ao cluster de Kubernetes: Será criado um pod para o micro serviço, que ficará exposto ao mundo exterior para que seja consumido pelos serviços e aplicações através de um recurso de Ingress. O motor que irá executar e gerenciar o cluster desenvolvido será o GKE, motor de kubernetes da GCP. Será utilizado uma estrutura de listas de permissão através de um proxy. Apenas os serviços permitidos poderão fazer contato com o micro serviço desenvolvido. O proxy irá trabalhar como um verificador de permissão, onde irá permitir apenas o recebimento de requisições de aplicações confiáveis.

Para o balanceamento de carga será utilizado o recurso padrão do Kubernetes para balanceamento de carga da aplicação, o Horizontal Pod Autoscaler. O valor de quantidade de réplicas mínimas será de 1, o valor de quantidade de réplicas máximas será de 10 e a métrica analisada para disparar o escalonamento automático será de porcentagem de CPU utilizada acima de 50%.

As mensagens recebidas serão entregues ao serviço de mensageria KubeMQ através de um proxy. Esse proxy será responsável por permitir apenas conexões de hosts confiáveis.

Google Cloud Spanner foi o banco de dados escolhido devido a suas características SQL distribuído, através de data sharding.

O cluster estará integrado com uma solução de agregador de Logs, o Splunk e outra solução de coleta e análise métricas, o New Relic.

Por fim, o cluster terá um CI/CD gerenciado por aplicações como Harness/CircleCI/Google Build. O processo de deploy será automático nos clusters GKE, apenas necessitando o trigger da pipeline.

## Dado a descrição acima e levando em consideração a sua experiência com aplicações desse tipo, responda as perguntas abaixo:

14. Qual a sua opinião sobre a coesão e acoplamento da arquitetura acima? Você vê vantagens ou desvantagens em relação a arquitetura anterior? *

15. Descreva suas percepções em relação a manutenção dessa aplicação? Você vê vantagens ou desvantagens em relação a arquitetura anterior? *

16. Como você avalia a performance e escalabilidade dessa arquitetura Quais os principais pontos positivos e negativos desse tópico nessa arquitetura? Você vê vantagens ou desvantagens em relação a arquitetura anterior? *

17. Em relação a disponibilidade da aplicação. Quais pontos são positivos ou negativos dessa arquitetura? Você vê vantagens ou desvantagens em relação a arquitetura anterior? *

18. O que você acha desse tipo processo de deploy da aplicação analisada? Você vê vantagens ou desvantagens em relação a arquitetura anterior? *

19. Algum comentário extra? *

**APÊNDICE B — RESPOSTAS DOS QUESTIONÁRIOS DE AVALIAÇÃO POR ESPECIALISTAS DA ARQUITETURA DE MICROSSERVIÇOS**

# Avaliação por especialistas de tecnologia - Arquitetura de micro serviços - TCC - Cassiano Jaeger Stradolini

Este formulário será utilizado como fonte de dados para a avaliação de uma arquitetura de micro serviços proposta por Cassiano Jaeger Stradolini como parte de seu projeto de conclusão de curso.

A primeira seção será focada em perguntas de perfil profissional, como maneira de entender as características e conhecimentos que o especialista possuí. A segunda será focada em perguntas relacionadas a arquitetura analisada atual e a proposta pelo trabalho.

Agradeço imensamente pelas respostas que você fornecer. Isso será muito importante para a conclusão e validação do projeto que foi desenvolvido.

OBS: Adicionarei todos que responderam aos agradecimentos <3 <3 <3

E-mail *

marlom.oliveira@e-core.com

## Análise de perfil profissional

Qual a sua idade? *

31

Quantos anos de experiência trabalhando com desenvolvimento? *

De 6 a 8 anos

Grau de escolaridade *

- ( ) Nível médio completo
- ( ) Nível superior incompleto
- ( ) Nível superior completo
- ( ) Pós-graduação incompleta
- (•) Pós-graduação completa
- ( ) Outro: .....

Cargo na empresa em que trabalha *

- ( ) Desenvolvedor - Junior/Pleno
- (•) Desenvolvedor - Sênior
- ( ) Lider tecnico
- ( ) Arquiteto
- ( ) Gerente de desenvolvimento
- ( ) Outro: .....

Experiência com micro serviços *

- (•) Sim
- ( ) Não

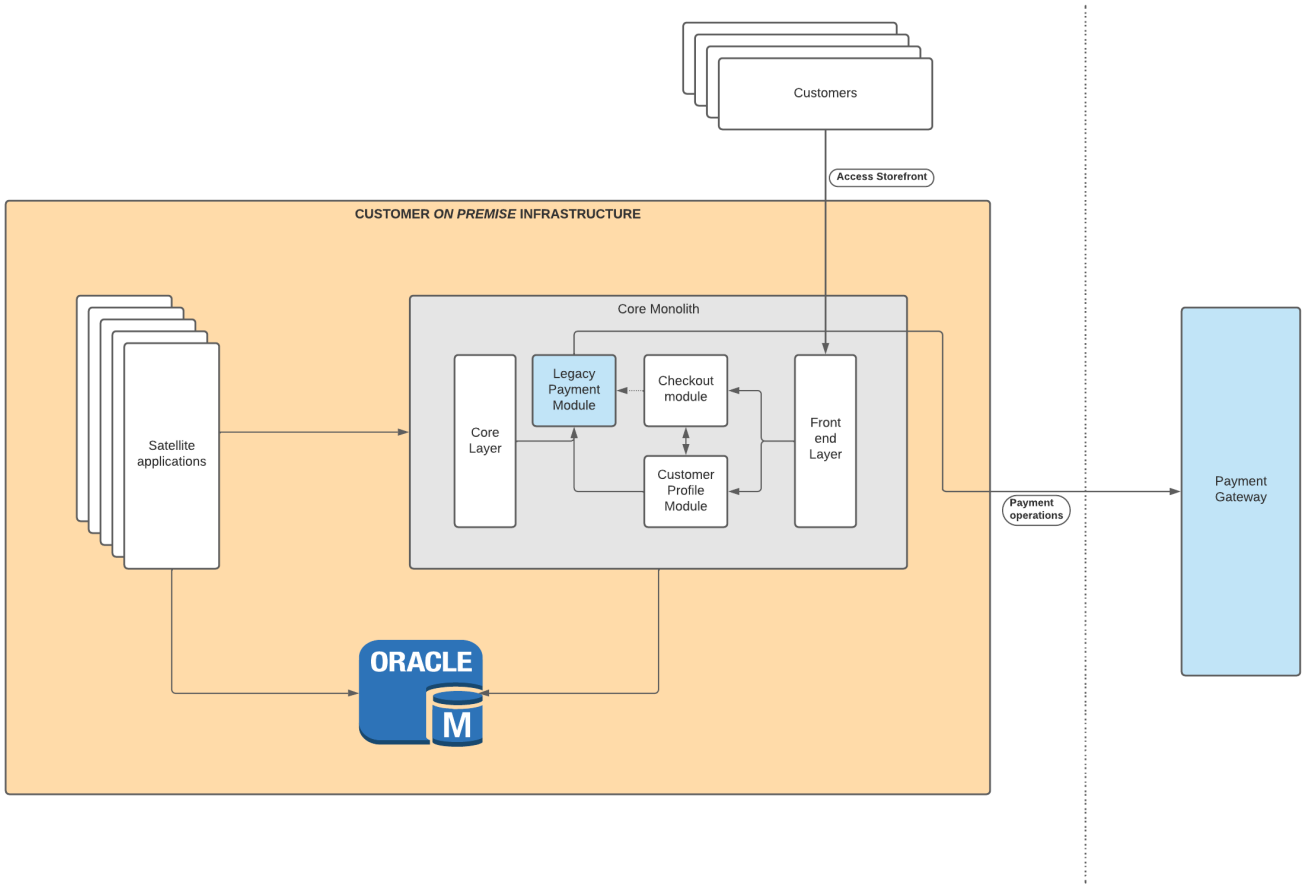
Experiência com aplicações monolíticas *

- (●) Sim
- ( ) Não

## Análise da arquitetura atual

A seção a seguir conterá informações sobre uma arquitetura monolítica e uma proposta de migração para uma arquitetura de micro serviços. Dadas essas informações, por favor responda as seguintes perguntas.

Diagrama da arquitetura atual



## Descrição da arquitetura atual

O monólito principal é uma aplicação Java 7 EE, desenvolvida e mantida a mais de 15 anos pela empresa e tem como foco principal ser o motor das regras de negócios de todo o e-Commerce. Esse motor possui diversas integrações com sistemas externos e aplicações satélites. A aplicação ainda recebe incrementos de funcionalidades ainda hoje e está em processo de deprecação.

Possui uma API REST para diversas integrações que o sistema possui com outras aplicações. Aplicações que fornecem serviços como apresentação de dados, métricas de clientes e serviços de envio de e-mail são alguns exemplos. A aplicação possui um sistema de autenticação BASIC AUTH, o que significa que o cliente da aplicação deve passar no header da requisição o ID e a senha concatenados pelo caractere dois pontos (:) e codificados em base64.

A aplicação em análise possui mais de 15 anos de existência e sua base de código é bastante amarrada. Diversos times de desenvolvimento trabalharam nesta base, e muitas vezes simultaneamente. Como visto anteriormente, o processo de qualidade de código não era muito organizado, cada time de desenvolvimento tinha seus padrões e práticas de qualidade. Alguns, no entanto, não tinham essa cultura madura. Portanto existem diversos métodos ou classes que possuem muitas responsabilidades e dependências fortes entre si, tornando sua manutenção complexa e custosa.

A aplicação é hospedada em uma infraestrutura interna do cliente. Por conta da infraestrutura de servidores on premise, é necessário um processo de deploy customizado para a realidade e estrutura disponível. O processo atual de deployment da aplicação consiste na geração dos arquivos executáveis através de uma pipeline automática gerenciada pelo github. Após gerado os artefatos, é preciso realizar o deployment do mesmo nos servidores através de uma ferramenta legado.

**Dado a descrição acima e levando em consideração a sua experiência com aplicações desse tipo, responda as perguntas abaixo:**

Qual a sua opinião sobre a coesão e acoplamento da arquitetura acima? *

A coesão não é necessariamente um problema, diferente do alto acoplamento de componentes. Criar um número grande de dependências entre componentes põe o projeto de software na fila da UTI.

Descreva suas percepções em relação a manutenção dessa aplicação? Quais a dificuldades e possíveis facilidades que esse tipo de arquitetura provém em relação a manutenibilidade da base de código? *

Para um sistema de 15 anos de idade que possui esta estrutura, deve ter um enorme efeito cascata nos demais componentes para qualquer mudança em qualquer parte do sistema. Suspeito que as maiores dificuldades sejam em isolar tais mudanças para evitar efeitos indesejados, e também em testá-las. Não consigo enxergar facilidades que um sistema monolito de 15 anos, sem sistematização de processo de qualidade inclusive, traga em relação a manutenibilidade de código. Deve ser dificil até para quem tá há 15 anos atuando nele e conhece tudo, pois, em tese, tem a visão dos impactos de qualquer mudança.

Como você avalia a performance e escalabilidade dessa arquitetura? Quais os principais pontos positivos e negativos desse tópico nessa arquitetura? *

Não consigo avaliar a performance, mas se este sistema se comunica apenas com um DB, talvez ele comporte escalabilidade horizontal. Apesar de que cada instância deveria requerer uma boa quantidade de poder de processamento. Logo, a escalabilidade horizontal, apesar de aplicável, seja cara. Todavia, não é uma impossibilidade técnica, apenas restrinção financeira. Por vim, várias instâncias se comunicando com um único DB deva funcionar bem. Não tão rápido, talvez, por concorrência por um único DB, mas escalar o DB pode ser endereçado separadamente.

Em relação a disponibilidade da aplicação. Quais pontos são positivos ou negativos dessa arquitetura? *

A disponibilidade é frágil. Uma falha em qualquer um dos componentes põe em risco todo o sistema. Suspeito que o processo de inicialização/re-inicialização dele não seja rápido dado o tamanho.

O que você acha desse tipo processo de deploy da aplicação analisada? *

O texto não fala se há um ambiente de homologação que antecede o ambiente de produção. Assumindo que a falta da descrição signifique também a inexistência do ambiente de homologação, então o processo de lançamento é arriscado. Executar em produção o compilado de 15 anos com alto acoplamento, feito por várias pessoas distintas, de times distintos, que talvez não se conversam, e sem um processo de qualidade agregado  considero bastante arriscado.

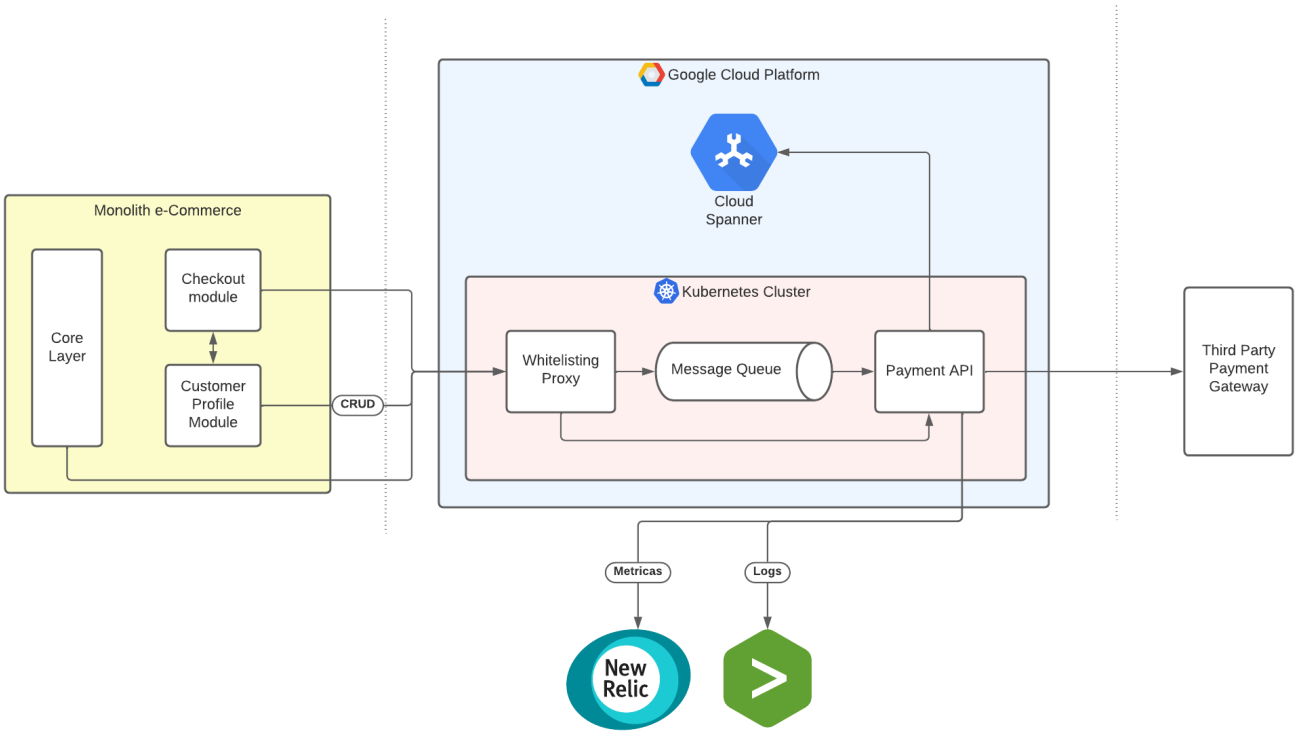
Algum comentário extra? *

É importante destacar que o problema deste sistema não é exclusivamente de sua arquitetura monolita. Há um conjunto de fatores que fazem dele uma espécie de bomba-relógio. Mesmo microserviços mal construídos são sistemas ruins e de baixa qualidade. Qualquer software capado de boas decisões e um processo sistemático de qualidade tendem ao risco técnico, não importa a arquitetura.

## Análise da arquitetura proposta

Esta seção terá como enfoque uma arquitetura de micro serviços proposta para fazer a migração do monólito. Esse trabalho abrange apenas o módulo de pagamento, indicado no diagrama abaixo como Legacy Payment Module (Caixa cinza descrita dentro do componente do monólito). Então todos os detalhes abaixo são referentes a esse módulo de pagamento migrado.

## Diagrama da arquitetura proposta



## Descrição da arquitetura proposta

A arquitetura proposta possui grandes diferenças em tecnologias e  abordagens em relação à arquitetura monolítica atual.

A implementação da proposta de arquitetura irá utilizar Java 18 + Spring Boot/Cloud como framework de desenvolvimento web e micro serviços.

Em relação ao cluster de Kubernetes: Será criado um pod para o micro serviço, que ficará exposto ao mundo exterior para que seja consumido pelos serviços e aplicações através de um recurso de Ingress. O motor que irá executar e gerenciar o cluster desenvolvido será o GKE, motor de kubernetes da GCP. Será utilizado uma estrutura de listas de permissão através de um proxy. Apenas os serviços permitidos poderão fazer contato com o micro serviço desenvolvido. O proxy irá trabalhar como um verificador de permissão, onde irá permitir apenas o recebimento de requisições de aplicações confiáveis.

Para o balanceamento de carga será utilizado o recurso padrão do Kubernetes para balanceamento de carga da aplicação, o Horizontal Pod Autoscaler. O valor de quantidade de réplicas mínimas será de 1, o valor de quantidade de réplicas máximas será de 10 e a métrica analisada para disparar o escalonamento automático será de porcentagem de CPU utilizada acima de 50%.

As mensagens recebidas serão entregues ao serviço de mensageria KubeMQ através de um proxy. Esse proxy será responsável por permitir apenas conexões de hosts confiáveis.

Google Cloud Spanner foi o banco de dados escolhido devido a suas características SQL distribuído, através de data sharding.

O cluster estará integrado com uma solução de agregador de Logs, o Splunk e outra solução de coleta e análise métricas, o New Relic.

Por fim, o cluster terá um CI/CD gerenciado por aplicações como Harness/CircleCI/Google Build. O processo de deploy será automático nos clusters GKE, apenas necessitando o trigger da pipeline.

## Dado a descrição acima e levando em consideração a sua experiência com aplicações desse tipo, responda as perguntas abaixo:

Qual a sua opinião sobre a coesão e acoplamento da arquitetura acima? Você vê vantagens ou desvantagens em relação a arquitetura anterior? *

A coesão continua a mesma. Mudou apenas o acoplamento. Temos agora dois sistemas separados que podem ser tratados de formas separadas. A vantagem é poder isolar os problemas oriundos de pagamentos e também suas mudanças, de modo a não afetar o sistema maior. A desvantagem é que agora temos 2 sistemas para cuidar e garantir a comunicação entre eles.

Descreva suas percepções em relação a manutenção dessa aplicação? Você vê vantagens ou desvantagens em relação a arquitetura anterior? *

Percebo que houve apenas a extração do componente de pagamento, e implantação do mesmo numa nova plataforma de execução. O uso de uma plataforma diferente para lançamento e execução, e que auto-gerencia a instância, não é parte da arquitetura do componente de pagamentos. Este, inclusive, ganhou dois novos módulos: (1) Whitelisting Proxy; (2) Message Qeue. Logo, olhando estritamente para o componente de pagamentos, ele aumentou de tamanho em comparação com seu  lugar de origem.

Como você avalia a performance e escalabilidade dessa arquitetura Quais os principais pontos positivos e negativos desse tópico nessa arquitetura? Você vê vantagens ou desvantagens em relação a arquitetura anterior? *

Do ponto de vista de performance, agora consumidor do componente de pagamentos não será afetado por possíveis lentidões da versão isolada do componente de pagamentos. Capacidade ganhar a partir do uso do serviço de messageria. Do ponto de vista de escalabilidade, agora é possível escalar isoladamente o componente de pagamentos. Esta escala pode ser vertical ou horizontal. Sendo horizontal (mais instâncias), as novas máquinas podem ter menor poder computacional (mais barato de escalar em comparação com a versão anterior). A desvantagem está na coordenação de três novos módulos para que o componente de pagamento funcione de forma esperada. Novamente, ele aumentou de tamanho.

Em relação a disponibilidade da aplicação. Quais pontos são positivos ou negativos dessa arquitetura? Você vê vantagens ou desvantagens em relação a arquitetura anterior? *

A garantia da disponibilidade aqui não tem relação com a arquitetura do componente, seja ela monolita ou não. Tem relação com a nova plataforma de execução do componente. Ela trouxe este benefício. Com o uso do Kubernetes, inclusive, ganha-se a automatização da escalabilidade, verificação de saúde das instâncias do componente e distribuidor de carga entre as instâncias.

O que você acha desse tipo processo de deploy da aplicação analisada? Você vê vantagens ou desvantagens em relação a arquitetura anterior? *

Apesar de usar ferramentas novas, o novo processo padece do mesmo problema do anterior. Aqui, novamente, pela ausência na descrição, estou entendendo também a ausência do recurso. Não há ambiente de homologação. Está tão fácil quanto antes pôr uma nova versão do componente em produção. Talvez até mais fácil uma vez que todo o processo está automatizado. O risco de problemas está à distância de um clique.

Algum comentário extra? *

Extendendo o que compentei anteriormente, o que faz um bom software não é a arquitetura apenas, mas um conjunto de fatores. Aqui, apesar do uso de novas tecnologias, não podemos concluir diretamente que, por conta delas, temos um software melhor.

É evidente que teve mudanças positivas (isolamento de efeitos colaterais, escalabilidade isolada do componente, barateamento da escalabilidade, assincronia na execução devido uso de serviço de messageria, etc), mas também ouve o aumento no componente, com a inclusão de proxy e o serviço de messageria.

Gostaria de destacar também que houve uma leve confusão na descrição dos recursos do Kubernetes. O AutoScaler é responsável apenas pela escalabilidade, o LoadBalancer quem é responsável pela distribuição de carga entre as possíveis múltiplas instâncias do componente.

Não está descrito no contexto a abrangência do serviço do e-Commerce. A literatura aponta que serviços de e-Commerce requerem um banco de dados com suporte ao ACID (os 4 pilares de uma transação), o que é suportado pelo banco de dados anterior. Temos agora o Google Cloud Spanner no novo desenho, que, apesar de suportar o ACID, ele é um banco de escala global. É também o banco de dados mais caro da Google. Assumindo novamente que a ausência desse detalhe no contexto significa a ausência da necessidade, talvez o Google Cloud SQL dê conta da demanda.

Por fim, o desenho trás melhorias. Mas, para dar mérito devido a cada coisa, parte das melhorias veio da isolação do componente de pagamento e parte da plataforma de execução escolhida. Não necessariamente é mérito de usar uma arquitetura de microserviço neste caso. Como antes, também, o calcanhar de aquiles do desenho anterior não era exclusivamente do fato do sistema ser monolito.

# Avaliação por especialistas de tecnologia - Arquitetura de micro serviços - TCC - Cassiano Jaeger Stradolini

Este formulário será utilizado como fonte de dados para a avaliação de uma arquitetura de micro serviços proposta por Cassiano Jaeger Stradolini como parte de seu projeto de conclusão de curso.

A primeira seção será focada em perguntas de perfil profissional, como maneira de entender as características e conhecimentos que o especialista possuí. A segunda será focada em perguntas relacionadas a arquitetura analisada atual e a proposta pelo trabalho.

Agradeço imensamente pelas respostas que você fornecer. Isso será muito importante para a conclusão e validação do projeto que foi desenvolvido.

OBS: Adicionarei todos que responderam aos agradecimentos <3 <3 <3

E-mail *

rafaelperfeito_07@hotmail.com

## Análise de perfil profissional

Qual a sua idade? *

29

Quantos anos de experiência trabalhando com desenvolvimento? *

De 4 a 6 anos

Grau de escolaridade *

- ( ) Nível médio completo
- ( ) Nível superior incompleto
- (•) Nível superior completo
- ( ) Pós-graduação incompleta
- ( ) Pós-graduação completa
- ( ) Outro: _____

Cargo na empresa em que trabalha *

- (•) Desenvolvedor - Junior/Pleno
- ( ) Desenvolvedor - Sênior
- ( ) Lider tecnico
- ( ) Arquiteto
- ( ) Gerente de desenvolvimento
- ( ) Outro: _____

Experiência com micro serviços *

- ( ) Sim
- (•) Não

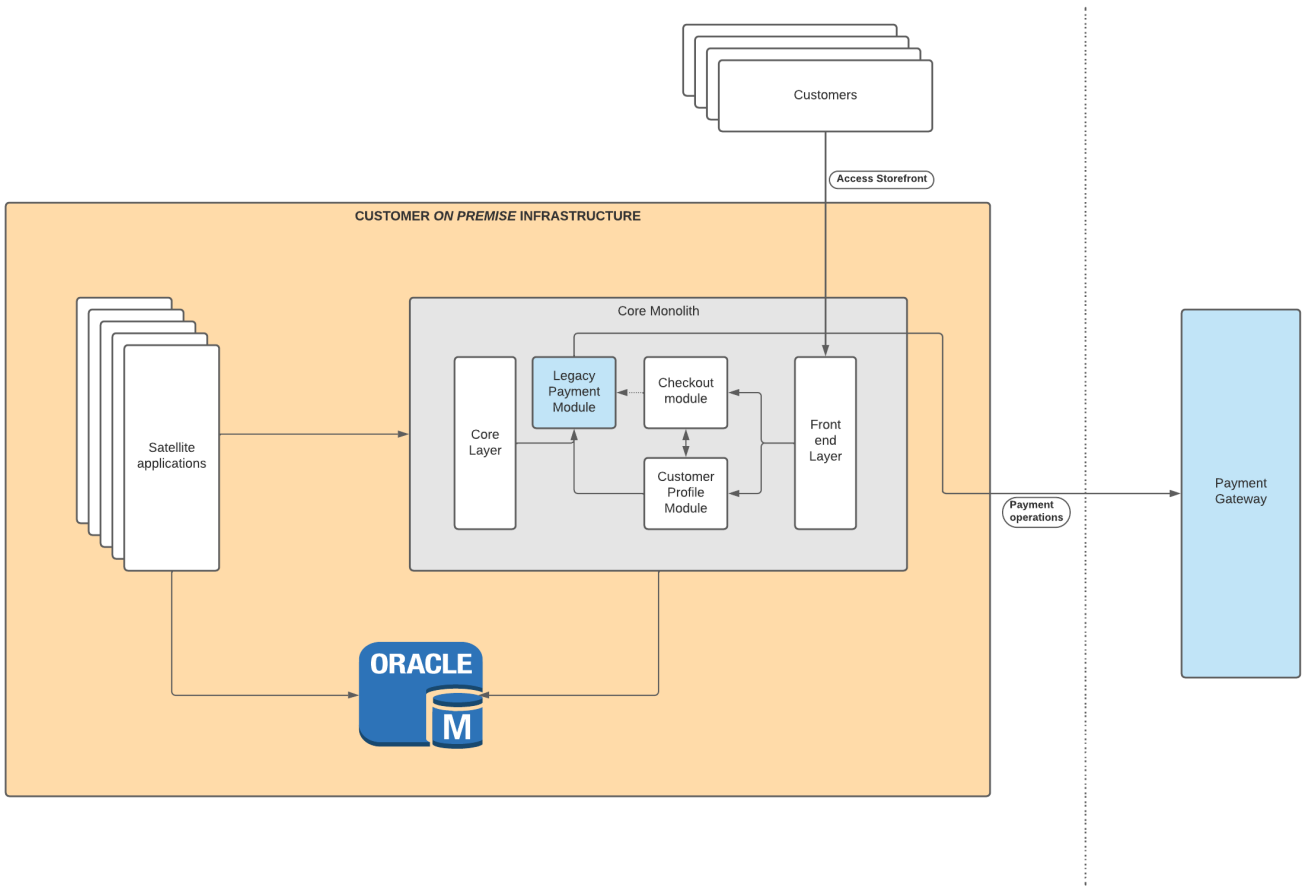
Experiência com aplicações monolíticas *

- (●) Sim
- ( ) Não

## Análise da arquitetura atual

A seção a seguir conterá informações sobre uma arquitetura monolítica e uma proposta de migração para uma arquitetura de micro serviços. Dadas essas informações, por favor responda as seguintes perguntas.

### Diagrama da arquitetura atual



## Descrição da arquitetura atual

O monólito principal é uma aplicação Java 7 EE, desenvolvida e mantida a mais de 15 anos pela empresa e tem como foco principal ser o motor das regras de negócios de todo o e-Commerce. Esse motor possui diversas integrações com sistemas externos e aplicações satélites. A aplicação ainda recebe incrementos de funcionalidades ainda hoje e está em processo de deprecação.

Possui uma API REST para diversas integrações que o sistema possui com outras aplicações. Aplicações que fornecem serviços como apresentação de dados, métricas de clientes e serviços de envio de e-mail são alguns exemplos. A aplicação possui um sistema de autenticação BASIC AUTH, o que significa que o cliente da aplicação deve passar no header da requisição o ID e a senha concatenados pelo caractere dois pontos (:) e codificados em base64.

A aplicação em análise possui mais de 15 anos de existência e sua base de código é bastante amarrada. Diversos times de desenvolvimento trabalharam nesta base, e muitas vezes simultaneamente. Como visto anteriormente, o processo de qualidade de código não era muito organizado, cada time de desenvolvimento tinha seus padrões e práticas de qualidade. Alguns, no entanto, não tinham essa cultura madura. Portanto existem diversos métodos ou classes que possuem muitas responsabilidades e dependências fortes entre si, tornando sua manutenção complexa e custosa.

A aplicação é hospedada em uma infraestrutura interna do cliente. Por conta da infraestrutura de servidores on premise, é necessário um processo de deploy customizado para a realidade e estrutura disponível. O processo atual de deployment da aplicação consiste na geração dos arquivos executáveis através de uma pipeline automática gerenciada pelo github. Após gerado os artefatos, é preciso realizar o deployment do mesmo nos servidores através de uma ferramenta legado.

**Dado a descrição acima e levando em consideração a sua experiência com aplicações desse tipo, responda as perguntas abaixo:**

Qual a sua opinião sobre a coesão e acoplamento da arquitetura acima? *

São princípios aparentemente inexistentes ou pouco utilizados no cenário atual, tendo em vista que as classes possuem várias responsabilidades e dependências fortes entre si.

Descreva suas percepções em relação a manutenção dessa aplicação? Quais a dificuldades e possíveis facilidades que esse tipo de arquitetura provém em relação a manutenibilidade da base de código? *

A manutenção e gerenciamento dessas classes são complexos e custosos, considerando que por exemplo uma mudança em uma classe irá afetar toda a cadeia de classes.

Como você avalia a performance e escalabilidade dessa arquitetura? Quais os principais pontos positivos e negativos desse tópico nessa arquitetura? *

Negativo:A escalabilidade on premise acaba sendo mais demorada e difícil de realizar, pois para escalar e necessário comprar mais hardware e software, considerar tempo de entrega, implementação, coisas que não ocorrem por exemplo ao utilizar um ambiente cloud.
Positivo: Não consigo pensar em algum.

Em relação a disponibilidade da aplicação. Quais pontos são positivos ou negativos dessa arquitetura? *

Negativo: Por tratar-se de servidores on premisse diversos fatores podem afetar a disponibilidade do sistema (queda de luz, internet, a queima de algum equipamento) considerando que normalmente os servidores ficam todos centralizados em um mesmo espaço físico a tendência é que caso ocorra algum problema isso afete todo a infraestrutura e deixe o sistema indisponível.

O que você acha desse tipo processo de deploy da aplicação analisada? *

É um processo defasado, demorado e custoso, tendo em vista que é necessário criar, e principalmente manter, diversos processos de deploy customizados para cada tipo de estrutura onde a aplicação será utilizada.

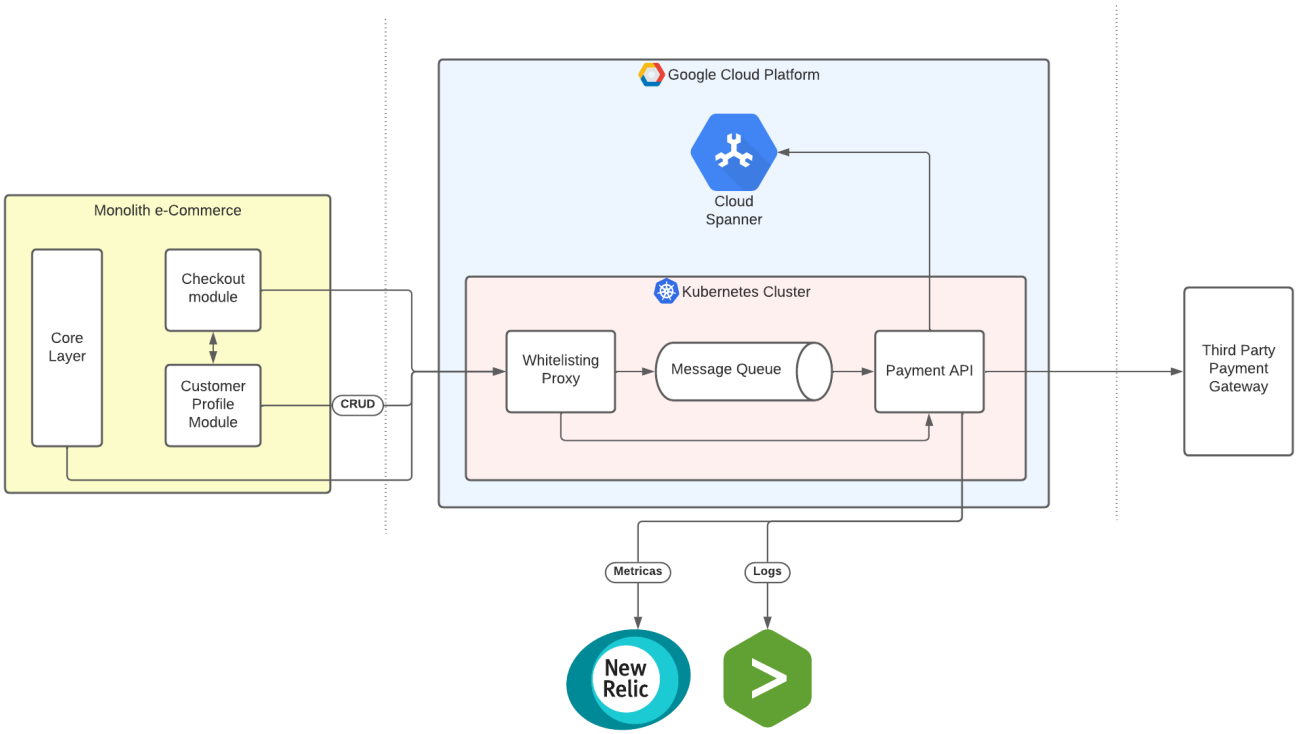
Algum comentário extra? *

Sistemas legados nesse tipo de estrutura acabam gerando diversos custos em função da complexidade de manutenção de código, dificuldade e demora no processo de deploy e atualização das aplicações, bem como problemas e perdas que podem ser ocasinadas pela indisponibilidade do mesmo.
Um estudo/análise mais detalhada para a elaboração de uma proposta de migração para micro serviços faz-se necessária e bastante útil aos interesses da empresa, uma vez que possibilita reduzir diversas características negativas existentes hoje na aplicação.

## Análise da arquitetura proposta

Esta seção terá como enfoque uma arquitetura de micro serviços proposta para fazer a migração do monólito. Esse trabalho abrange apenas o módulo de pagamento, indicado no diagrama abaixo como Legacy Payment Module (Caixa cinza descrita dentro do componente do monólito). Então todos os detalhes abaixo são referentes a esse módulo de pagamento migrado.

## Diagrama da arquitetura proposta



## Descrição da arquitetura proposta

A arquitetura proposta possui grandes diferenças em tecnologias e  abordagens em relação à arquitetura monolítica atual.

A implementação da proposta de arquitetura irá utilizar Java 18 + Spring Boot/Cloud como framework de desenvolvimento web e micro serviços.

Em relação ao cluster de Kubernetes: Será criado um pod para o micro serviço, que ficará exposto ao mundo exterior para que seja consumido pelos serviços e aplicações através de um recurso de Ingress. O motor que irá executar e gerenciar o cluster desenvolvido será o GKE, motor de kubernetes da GCP. Será utilizado uma estrutura de listas de permissão através de um proxy. Apenas os serviços permitidos poderão fazer contato com o micro serviço desenvolvido. O proxy irá trabalhar como um verificador de permissão, onde irá permitir apenas o recebimento de requisições de aplicações confiáveis.

Para o balanceamento de carga será utilizado o recurso padrão do Kubernetes para balanceamento de carga da aplicação, o Horizontal Pod Autoscaler. O valor de quantidade de réplicas mínimas será de 1, o valor de quantidade de réplicas máximas será de 10 e a métrica analisada para disparar o escalonamento automático será de porcentagem de CPU utilizada acima de 50%.

As mensagens recebidas serão entregues ao serviço de mensageria KubeMQ através de um proxy. Esse proxy será responsável por permitir apenas conexões de hosts confiáveis.

Google Cloud Spanner foi o banco de dados escolhido devido a suas características SQL distribuído, através de data sharding.

O cluster estará integrado com uma solução de agregador de Logs, o Splunk e outra solução de coleta e análise métricas, o New Relic.

Por fim, o cluster terá um CI/CD gerenciado por aplicações como Harness/CircleCI/Google Build. O processo de deploy será automático nos clusters GKE, apenas necessitando o trigger da pipeline.

## Dado a descrição acima e levando em consideração a sua experiência com aplicações desse tipo, responda as perguntas abaixo:

Qual a sua opinião sobre a coesão e acoplamento da arquitetura acima? Você vê vantagens ou desvantagens em relação a arquitetura anterior? *

Percebo vantagens em relação ao modelo de arquitetura anterior, pois ao migrar a aplicação para micro serviços é possível desacoplar as funções entre as classes, fazendo cada uma responsável pelo seu papel e permitindo uma mais fácil reutilização das funções bem como uma maior simplicidade e agilidade na manutenção dos códigos fonte.
Micro serviços também permitem uma rápida escalabilidade e possuem boa sinergia com containers.

Descreva suas percepções em relação a manutenção dessa aplicação? Você vê vantagens ou desvantagens em relação a arquitetura anterior? *

Vantagens com relação a arquitetura anterior, pelos motivos citados de acima em termos de isolamento de funções, reusabilidade, legibilidade e manutenção são alguns dos exemplos de benefícios dessa migração.

Como você avalia a performance e escalabilidade dessa arquitetura Quais os principais pontos positivos e negativos desse tópico nessa arquitetura? Você vê vantagens ou desvantagens em relação a arquitetura anterior? *

Esse é outro ponto de vantagem em relação a arquitetura anterior, considerando que a arquitetura de micro serviços momento permite a escalabilidade e elasticidade em situações de alta volumetria, aumentando e reduzindo instâncias de serviços de conforme a necessidade, consumindo recursos físicos sob demanda e apenas para aquela funcionalidade (micro serviço) que está sendo muito requisitado no momento.

Em relação a disponibilidade da aplicação. Quais pontos são positivos ou negativos dessa arquitetura? Você vê vantagens ou desvantagens em relação a arquitetura anterior? *

Positivo: Menor risco de indisponibilidade do sistema, uma vez que a aplicação estará implementada através de vários micro serviços e não um único processo monolítico. Por serem recursos isolados um erro em um dos serviços não causa uma parada de todo o sistema e ainda é possível controlar para que ao ser detectada uma falha no serviço as requisições sejam transferidas para outra instância do serviço, evitando a parada das requisições.
Vantagens em relação ao modelo anterior.

O que você acha desse tipo processo de deploy da aplicação analisada? Você vê vantagens ou desvantagens em relação a arquitetura anterior? *

Otimiza o processo de deploy como um todo, permite uma automatização, maior agilidade e uma maior adaptabilidade, pois não é mais dependente dos diversos ambientes no cliente on premisse.

Algum comentário extra? *

Nada a declarar.

# Avaliação por especialistas de tecnologia - Arquitetura de micro serviços - TCC - Cassiano Jaeger Stradolini

Este formulário será utilizado como fonte de dados para a avaliação de uma arquitetura de micro serviços proposta por Cassiano Jaeger Stradolini como parte de seu projeto de conclusão de curso.

A primeira seção será focada em perguntas de perfil profissional, como maneira de entender as características e conhecimentos que o especialista possuí. A segunda será focada em perguntas relacionadas a arquitetura analisada atual e a proposta pelo trabalho.

Agradeço imensamente pelas respostas que você fornecer. Isso será muito importante para a conclusão e validação do projeto que foi desenvolvido.

OBS: Adicionarei todos que responderam aos agradecimentos <3 <3 <3

E-mail *

caetanoj.ss@gmail.com

## Análise de perfil profissional

Qual a sua idade? *

26

Quantos anos de experiência trabalhando com desenvolvimento? *

De 4 a 6 anos

Grau de escolaridade *

- ( ) Nível médio completo
- (•) Nível superior incompleto
- ( ) Nível superior completo
- ( ) Pós-graduação incompleta
- ( ) Pós-graduação completa
- ( ) Outro: .....

Cargo na empresa em que trabalha *

- (•) Desenvolvedor - Junior/Pleno
- ( ) Desenvolvedor - Sênior
- ( ) Lider tecnico
- ( ) Arquiteto
- ( ) Gerente de desenvolvimento
- ( ) Outro: .....

Experiência com micro serviços *

- (•) Sim
- ( ) Não

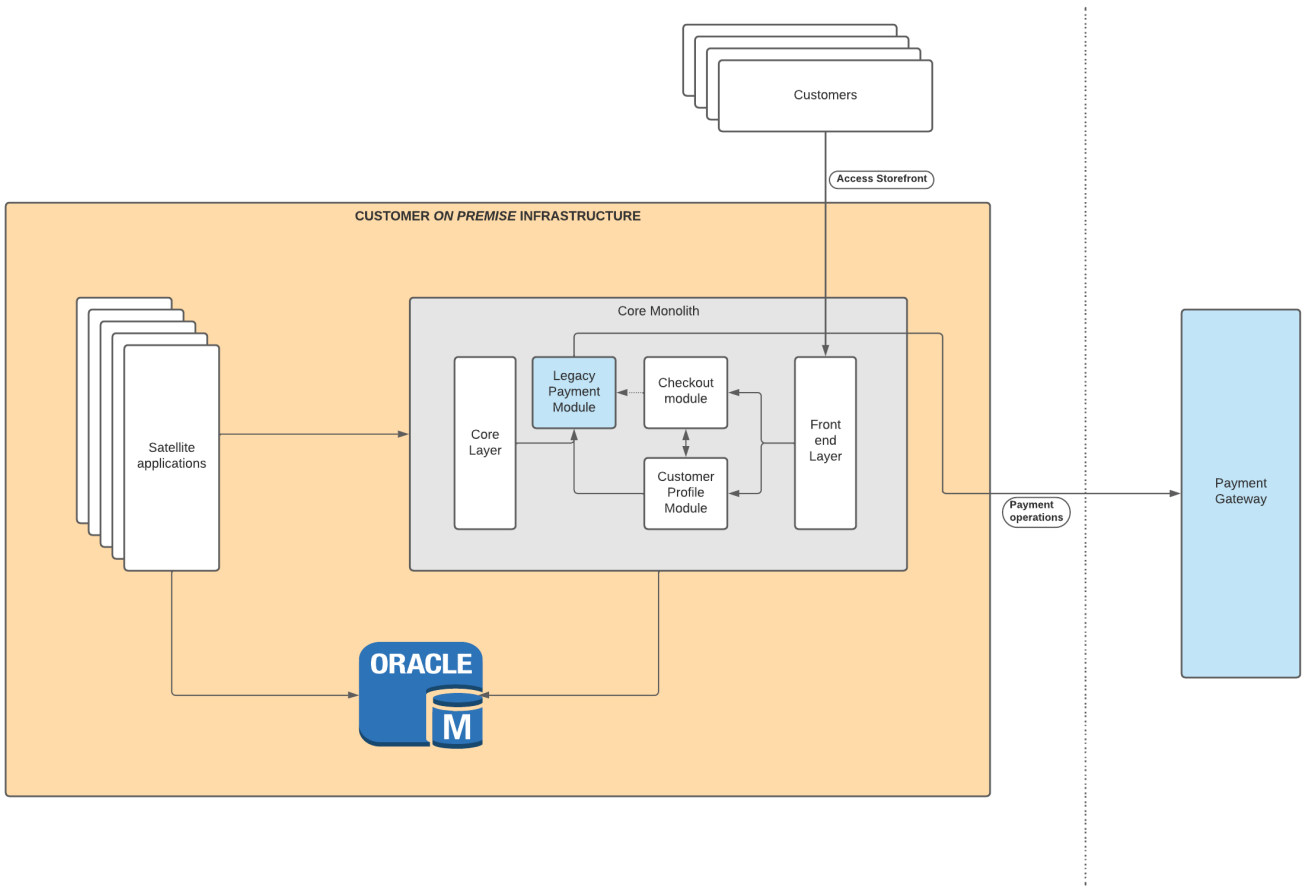
Experiência com aplicações monolíticas *

- (●) Sim
- ( ) Não

## Análise da arquitetura atual

A seção a seguir conterá informações sobre uma arquitetura monolítica e uma proposta de migração para uma arquitetura de micro serviços. Dadas essas informações, por favor responda as seguintes perguntas.

### Diagrama da arquitetura atual



## Descrição da arquitetura atual

O monólito principal é uma aplicação Java 7 EE, desenvolvida e mantida a mais de 15 anos pela empresa e tem como foco principal ser o motor das regras de negócios de todo o e-Commerce. Esse motor possui diversas integrações com sistemas externos e aplicações satélites. A aplicação ainda recebe incrementos de funcionalidades ainda hoje e está em processo de deprecação.

Possui uma API REST para diversas integrações que o sistema possui com outras aplicações. Aplicações que fornecem serviços como apresentação de dados, métricas de clientes e serviços de envio de e-mail são alguns exemplos. A aplicação possui um sistema de autenticação BASIC AUTH, o que significa que o cliente da aplicação deve passar no header da requisição o ID e a senha concatenados pelo caractere dois pontos (:) e codificados em base64.

A aplicação em análise possui mais de 15 anos de existência e sua base de código é bastante amarrada. Diversos times de desenvolvimento trabalharam nesta base, e muitas vezes simultaneamente. Como visto anteriormente, o processo de qualidade de código não era muito organizado, cada time de desenvolvimento tinha seus padrões e práticas de qualidade. Alguns, no entanto, não tinham essa cultura madura. Portanto existem diversos métodos ou classes que possuem muitas responsabilidades e dependências fortes entre si, tornando sua manutenção complexa e custosa.

A aplicação é hospedada em uma infraestrutura interna do cliente. Por conta da infraestrutura de servidores on premise, é necessário um processo de deploy customizado para a realidade e estrutura disponível. O processo atual de deployment da aplicação consiste na geração dos arquivos executáveis através de uma pipeline automática gerenciada pelo github. Após gerado os artefatos, é preciso realizar o deployment do mesmo nos servidores através de uma ferramenta legado.

**Dado a descrição acima e levando em consideração a sua experiência com aplicações desse tipo, responda as perguntas abaixo:**

Qual a sua opinião sobre a coesão e acoplamento da arquitetura acima? *

É bem visível os problemas que teremos em relação a coesão e acoplamento na arquitetura acima, teremos classes enormes com métodos gigantescos que fazem 1001 o que primeiro dificulta e muito a manutenibilidade das classes. A baixa coesão e alto acoplamento da arquitetura acima prejudicam e muito o projeto.

Descreva suas percepções em relação a manutenção dessa aplicação? Quais a dificuldades e possíveis facilidades que esse tipo de arquitetura provém em relação a manutenibilidade da base de código? *

Com certeza uma das maiores dificuldades em relação a manutenção desta aplicação será com inserção de novos código e testes, pois será muito mais difícil de diminuir o acoplamento existente sem ter que fazer inúmeras refatorações no código.

Não consigo pensar em facilidades ao usar uma arquitetura monolito neste caso.

Como você avalia a performance e escalabilidade dessa arquitetura? Quais os principais pontos positivos e negativos desse tópico nessa arquitetura? *

A performance e escalabilidade em uma arquitetura monolitos são muito prejudicados:

Performance é extremamente afetada neste caso pois todo o programa é carregado para a memória e mesmo componentes quase não usados estarão sempre carregados consumindo recursos da máquina.

Escalabilidade é quase inexistente neste tipo de arquitetura pois por exemplo, se o legacy payment module está sobrecarregado e precisa subir uma nova aplicação para suportar mais requests, é necessário escalar toda a aplicação sendo que só um pequeno pedaço dela que realmente precisaria ser escalada.

Em relação a disponibilidade da aplicação. Quais pontos são positivos ou negativos dessa arquitetura? *

Os problemas de disponibilidade ficam evidentes quando algum erro acontece dentro da aplicação monolito, ao invés de só um pedaço ficar offline (como por exemplo, a parte de pagamentos estar offline porém ainda ser possível usar o frontend para ver os produtos) a aplicação toda acaba por ficar offline e é necessário subir ela novamente para ter o serviço online novamente e isso pode acarretar em inúmeros transtornos para a empresa e seus clientes.

O que você acha desse tipo processo de deploy da aplicação analisada? *

A parte do github ter uma pipeline automática para gerar os executáveis parece ser bem ok, mas não sei opinar sobre a aplicação legado pois não sei os detalhes desse deploy.

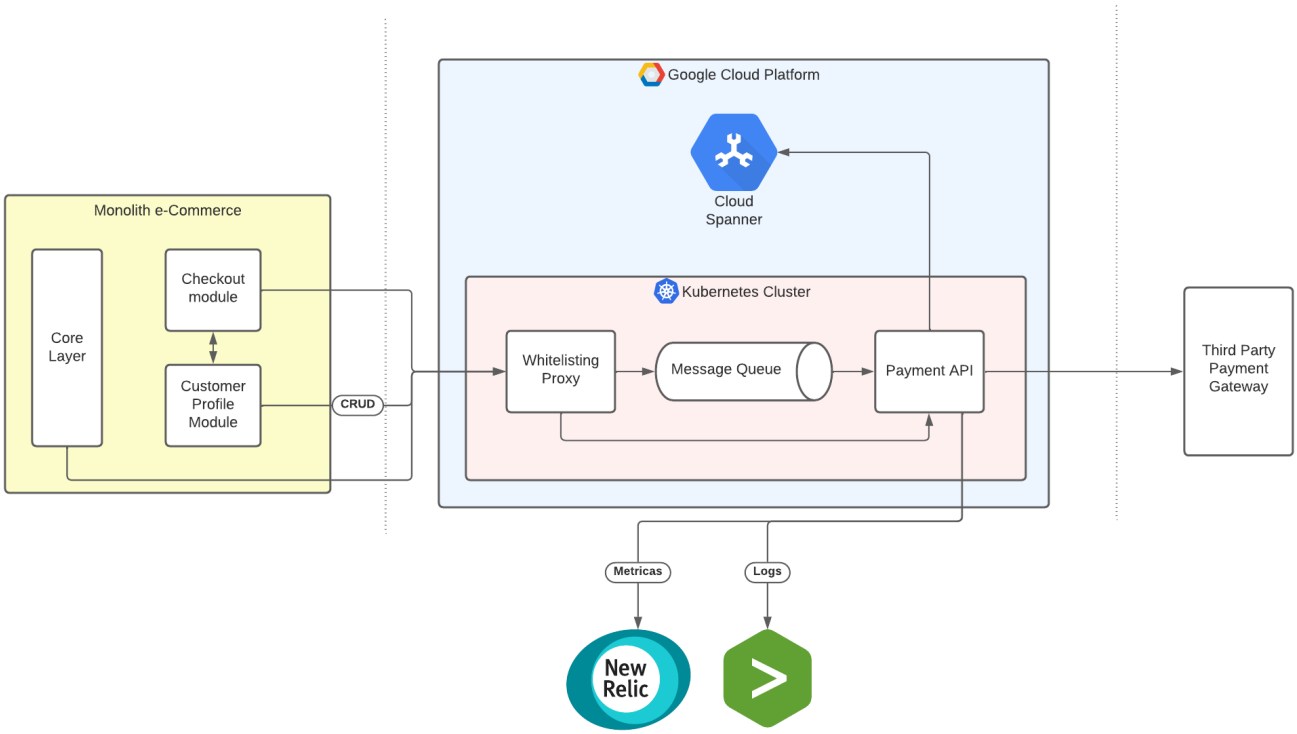
Algum comentário extra? *

ai é foda

## Análise da arquitetura proposta

Esta seção terá como enfoque uma arquitetura de micro serviços proposta para fazer a migração do monólito. Esse trabalho abrange apenas o módulo de pagamento, indicado no diagrama abaixo como Legacy Payment Module (Caixa cinza descrita dentro do componente do monólito). Então todos os detalhes abaixo são referentes a esse módulo de pagamento migrado.

## Diagrama da arquitetura proposta



## Descrição da arquitetura proposta

A arquitetura proposta possui grandes diferenças em tecnologias e  abordagens em relação à arquitetura monolítica atual.

A implementação da proposta de arquitetura irá utilizar Java 18 + Spring Boot/Cloud como framework de desenvolvimento web e micro serviços.

Em relação ao cluster de Kubernetes: Será criado um pod para o micro serviço, que ficará exposto ao mundo exterior para que seja consumido pelos serviços e aplicações através de um recurso de Ingress. O motor que irá executar e gerenciar o cluster desenvolvido será o GKE, motor de kubernetes da GCP. Será utilizado uma estrutura de listas de permissão através de um proxy. Apenas os serviços permitidos poderão fazer contato com o micro serviço desenvolvido. O proxy irá trabalhar como um verificador de permissão, onde irá permitir apenas o recebimento de requisições de aplicações confiáveis.

Para o balanceamento de carga será utilizado o recurso padrão do Kubernetes para balanceamento de carga da aplicação, o Horizontal Pod Autoscaler. O valor de quantidade de réplicas mínimas será de 1, o valor de quantidade de réplicas máximas será de 10 e a métrica analisada para disparar o escalonamento automático será de porcentagem de CPU utilizada acima de 50%.

As mensagens recebidas serão entregues ao serviço de mensageria KubeMQ através de um proxy. Esse proxy será responsável por permitir apenas conexões de hosts confiáveis.

Google Cloud Spanner foi o banco de dados escolhido devido a suas características SQL distribuído, através de data sharding.

O cluster estará integrado com uma solução de agregador de Logs, o Splunk e outra solução de coleta e análise métricas, o New Relic.

Por fim, o cluster terá um CI/CD gerenciado por aplicações como Harness/CircleCI/Google Build. O processo de deploy será automático nos clusters GKE, apenas necessitando o trigger da pipeline.

## Dado a descrição acima e levando em consideração a sua experiência com aplicações desse tipo, responda as perguntas abaixo:

Qual a sua opinião sobre a coesão e acoplamento da arquitetura acima? Você vê vantagens ou desvantagens em relação a arquitetura anterior? *

Existem vantagens na utilização desta arquitetura, com a criação do novo módulo de pagamento do zero é possível garantir que toda a aplicação seja coesa em suas classes e que o acoplamento entre elas seja mínimo.

Descreva suas percepções em relação a manutenção dessa aplicação? Você vê vantagens ou desvantagens em relação a arquitetura anterior? *

Em relação a manutenção eu só consigo ver vantagens, pois será muito mais fácil de modificar o código e expandi-lo (isso se o código for coeso e com baixo acoplamento) e será possível ter testes para toda a aplicação, facilitando na identificação e correção de bugs.

Como você avalia a performance e escalabilidade dessa arquitetura Quais os principais pontos positivos e negativos desse tópico nessa arquitetura? Você vê vantagens ou desvantagens em relação a arquitetura anterior? *

Os principais pontos positivos desta nova arquitetura na minha visão é a possibilidade de escalar somente o módulo de pagamento quando ele tiver muitos acessos, assim melhorando a performance da aplicação como um todo pois não será mais preciso escalar toda a parte legado (isso se o que for escalado seja o módulo de pagamento novo). lembrando que se a sobrecarga for em outros módulos legados não surtirá efeito em performance e escalabilidade.

Em relação a disponibilidade da aplicação. Quais pontos são positivos ou negativos dessa arquitetura? Você vê vantagens ou desvantagens em relação a arquitetura anterior? *

Em relação em disponibilidade é possível ver uma grande melhora neste quesito, pois será possível manter o sistema de pagamentos sempre ativo e funcionando, caso algum dos pods tenha problema (offline ou algum erro interno), as requisições podem ser enviadas a outro pod de pagamento ou escalar um novo pode e assim não irá prejudicar o cliente.

A vantagem dessa arquitetura sobre a monolita é clara, enquanto a aplicação toda fica offline quando algo acontece na parte de pagamento, com a nova arquitetura é possível manter ela rodando e funcionando sem problemas pois se algo acontecer com algum pod de pagamento, é só enviar as requests para outro pod que está funcionando (ou escalar outro pod).

O que você acha desse tipo processo de deploy da aplicação analisada? Você vê vantagens ou desvantagens em relação a arquitetura anterior? *

Este processo de Deploy da nova arquitetura é mais simples e fácil do que a da arquitetura monolito, e muito se deve pelo uso de Kubernets e pipeline automático usando Harness/CircleCI/Google Build e automaticamente fazendo o deploy no GCP apenas com 1 click, bem diferente da arquitetura anterior que era preciso usar uma aplicação legado para fazer o deploy nos servidores de deploy.

Algum comentário extra? *

Me obrigue

# Avaliação por especialistas de tecnologia - Arquitetura de micro serviços - TCC - Cassiano Jaeger Stradolini

Este formulário será utilizado como fonte de dados para a avaliação de uma arquitetura de micro serviços proposta por Cassiano Jaeger Stradolini como parte de seu projeto de conclusão de curso.

A primeira seção será focada em perguntas de perfil profissional, como maneira de entender as características e conhecimentos que o especialista possuí. A segunda será focada em perguntas relacionadas a arquitetura analisada atual e a proposta pelo trabalho.

Agradeço imensamente pelas respostas que você fornecer. Isso será muito importante para a conclusão e validação do projeto que foi desenvolvido.

OBS: Adicionarei todos que responderam aos agradecimentos <3 <3 <3

E-mail *

ambrosi.vinicius@gmail.com

## Análise de perfil profissional

Qual a sua idade? *

29

Quantos anos de experiência trabalhando com desenvolvimento? *

De 6 a 8 anos

Grau de escolaridade *

- ( ) Nível médio completo
- (•) Nível superior incompleto
- ( ) Nível superior completo
- ( ) Pós-graduação incompleta
- ( ) Pós-graduação completa
- ( ) Outro: _____

Cargo na empresa em que trabalha *

- ( ) Desenvolvedor - Junior/Pleno
- (•) Desenvolvedor - Sênior
- ( ) Lider tecnico
- ( ) Arquiteto
- ( ) Gerente de desenvolvimento
- ( ) Outro: _____

Experiência com micro serviços *

- (•) Sim
- ( ) Não

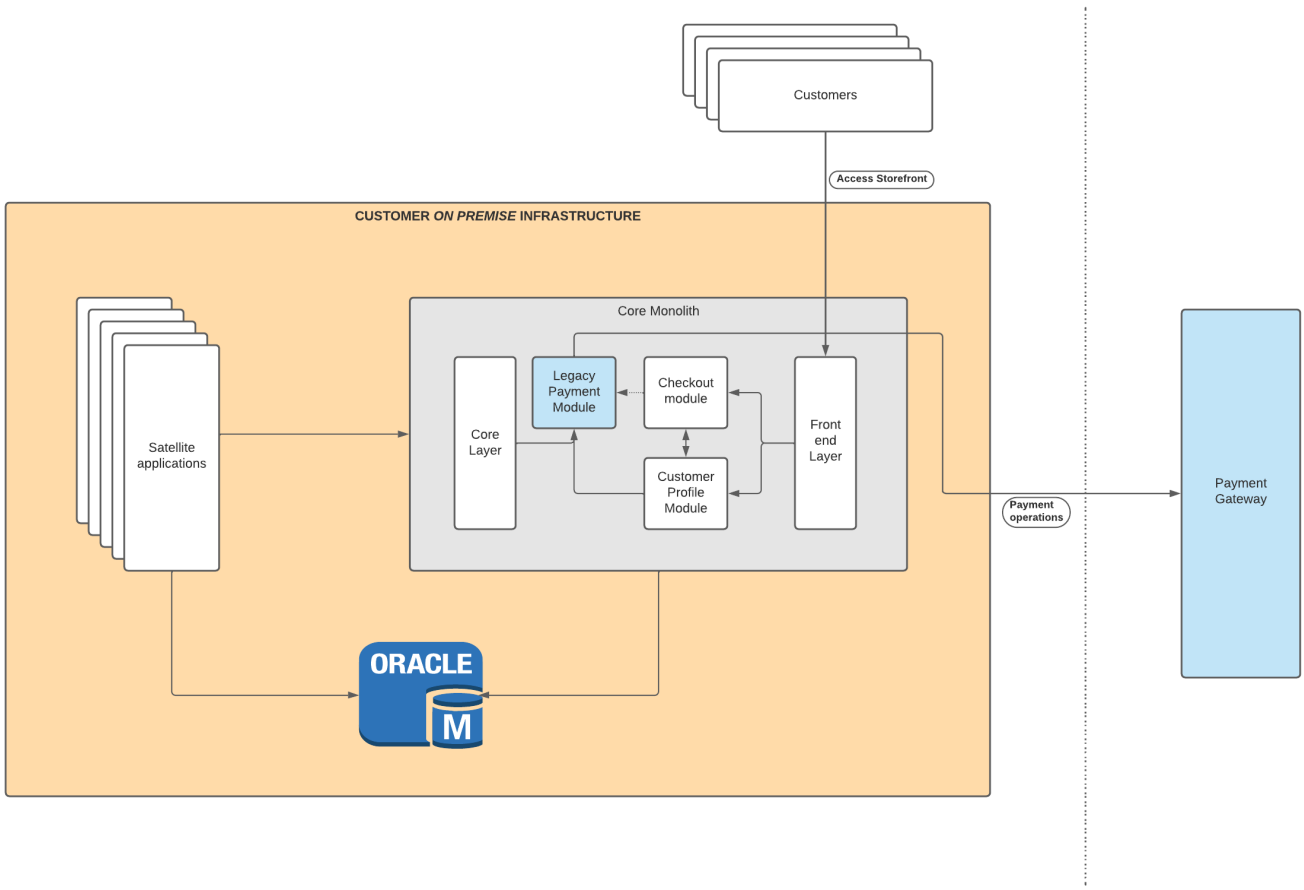
Experiência com aplicações monolíticas *

- (●) Sim
- ( ) Não

## Análise da arquitetura atual

A seção a seguir conterá informações sobre uma arquitetura monolítica e uma proposta de migração para uma arquitetura de micro serviços. Dadas essas informações, por favor responda as seguintes perguntas.

### Diagrama da arquitetura atual



## Descrição da arquitetura atual

O monólito principal é uma aplicação Java 7 EE, desenvolvida e mantida a mais de 15 anos pela empresa e tem como foco principal ser o motor das regras de negócios de todo o e-Commerce. Esse motor possui diversas integrações com sistemas externos e aplicações satélites. A aplicação ainda recebe incrementos de funcionalidades ainda hoje e está em processo de deprecação.

Possui uma API REST para diversas integrações que o sistema possui com outras aplicações. Aplicações que fornecem serviços como apresentação de dados, métricas de clientes e serviços de envio de e-mail são alguns exemplos. A aplicação possui um sistema de autenticação BASIC AUTH, o que significa que o cliente da aplicação deve passar no header da requisição o ID e a senha concatenados pelo caractere dois pontos (:) e codificados em base64.

A aplicação em análise possui mais de 15 anos de existência e sua base de código é bastante amarrada. Diversos times de desenvolvimento trabalharam nesta base, e muitas vezes simultaneamente. Como visto anteriormente, o processo de qualidade de código não era muito organizado, cada time de desenvolvimento tinha seus padrões e práticas de qualidade. Alguns, no entanto, não tinham essa cultura madura. Portanto existem diversos métodos ou classes que possuem muitas responsabilidades e dependências fortes entre si, tornando sua manutenção complexa e custosa.

A aplicação é hospedada em uma infraestrutura interna do cliente. Por conta da infraestrutura de servidores on premise, é necessário um processo de deploy customizado para a realidade e estrutura disponível. O processo atual de deployment da aplicação consiste na geração dos arquivos executáveis através de uma pipeline automática gerenciada pelo github. Após gerado os artefatos, é preciso realizar o deployment do mesmo nos servidores através de uma ferramenta legado.

## Dado a descrição acima e levando em consideração a sua experiência com aplicações desse tipo, responda as perguntas abaixo:

### Qual a sua opinião sobre a coesão e acoplamento da arquitetura acima? *

Considerando a frase "existem diversos métodos ou classes que possuem muitas responsabilidades e dependências fortes entre si, tornando sua manutenção complexa e custosa." fica explícito que a solução aparenta ser pouco coesa, e fortemente acoplada a nível de código, o que implica na arquitetura também. E se de fato verdade, demonstra que a solução estar sendo mantida até os dias de hoje é supreendente, e a execução simultânea de diferentes times em cima dessa base já deve ter tido como consequência inúmeros efeitos indesejados. Esse deve também ter servido como impedimento e/ou limitação inviábilizando a migração e evolução da arquitetura ao longos dos anos. Uma base amarrada torna evolução mais custosa e menos encorajadora.

O fato da arquitetura ser monolítica não é um problema, mas sim a falta de seu potencial para evolução. Como diz Robert C Martin em Clean Architecture:
“A good architecture will allow a system to be born as a monolith, deployed in a single file, but then to grow into a set of independently deployable units, and then all the way to independent services and/or micro-services.”

O princípio mais importante em arquiteturas de qualidade é a sua escalabilidade e capacidade de evolução.

Descreva suas percepções em relação a manutenção dessa aplicação? Quais a dificuldades e possíveis facilidades que esse tipo de arquitetura provém em relação a manutenibilidade da base de código? *

Soluções monolíticas tendem a ser menos custosas em termo de manutenção, porém tendem a ser mais sucetíveis a regressões generalizadas de impacto geral. Além de terem uma escalabilidade limitada.

Uma solução monolítica geralmente possui uma manutenção mais simples e direta. Não precisa lidar com as integrações entre micro serviços e suas complexidades agregadas. Requer menor senioridade da equipe em relação a práticas de devops (é muito mais fácil fazer o deploy de um monólito do que de inúmeros serviços). E o time não precisa ter conhecimento específico de cada componente.

Entretanto, as dificuldades surgem na escalabilidade e no tamanho da explosão na introdução de problemas. Em arquiteturas monolíticas, por mais que possam também escalar tanto horizontalmente como verticalmente, não é possível escalar por componente ou funcionalidade. Se um componente da solução consome muito recurso, toda a solução é afetada. Assim como na introdução de problemas fatais. Enquanto em soluções de micro serviços na introdução de um problema fatal um determinado componente pode ficar indisponível, na monolítica todos os componentes sofrem. Logo, tendo uma explosão muito maior que a isolada na sua contra parte.

Como você avalia a performance e escalabilidade dessa arquitetura? Quais os principais pontos positivos e negativos desse tópico nessa arquitetura? *

Acredito que o único ponto positivo para parte de performance e escalabilidade na arquitetura proposta é não ter que lidar com o throughput nas integrações entre múlitplos serviços. Em termos de escalabilidade, a arquitetura monolítica não tem como esse seu foco principal.

Acredito que os pontos que prejudicam a escalabilidade e performance são:
1. Dificuldade de escalar visando a arquitetura monolítica como explicado na questão anterior. Caso fosse uma aplicação com design coeso e pouco acoplado seria de fácil resolução, mas não é o caso aqui. Então a arquitetura atual tem um grande peso nesse ponto
2. Alta demanda em qualquer dos componentes do core monolith pode impactar a experiência do usuário, já que as soluções de backend e frontend são acopladas
3. O banco compartilhado entre as aplicações satélite e core monolith pode estar sobrecarregado por alguma atividade de algum dos lados, apresentando baixa performance para outras operações.

Em relação a disponibilidade da aplicação. Quais pontos são positivos ou negativos dessa arquitetura? *

Pontos negativos:
1. Acesso simultâneo ou até mesmo concorrente ao banco pelas aplicações satélites e core monolith põem em risco a consistência dos dados
2. O banco não possui estratégia de replicação para um banco de backup. Isso impossibilita  fazer o takeover da demanda caso a instância primária fique comprometida
3. Falta de load balancing tanto no acesso de clientes como das aplicações satélite ao core monolith, o qual possui um único nodo. Podendo causar indisponibilidade tanto para clientes como para as aplicações satélites em caso de falha ou em momentos de pico.

Além dos pontos já mencionados na pergunta acima, que podem também causar problemas de disponibilidade. Falta de escalabilidade leva a indisponibilidade.

O que você acha desse tipo processo de deploy da aplicação analisada? *

Poderia ser pior, pelo menos conta com uma pipeline para geração dos artefatos.

Mas considerando o estado da arte, soluções que visam facilidade e evolução, o processo de deploy inteiro deveria ser automatizado mesmo fornecendo métodos de controle.

A infraestrutura ser do cliente tende a ser um problema grande nesse tipo de solução, trazendo todos os tipos de impedimentos possíveis, que geralmente limitam as ferramentas que possuem acesso ao ambiente possibilitanto uma maior agilidade. O ideal, seriam soluções disponíveis da nuvem de verdade como Azure (a melhor plataforma cloud do mercado - indiscutível). Sendo no geral de público acesso quando devidamente autenticado.

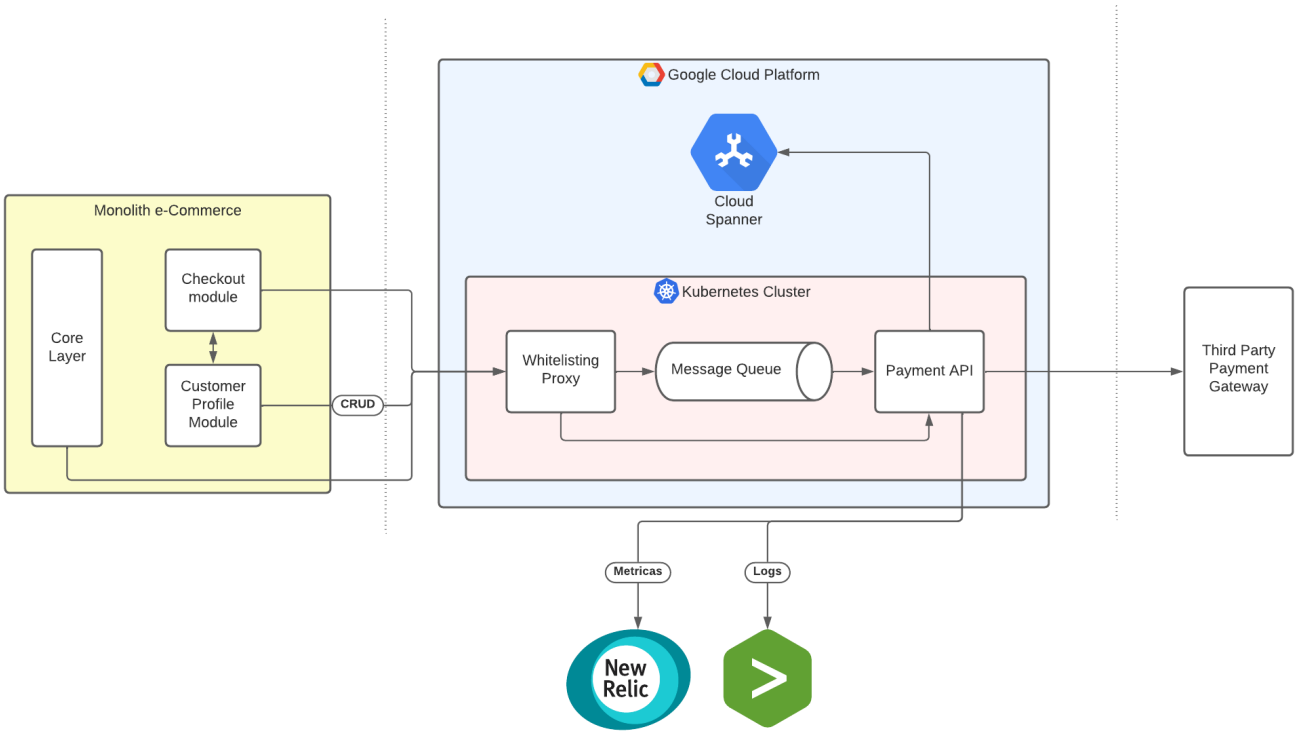
Algum comentário extra? *

raibrus

## Análise da arquitetura proposta

Esta seção terá como enfoque uma arquitetura de micro serviços proposta para fazer a migração do monólito. Esse trabalho abrange apenas o módulo de pagamento, indicado no diagrama abaixo como Legacy Payment Module (Caixa cinza descrita dentro do componente do monólito). Então todos os detalhes abaixo são referentes a esse módulo de pagamento migrado.

## Diagrama da arquitetura proposta



## Descrição da arquitetura proposta

A arquitetura proposta possui grandes diferenças em tecnologias e  abordagens em relação à arquitetura monolítica atual.

A implementação da proposta de arquitetura irá utilizar Java 18 + Spring Boot/Cloud como framework de desenvolvimento web e micro serviços.

Em relação ao cluster de Kubernetes: Será criado um pod para o micro serviço, que ficará exposto ao mundo exterior para que seja consumido pelos serviços e aplicações através de um recurso de Ingress. O motor que irá executar e gerenciar o cluster desenvolvido será o GKE, motor de kubernetes da GCP. Será utilizado uma estrutura de listas de permissão através de um proxy. Apenas os serviços permitidos poderão fazer contato com o micro serviço desenvolvido. O proxy irá trabalhar como um verificador de permissão, onde irá permitir apenas o recebimento de requisições de aplicações confiáveis.

Para o balanceamento de carga será utilizado o recurso padrão do Kubernetes para balanceamento de carga da aplicação, o Horizontal Pod Autoscaler. O valor de quantidade de réplicas mínimas será de 1, o valor de quantidade de réplicas máximas será de 10 e a métrica analisada para disparar o escalonamento automático será de porcentagem de CPU utilizada acima de 50%.

As mensagens recebidas serão entregues ao serviço de mensageria KubeMQ através de um proxy. Esse proxy será responsável por permitir apenas conexões de hosts confiáveis.

Google Cloud Spanner foi o banco de dados escolhido devido a suas características SQL distribuído, através de data sharding.

O cluster estará integrado com uma solução de agregador de Logs, o Splunk e outra solução de coleta e análise métricas, o New Relic.

Por fim, o cluster terá um CI/CD gerenciado por aplicações como Harness/CircleCI/Google Build. O processo de deploy será automático nos clusters GKE, apenas necessitando o trigger da pipeline.

## Dado a descrição acima e levando em consideração a sua experiência com aplicações desse tipo, responda as perguntas abaixo:

Qual a sua opinião sobre a coesão e acoplamento da arquitetura acima? Você vê vantagens ou desvantagens em relação a arquitetura anterior? *

A separação do componente de pagamento com certeza melhora a coesão, mas a melhoria em acomplamento é relativa.

Tem evolução no acoplamento se e somente se a implementação no Monolith e-Commerce traz abstração no serviço sendo consumido, e provem de um framework que agilize a troca para um outro serviço, caso necessário. Caso contrário, os problemas de acoplamento, por mais que reduzidos, persistem. Inlcusive problemas de acoplamento em arquiteturas de micro serviços são muito comuns quando os domínios não foram bem separados e/ou consomem recursos compartilhados.

O mesmo se aplica ao componente de pagamento de forma isolada, por mais que o impacto possa ser reduzido. Se o objetivo for vender o serviço para diferentes clientes é importante considerar como trabalhar nessa alimentação de necessidades diferentes.

As desvantagens que vejo são relacionado a maturidade do time. É uma migração grande, que vai de uma solução ultrapassada para uma utilizando de muitos recursos novos. Enquanto positivo, caso o time não tenha maturidade para implementar é possível enfrentar novamente problema de coesão e acoplamento, por mais que o escopo seja diferente. Separar caixas não resolve esses problemas obrigatoriamente.

Um questionamento final. Por que o Monolith e-Commerce se comunica com o componente de pagamento? Não seria mais interessante remover essa iteração, e ter a solução de frontend se comunicar diretamente com o componente? Isso poderia desacoplar mais a solução de backend.

Descreva suas percepções em relação a manutenção dessa aplicação? Você vê vantagens ou desvantagens em relação a arquitetura anterior? *

As melhorias de manutenção parecem claras para mim. Algumas delas são:

1. O mais óbvio em relação a arquitetura de micro serviços - o Monolith e-Commerce e o Componente de pagamento são isolados. Ou seja, o impacto de problemas introduzidos em um dos módulos tem menos probabilidade de afetar o outro programa. Um problema no componente de pagamento, afetaria somente pagamento no Monolith e-Commerce, o qual conseguiria ainda seguir com todas as outras operações

2. Utilização de tecnologias mais recentes (Java 18, Spring Boot, etc) tendem a ter funcionalidades que facilitam a implenetação

3. Ferramentas de monitoração mais maduras que possbilitam agilizar troubleshooting de qualquer problema

As desvantagems em relação a manutenção, vejo por uma perspectiva mais operacional. A manutenção do componente novo vai necessitar de conhecimento sobre:

1. Containerização - Containers, Kubernetes
2. Plataform Cloud - Google Cloud, GCP Proxy, GCP GKE, Configurar Autoscalling, Spanner
3. Message Queue - KubeMQ
4. Monitoramento - Splunk, New Relic
5. DevOps - CI/CD qualquer

Por mais que evolução seja desejado, a complexidade na manutenção cresce exponencialmente e necessita conhecimento de todas as ferramentas acima em profundidade para evitar problemas maiores. Além de que isso aumentara muito o custo da solução.

Como você avalia a performance e escalabilidade dessa arquitetura Quais os principais pontos positivos e negativos desse tópico nessa arquitetura? Você vê vantagens ou desvantagens em relação a arquitetura anterior? *

O foco da nova arquitetura parece ser o estado da arte em performance e escalabilidade, tendo como efeito colateral um aumento significativo no custo operacional da solução.

Um questionamento: a utilização do componente de pagamentos é a de maior uso da solução? O que justifica o overhead de arquitetura/design e o aumento de custo para esse componente? Existem outros componentes com mais uso do que o componente de pagamentos? Imagino que sim, já que para pagar é necessario passar perto pelo menos dos módulos de Profile ou Checkout.

Mas considerando os benefícios do modelo de arquitetura, acredito ter 3 camadas de benefício, sendo:

1. Horizontal Scaling do Componente de Pagamentos. O benefício aqui é óbvio, o componente não vai, geralmente, ser afogado com requests. Ter o orquestrador para auto escalar a solução é um óbvio beneficio de performance e escalabilidade, já que as chances de ter requests em espera é reduzida.

2. Bando de dados SQL com Sharding. A distribuição de dados é uma forma inteligente de remover bottlenecks tanto na escrita como leitura no banco de dados. Um único ponto aqui, é que não aparenta ter sido definida nenhuma estratégia de recovery caso o banco de dados morra. Esta implicito na estrategia de sharding? Caso não, é um problema grande de escalabilidade, podendo ser um single point of failure da solução.

3. Message Queue. A utilização de um MQ melhora ainda mais o benficio do ponto 1. Além ter a capacidade de escalar, o MQ tira muitas preocupações do Monolith e-Commerce de lidar com retries, enfileramento e assim por diante.

Em relação a disponibilidade da aplicação. Quais pontos são positivos ou negativos dessa arquitetura? Você vê vantagens ou desvantagens em relação a arquitetura anterior? *

Acredito ter mencionado alguns na resposta acima, mas no geral a arrquitetura parece estar se encaminhando para um alto nível de disponibilidade? Foi pensado algum SLA de disponibilidade? 2/3/4 noves? Seria interessante também ter um pouco de noção do dimensionamento e de utilização esperada para responder por completo, mas alguns pontos que notei foram:

O proxy (negando acesso indevido) somado com o message queue (para tratar de enfileiramento e recuperação) e auto scaling do orquestrador deve fornecer um bom índice de disponibilidade. Mas acredito ser interessante fazer o decoupling do Monolith e-Commerce do Componente de pagamento. Com o tráfego tendo que se comunicar com ele, pode ainda trazer impactos na integração entres o módulos, e o processo de deploy do monolith parece ter sido mantido no modelo anterior. Além desse ponto, a integração com o third party pode também trazer implicações de disponibilidade.

Mas no geral, como mencionado, acredito que traga uma melhoria grande nesse processo com a nova arquitetura.

O que você acha desse tipo processo de deploy da aplicação analisada? Você vê vantagens ou desvantagens em relação a arquitetura anterior? *

O processo parece ser o desejado - tudo automatizado. Esse tipo de arquitetura requer um nível alto de maturidade de devops, que caso não existente pode ser um grande impossibilitador.

Algum comentário extra? *

Not raibrus.
Devia ter usado azure, a melhor plataforma cloud do mercado.