# Microsserviços aplicados no gerenciamento de dados de vistorias imobiliárias: um estudo de caso

**Rodrigo H. Müller [1,2], Cristina Meinhardt[1,3], Odorico M. Mendizabal[1,3]**

[1]PPGCC - Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC)

[2]Rede Vistorias Franchising S/A

[3]PGComp - Centro de Ciências Computacionais
Universidade Federal do Rio Grande (FURG)

rodrigo.h.muller@posgrad.ufsc.br

{cristina.meinhardt, odorico.mendizabal}@ufsc.br

***Abstract.*** *As arquiteturas de microsserviço são cada vez mais adotadas na indústria e setor empresarial, pois oferecem alta escalabilidade, disponibilidade e velocidade de desenvolvimento. A literatura acadêmica propôs uma variedade de abordagens para automatizar o dimensionamento e alocação de recursos, recuperação de falhas, ajuste de parâmetros e muitas outras tarefas que permitem o auto-gerenciamento de microsserviços. A avaliação dessas abordagens em um cenário realista requer aplicações com comportamentos diversificados, demandando requisitos específicos quanto a escalabilidade, desempenho e confiabilidade. Este trabalho apresenta uma solução baseada em uma arquitetura de microsserviços, que consiste de onze serviços, fornecendo uma visão real para avaliação da aplicabilidade de novas abordagens para microsserviços e sistemas de software autônomos em geral. O problema alvo desta solução é o gerenciamento de dados de vistorias imobiliárias. Este trabalho apresenta uma visão geral da arquitetura adotada, indicando os desafios e aprendizados obtidos durante a implementação e manutenção deste sistema.*

## 1. Introdução

Computação em nuvem tem se demonstrado um modelo adequado para desenvolvimento e implantação de aplicações Web. Por um lado, plataformas de computação em nuvem oferecem suporte à configuração, implantação e migração de máquinas virtuais ou contêineres, permitindo um desenvolvimento rápido e personalizado de aplicações. Por outro, modelos de cobrança comuns nestes ambientes, como o pagamento sob demanda (como o *pay-as-you-go*) permitem que a alocação física e manutenção da infraestrutura sejam terceirizados. Características como elasticidade também favorecem um melhor provisionamento de recursos, permitindo que oscilações na carga e demanda de recursos seja ajustada em tempo de execução.

Diversas aplicações vêm sendo desenvolvidas e implantadas sobre plataformas de computação em nuvem. Empresas como Google, Amazon e Spotify oferecem aplicações e plataformas sofisticadas com soluções para desenvolvimento [Jonas et al. 2019], armazenamento [Bernstein et al. 2011] ou entretenimento [Adhikari et al. 2012] completamente disponibilizados em plataformas de computação em nuvem. Mais recentemente,

observa-se uma tendência em desenvolver aplicações para computação em nuvem sobre arquiteturas baseadas em microsserviços ao invés dos modelos monolíticos tradicionais [Haselböck et al. 2017, Gan and Delimitrou 2018]. Uma das primeiras empresas a adotar essa estratégia de microsserviços foi a Netflix, que começou com a migração de seu monolito em diferentes serviços ainda em 2008 [Bukoski et al. 2016]. Entre as vantagens para realizar essa migração, está o fato de que as implementações de cada equipe são desacopladas, permitindo que diferentes times definam uma melhor estratégia de implementação. Outro ponto favorável à adoção de microsserviços é a fácil escalabilidade de partes específicas de uma aplicação. Enquanto que, em um monolito, todo o código seria replicado para mais instâncias, utilizando diferentes serviços é possível escalar somente aqueles em que a elasticidade se faz necessária. Por outro lado, a implantação de tais arquiteturas implicam em um aumento na complexidade do sistema, já que os serviços devem comunicar-se entre si constantemente, requerendo uma infraestrutura estável, e padrões bem definidos de comunicação [Fowler 2017]. Além disso, por se tratar de um sistema amplamente distribuído, o sistema está sujeito a falhas que não aconteceriam em um monolito como, por exemplo, transações distribuídas envolvendo diversos serviços. Estratégias como o padrão SAGA [Garcia-Molina and Salem 1987, Limón et al. 2018] surgem como alternativas para conferir maior robustez a arquiteturas de microsserviços.

Este trabalho apresenta um caso real de implementação e manutenção de uma aplicação para uma empresa de vistorias imobiliárias, através da criação da vistoria por dispositivos móveis e geração do laudo da vistoria para ser entregue às partes envolvidas implantada na forma de microsserviços. Além disso, há a integração com uma série de parceiros, como CRM (*Customer Relationship Management*, ou Gestão de Relacionamento com o Cliente) imobiliários e grandes grupos imobiliários brasileiros. Na implementação desta solução são adotados os serviços de nuvem computacional providos pela *Amazon Web Services* (AWS), utilizando os serviços de armazenamento de dados Amazon *Simple Storage Service* (S3), instâncias de máquinas virtuais Amazon *Elastic Compute Cloud* (EC2) e banco de dados relacionais Amazon *Relational Database Service* (RDS) providos pela empresa, e utilizando uma solução de contêineres baseados em Docker. Além da descrição detalhada da arquitetura, são discutidas as tecnologias utilizadas e as principais decisões de projeto, visando atender tanto requisitos de escalabilidade, integração e extensibilidade. Aspectos relacionados ao ciclo de vida da aplicação, desenvolvimento, teste, implantação e manutenção de serviços também são apresentados. Espera-se, dessa forma, apresentar ao leitor um caso de uso de aplicação baseada em microsserviços desenvolvida e implantada integralmente em uma plataforma de computação em nuvem, destacando os desafios enfrentados e lições aprendidas.

O restante do trabalho está organizado como segue. A Seção 2 descreve as principais tecnologias utilizadas. A Seção 3 detalha a aplicação apresentada, assim como a arquitetura de microsserviços implementada, e os detalhes de configuração e manutenção. A Seção 4 discute as principais lições aprendidas e desafios na construção desta aplicação, enquanto a Seção 5 apresenta as considerações finais.

## 2. Referencial teórico e tecnologias adotadas

Esta seção apresenta os detalhes sobre a metodologia de desenvolvimento e tecnologias adotadas no desenvolvimento, implantação e execução da arquitetura proposta.

O processo de desenvolvimento de software é baseado em metodologias ágeis, sendo uma das características deste método o compartilhamento de código entre equipes e integração frequente de novas funcionalidades ao código base já existente. Um desafio, entretanto, é garantir que novas funcionalidades tenham sido testadas e que novas versões do software não comprometam funcionalidades do sistema já em operação. Portanto, o processo de desenvolvimento de software adotado pela equipe inclui a prática de *Integração Contínua*, onde novas funcionalidades ou novos serviços são integrados de forma automática ao processo de implantação [Karlesky et al. 2007, Polo et al. 2007]. Cada integração feita pelo desenvolvedor é verificada por uma ferramenta de implantação automatizada. Um teste completo é realizado e relatórios sobre o estado do *software* após o teste são produzidos. Assim, erros na integração são rapidamente detectados, reduzindo problemas em fases mais avançadas do ciclo de desenvolvimento.

Para permitir uma maior flexibilidade na configuração dos componentes do sistema, opta-se pela tecnologia de contêineres [Casalicchio 2017, Maenhaut et al. 2019]. Contêineres são compostos por imagens contendo bibliotecas, código e configurações necessárias para a execução de um serviço. A partir de uma imagem, outras aplicações, bibliotecas ou configurações adicionais podem ser aplicadas para a criação de novas imagens para os contêineres. Dessa forma, o processo de atualização e extensão de serviços em uma arquitetura complexa e distribuída pode ser facilitado pelo uso desta tecnologia. Além da portabilidade, contêineres também se apresentam como uma tecnologia de virtualização leve, especialmente quando comparados com máquinas virtuais tradicionais [Felter et al. 2015]. Na arquitetura proposta, contêineres são criados para cada microsserviço disponível para a aplicação. O uso de um gerenciador de contêineres e orquestrador de contêineres facilitam o desenvolvimento e manutenção durante o ciclo de vida da aplicação, além de padronizar o processo de integração de uma grande variedade de microsserviços.

A orquestração de contêineres permite ajustar a execução e uso de contêineres em tempo de execução. Uma plataforma para gerenciamento e orquestração permite relacionar os diversos contêineres disponibilizados, de forma a atender requisitos de disponibilidade e desempenho, mantendo níveis de acordo para qualidade dos serviços. Portanto, através da orquestração de contêineres, pode-se estabelecer interfaces de comunicação entre microsserviços, prover redundância e elasticidade, além da implantação de novas funcionalidades de forma contínua, sem interrupção do serviço [Khan 2017, Rufino et al. 2017, Casalicchio and Iannucci 2020]. Ferramentas de orquestração de contêineres também auxiliam na inicialização de réplicas de serviços de forma transparante, sendo possível manter alta disponibilidade e escalabilidade. O balanceamento de carga distribui requisições entre réplicas de um serviço, reduzindo as chances de que o serviço encontra-se saturado ou presencie períodos de lentidão por saturação no uso de seus serviços.

### 2.1. Jenkins

Jenkins[1] é uma ferramenta popular para *Integração Contínua*, usada para automatizar a implantação de *software* e testes. Jenkins é implementada em Java, permitindo a execução de *scripts* Ant e Maven, além de *shell scripts* para sistemas Linux, Windows e *Unix-like*. *Scripts* de implantação podem gerar relatórios de teste unitário, utilizando JUnit, por

[1] Jenkins: https://jenkins.io/

exemplo, além de relatórios personalizados. *Scripts* de implantação podem ser disparados por *commits*, via API ou ainda, a partir do estado de execução de outros *scripts*. Dessa forma, pode-se configurar o Jenkins para implantar um módulo apenas após outro(s) módulo(s) tenha(m) sido implantado(s) com sucesso.

### 2.2. Docker

Docker[2] é um projeto de código aberto para automatizar a implantação de aplicações Linux em contêineres portáveis. O Docker ampliou o conceito de contêineres Linux (LXC) [Rosen 2014] de forma a disponibilizar APIs (*Application Programming Interfaces*) em nível de núcleo e de aplicação. Também são permitidas a criação de espaços de nomes para isolar completamente a visão das aplicações do subsistema hospedeiro [Peinl et al. 2016]. Imagens de contêineres Docker são tipicamente pequenas, devido ao uso de sistema de unificação de arquivos em camadas (ex. AUFS). Apenas arquivos que não existam no hospedeiro são copiados efetivamente para a imagem do contêiner. Assim, com virtualização em nível de sistema e sistema de arquivos em camadas, contêineres podem ser criados ou distribuídos rapidamente. Para organizar a implantação de serviços e aplicações, arquivos *dockerfile* são *scripts* compostos de vários comandos e argumentos para automaticamente executar ações em uma imagem base, criando novas imagens [Liu and Zhao 2014].

### 2.3. Docker Swarm

Docker Swarm[3] é um arcabouço para orquestração integrado ao Docker. Uma das principais funcionalidades do Docker Swarm é permitir o ajuste da escalabilidade de forma automática. Dessa forma, o número de nodos pode ser acrescido para atender uma demanda de carga crescente. Novos nodos também são adicionados automaticamente na presença de falhas, de modo que o número de réplicas operacionais informado na configuração do sistema seja mantido. Orquestradores de contêineres, como o Docker Swarm, utilizam ferramentas de monitoramento para avaliar o estado dos serviços e, em caso de procedimentos de verificação identifiquem anomalias, contêineres são reiniciados.

### 2.4. Serviços de computação em nuvem: Amazon AWS

Existem três grandes possibilidades de serviços quando falamos de computação em nuvem: SaaS, IaaS e PaaS. No modelo SaaS o software está hospedado em um servidor e suas licenças são distribuídas entre vários clientes, ou seja, o software é vendido como um serviço que inclui a infraestrutura e licenças, pagando-se apenas pela sua utilização. Este trabalho esta mais relacionado ao modelo IaaS, onde toda infraestrutura de TI pode ser contratada como serviço, por exemplo, como a infraestrutura disponibilizada pela Amazon AWS. O último modelo, PaaS, fornece serviço personalizado com facilidade de configuração, disponibilizando recursos de desenvolvimento para que empresas possam desenvolver suas aplicações sem ter de começar do zero, por exemplo, os recursos disponibilizados pela Google App Engine.

Amazon AWS (*Amazon Web Services*) oferece uma grande variedade de serviços para computação em nuvem para o usuário final, por exemplo, recursos computacionais, como unidades de armazenamento e processamento, cujas capacidades podem

[2]Docker: `www.docker.com`

[3]Docker Swarm: `https://docs.docker.com/engine/swarm/`

ser ajustáveis e estas podem ser adquiridas de forma dinâmica, através de diferentes formas de cobrança pelo uso, de acordo com a necessidade dos clientes. As próximas subseções detalham os principais serviços disponibilizados pela Amazon AWS adotados na arquitetura proposta.

### 2.4.1. Amazon EC2

EC2 (*Elastic Compute Cloud*)[4] é um serviço Web que oferece instâncias de nodos com recursos computacionais com capacidades redimensionáveis. Através de interface Web, usuários podem lançar instâncias virtuais, sendo que cada instância pode ser configurada com imagens de contêineres adequadas para a sua função.

Os tipos de instâncias disponibilizados assumem formatos pré-definidos de recursos em função do número e frequência de processamento (expressados por ECU – *Elastic Computing Unit*), memória principal e armazenamento local. A cobrança de instâncias EC2 pode ser *sob-demanda*, *reservada* ou *spot instances*. Na primeira, o usuário paga uma taxa por hora de uso de acordo com as configurações das instâncias alocadas; na reserva de instâncias, o usuário paga um valor pela alocação das instâncias, além de uma taxa por uso (mais baixa que a cobrada pela modalidade de uso *sob demanda*); ao contrário das anteriores, a cobrança por *spot instances* é dinâmica e sempre alterna. O usuário paga uma taxa baixa por hora de uso, mas é associado um valor limite para o preço da instância. A instância será executada enquanto o custo da máquina for menor que o preço máximo aceito pelo usuário. Quando o custo de uso da máquina ficar maior que o preço acordado, o AWS encerrará a instância.

### 2.4.2. Amazon S3

Amazon S3 (*Simple Storage Service*)[5] é um serviço robusto de armazenamento capaz de armazenar e recuperar grandes quantidades de dados. Neste serviço, objetos são armazenados de forma redundante em múltiplos dispositivos com distanciamento físico em uma região Amazon S3, a fim de prover maior disponibilidade e durabilidade. Internamente, os dados são armazenados como objetos em partições. Um objeto é composto de um arquivo e, opcionalmente, metadados que descrevem o arquivo.

### 2.4.3. Amazon RDS

Amazon RDS (*Relational Database Service*)[6] é um serviço de banco de dados relacional oferecido pela Amazon AWS. O RDS facilita o processo de configuração de bancos de dados relacionais em ambientes de computação em nuvem, além de prover maior escalabilidade através do redimensionamento das instâncias executoras do serviço. O RDS oferece acesso e integração com bancos de dados MySQL, Oracle, SQL Server e PostgreSQL, facilitando o processo de desenvolvimento. Além disso, o Amazon RDS aplica

[4] Amazon EC2: `https://aws.amazon.com/ec2/`
[5] Amazon S3: `https://aws.amazon.com/s3`
[6] Amazon RDS: `https://aws.amazon.com/rds`

atualizações e realiza *backups* bases de dados, evitando que o cliente tenha que se preocupar com a manutenção do sistema de banco de dados. Assim como as instâncias EC2, instâncias de bancos de dados RDS também podem apresentar diferentes configurações referentes à capacidade de processamento e memória, com variações no preço de acordo com o tipo de instância. É comum associar serviços de banco de dados RDS ao serviço de armazenamento S3.

### 2.5. Redis

O Redis[7] é um serviço de código aberto (licenciado pela BSD) de armazenamento de estrutura de dados em memória. Ele é usado como banco de dados, cache e intermediário de mensagens. Uma das vantagens do Redis é o suporte a diferentes estruturas de dados como strings, hashes, listas, conjuntos, bitmaps, dentre outras, sendo possível realizar operações atômicas nestas estruturas.

O Redis destaca-se por um alto desempenho devido a sua estratégia de trabalhar com um conjunto de dados na memória. Dependendo do caso de uso, é possível implementar a persistência em disco periodicamente ou através da criação de *logs* de comandos.

## 3. Aplicação para gerenciamento de vistorias imobiliárias

O estudo de caso deste trabalho é uma empresa provedora de serviços para vistorias em ativos imobiliários. Embora não seja obrigatória pela lei, a vistoria tem se tornado um procedimento padrão, pois garante maior transparência entre as partes envolvidas, indicando o estado inicial do imóvel, sendo utilizada para posterior conferência ou em disputas jurídicas. Atualmente essa empresa atua no modelo de franquias, atuando em mais de 300 cidades. Um exemplo de vistoria realizada é a vistoria de entrada, que consiste em examinar o imóvel, seus ambientes e itens, verificando suas características e atestando estado de conservação através de fotos e vídeos do local coletados por um vistoriador credenciado com o auxílio de um *tablet*. Ao final de uma locação, a vistoria de saída vai comparar o estado do imóvel e seus itens ao relatório de vistoria de entrada, apontando as divergências encontradas. Depois de uma vistoria de saída podem ser identificados reparos a serem feitos pelo inquilino. A vistoria de conferência vai atestar se os reparos realizados atendem as necessidades, conforme os dados coletados na vistoria de entrada. O produto final da aplicação ao cliente é um relatório no formato PDF contendo todas as informações pertinentes sobre o imóvel.

Observando estas características, destacam-se como principais requisitos desta aplicação: armazenamento persistente de grandes volumes de dados, comunicação direta entre esse armazenamento e os *tablets*, através de uma API, *backups* automatizados, e a possibilidade de aumentar os recursos computacionais, caso haja necessidade.

### 3.1. Arquitetura

A arquitetura atual da aplicação é composta de alguns serviços e suas interfaces, conforme mostra a Figura 1. Os componentes da arquitetura são descritos a seguir.

- **Amazon S3:** o Amazon S3 funciona como servidor de dados para armazenamento das informações coletadas nas vistorias realizadas. Os dados são coletados por

[7]Redis: https://redis.io/

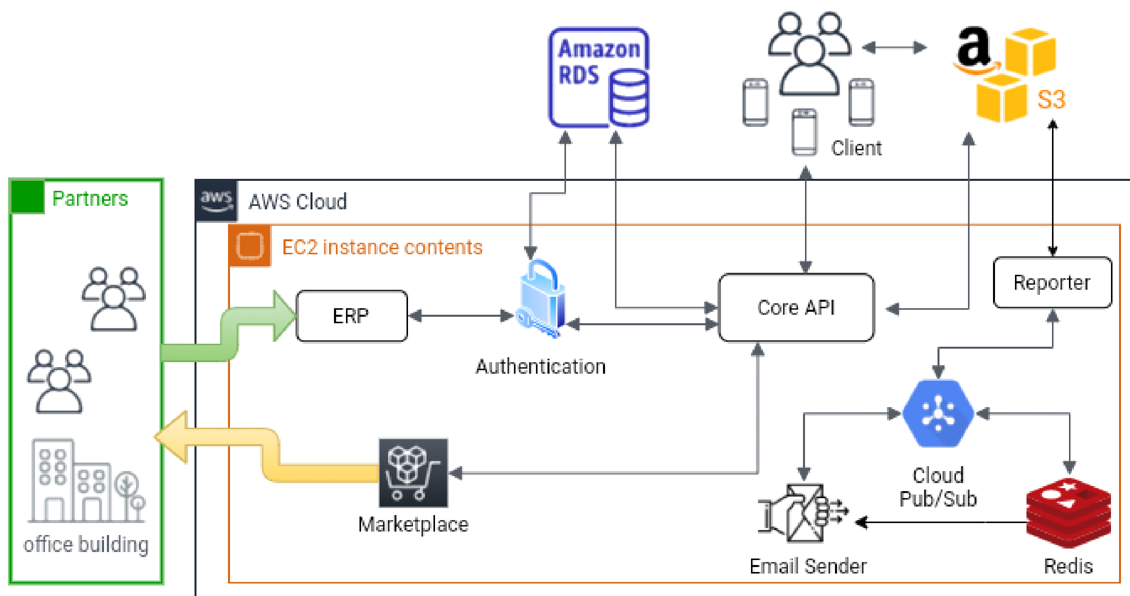


**Figure 1. Arquitetura da aplicação para gerenciamento de vistoria imobiliária.**

*tablets*, como as fotos dos imóveis. A escolha do Amazon S3 se dá pelo fato deste manter o histórico de alterações dos arquivos, realizar *backups* automáticos, e por prover uma API de acesso tanto para os dispositivos, diretamente, quando para outros serviços da arquitetura.

- **Client:** Este módulo descreve os usuários do sistema, através dos dispositivos móveis utilizados pelos vistoriadores; operadores das franquias, que recebem os pedidos e designam a um vistoriador; ou os clientes finais, imobiliárias e construtoras, que solicitam os pedidos através um painel próprio.
- **Amazon RDS:** adotou-se o Amazon RDS para a persistência de dados, utilizando o Sistema Gerenciador de Banco de Dados (SGBD) PostgreSQL. Esta solução abstrai a manutenção do SGBD, assim como instalação, configuração e realização de *backups*.
- ***Partners*:** designa-se como parceiros todas as empresas terceiras que integram com a API da aplicação. Um exemplo de parceiro são sistemas de CRM já utilizados pelas imobiliárias, agregando o pedido de vistoria no painel já utilizado por elas.

Além dos módulos descritos, também são apresentados serviços implementados como instâncias no EC2.

- **Core API:** Este serviço é responsável pela parte financeira (créditos que o usuário possui, reembolsos e consumo dos pacotes adquiridos), e pelo salvamento de dados dos pedidos de vistoria (local do imóvel, agendamento, e a qual franquia o pedido pertence).
- **Autenticação:** serviço utilizado para garantir a autenticação e autorização dos usuários do sistema: cliente final (imobiliária, por exemplo), operador da franquia, vistoriador, etc.
- ***Marketplace*:** serviço criado para permitir que os clientes possam acessar serviços de terceiros, tal qual assinatura eletrônica, sem que precisem fazer cadastros nessas plataformas, utilizando os créditos que eles já possuem com a empresa.

- **ERP:** permite que ERP, CRM e outros *softwares* possam usar os serviços disponibilizados de através de uma API simples. Optou-se por criar um serviço à parte do *Core API*, para adaptar a validação dos dados vindo de terceiros, bem como o armazenamento das informações pertinentes somente a esse módulos.
- **Reporter**: serviço responsável pela geração dos laudos entregues às partes envolvidas na transação imobiliária. Para isso, este módulo acessa o S3, de onde obtém os arquivos necessários, recebe os pedidos assincronamente, e retorna um evento para todos os inscritos.
- **Redis**: o Redis é utilizado como um serviço de filas. Embora a AWS possua um serviço para isso, optou-se por utilizar uma solução local, uma vez que esses dados são voláteis, não sendo necessário um armazenamento permanente.
- **pub/sub**: a comunicação entre os serviços pode se dar de forma síncrona, através de chamadas HTTP, ou de forma assíncrona, utilizando uma estratégia publicar/assinar (*pubsub*). Para isso, existe um serviço responsável por receber as mensagens de filas do Redis, e encaminhá-las aos serviços que se inscreveram em determinado evento.
- ***Email Sender***: serviço responsável por envios de emails transacionais da empresa, como o envio dos relatórios ao cliente, informações de que o pedido foi aceito ou rejeitado, entre outros.

A geração de relatórios de vistoria muitas vezes chega a centenas de páginas, sendo um processo demorado, pois além da geração do arquivo em si, é necessário realizar o *download* em grande número de arquivos armazenados em servidores do S3 da Amazon. As requisições a esse serviço chegam através de uma fila no Redis. Por se tratar de um serviço *stateless*, este é facilmente replicável, bastando aumentar o número de réplicas através do Docker Swarm.

A principal vantagem de utilizar um serviço de eventos é que as chamadas não são bloqueantes. Por exemplo, ao solicitar um relatório, é esperado que haja um tempo entre a requisição e o processamento dos dados para a geração do relatório de vistoria em formato PDF. Caso isso fosse feito de forma bloqueante, a conexão ficaria ocupada por muito tempo, o que limitaria a quantidade de requisições atendidas, prejudicando a experiência do usuário. Porém, ao implementar uma solução baseada em eventos, um evento é emitido quando um novo relatório é solicitado, e outro evento é gerado quando o processamento foi concluído, informando ao(s) solicitante(s) que a ação foi concluída, e estes ficam responsáveis por dar continuidade ao processo.

Outro ponto importante de arquiteturas baseadas em eventos é que, não raramente, muitos serviços vão depender de uma mesma mensagem. Por exemplo, após o relatório ser gerado, são emitidos eventos para que um email seja enviado, para que o *hash* do arquivo seja salvo no banco de dados para fins de cache, e também para que uma empresa parceira possa ser notificada sobre a finalização de determinado pedido, através do envio de um relatório para consulta. Delegar essas tarefas a serviços específicos, que comunicam-se de forma assíncrona e possuem estratégias para lidar com falhas, torna a arquitetura mais escalável e fácil de manter.

Devido à natureza da aplicação, que presta serviço majoritariamente para outras empresas, algumas integrações foram feitas com outros sistemas, que já são utilizados pelos clientes. Por se tratar de integrações com serviços de terceiros, mudanças abruptas

devem ser evitadas, já que demandariam esforço por parte da outra equipe, o que pode ser inviável. Por isso, optou-se em utilizar um serviço de integração desacoplado do serviço principal. Sua principal função é expor uma API pública, que pode ser acessada pelos parceiros, e fazer um mapeamento dessa API pública para a API privada. Esta integração é desenvolvida seguindo o padrão de projetos *proxy*. Isso permite uma maior flexibilidade interna, enquanto mantém uma API estável para consumo externo.

### 3.2. Implantação e manutenção

Atualmente, existem três ambientes para a aplicação: desenvolvimento, *sandbox* e produção. O ambiente de desenvolvimento é voltado para testes realizados pela equipe de desenvolvimento. Esse ambiente é bastante instável, já que as melhorias e correções de defeitos são testadas localmente. Além disso, neste ambiente há uma variedade de bibliotecas e configurações específicas ao ambiente de desenvolvimento. O ambiente *sandbox* se aproxima do ambiente de produção e é utilizado para revisar se o sistema está se comportando adequadamente através de testes de usabilidade e testes funcionais. Este ambiente também fica disponível às empresas parceiras, para que elas possam testar as integrações e também usufruir de um ambiente de testes. Neste ambiente, espera-se que os serviços encontrem-se alinhados com os que estão em produção ou então que sejam *release candidate*. Por fim, temos o ambiente de produção, que é acessado pelos clientes e parceiros.

Como ferramenta de versionamento de código, utiliza-se o Git e a plataforma de código-aberto Gitea[8], para a criação de *pull-requests* e *code reviews*. As versões dos serviços seguem o modo proposto pelo Semver[9], que indica a convenção Major.Minor.Patch para o versionamento de um artefato. As versões de *patch* são somente para correções de defeitos, enquanto a *minor* é utilizada quando refatorações internas são feitas, ou novas funcionalidades são desenvolvidas, mantendo a compatibilidade com código preexistente. Por fim, a mudança *major* indica que poderá haver quebra de compatibilidade com a versão anterior, e que códigos que dependam desse serviço possivelmente precisarão ser refatorados. Essa abordagem é utilizada tanto nas API públicas, quanto naquelas que são disponíveis aos parceiros.

O processo de implementação é feito através do Jenkins, que foi configurado para executar os testes unitários do projeto antes de gerar a imagem do contêiner. Caso algum dos testes falhe, o processo é interrompido, garantindo que as imagens do Docker só são geradas caso os testes passem, evitando que versões incorretas sejam lançadas. Além disso, ele faz o *upload* automático da imagem do Docker para o serviço de armazenamento de imagens, *Register*, para que estas imagens possam ser acessadas pelas máquinas que as necessitarem.

Embora existam outras opções bastante utilizadas para orquestração de contêineres, como o Kubernetes[10] e o Apache Mesos[11], a escolha do Docker Swarm se deu pelo perfil da equipe de desenvolvimento e requisitos da infraestrutura. Do ponto de vista da equipe, com o Docker Swarm é esperado um aprendizado mais rápido, pois este

[8]Gitea: `https://gitea.io/en-us/`
[9]Semver: `https://semver.org/`
[10]Kubernetes: `https://kubernetes.io/pt/`
[11]Apache Mesos: `http://mesos.apache.org/`

orquestrador oferece recursos mais simples para gerenciamento, quando comparado ao Kubernetes e Apache Mesos. Este fator é importante no contexto desta aplicação, pois a equipe de trabalho é enxuta. Do ponto de vista a aplicação e infraestrutura, Docker Swarm tem se mostrado adequado para *clusters* pequenos e implantação e uso em ambientes de produção pequenos [Hoque et al. 2017, Mercl and Pavlik 2019].

O monitoramento dos *logs* é feito através da ferramenta Kibana, que fornece painéis de controle que permitem observar a carga de consultas e entender a maneira como as solicitações fluem pelos aplicativos. Com o Kibana integrado no sistema é possível visualizar, de forma agregada, os *logs* gerados por todas as aplicações rodando nos contêineres. Além dos *logs*, a ferramenta também permite criar páginas de visualização personalizáveis. Uma destas páginas de visualização que é bastante utilizada agrupa os dados de quantos pedidos foram feitos por um período de tempo customizável, como por exemplo doze ou quatro horas, bem como a métrica de quantos relatórios foram solicitados pelos clientes, e quantos destes já haviam sido gerados anteriormente, e foram pegos do cache, evitando reprocessamento. Além disso, exibe gráficos das respostas HTTP da URL que é responsável pela criação de pedidos. Com isso, é possível verificar rapidamente se houveram casos atípicos, como erros fatais, que retornam o status HTTP 500, ou erros de conexão, como falha na conexão, dando *timeout* na requisição, ou outros retornos que não o esperado - resposta 200 do HTTP.

Além disso, a empresa utiliza o *Rocket.Chat*[12] como aplicativo padrão para comunicação entre os funcionários, sendo que mensagens de alerta são configuradas para envio automático através de requisições HTTP autenticadas. A maioria dos serviços também implementa um *logger* que registra os erros fatais para um canal específico, em que só os desenvolvedores participam. Com isso, as mensagens de erros são visualizadas rapidamente por membros da equipe, podendo ser direcionadas ao responsável. Isso é importante principalmente para os serviços que consomem os eventos do *pubsub*, já que o processamento ocorre após o retorno ao cliente, o que poderia causar a impressão de que tudo ocorreu corretamente. Em caso de erros, estes são corrigidos e reprocessados manualmente.

## 4. Lições Aprendidas

A adoção de uma arquitetura dividida em serviços nos permitiu flexibilidade para adotar diferentes tecnologias, seja na linguagem de programação, bibliotecas ou até mesmo com bancos de dados distintos. Esta liberdade de escolha de tecnologia pode ser puramente técnica, como por exemplo, a linguagem de programação possuir elementos que justifiquem sua escolha, ou então por questões de mercado. Um exemplo disso seria a dificuldade de encontrar programadores experientes em uma determinada linguagem, enquanto em outra, a mão de obra disponível seria maior.

Além disso, como os módulos estão isolados em contêineres, mover determinado serviço para outras máquinas é bastante simples, além de permitir aplicar escalabilidade horizontal, ou adquirir instâncias com maior processamento de CPU ou mais memória, se for desejável. Um exemplo de serviço que poderia se beneficiar desta estratégia seria o de geração de relatórios, já que a renderização do PDF é uma operação de uso intensivo de CPU.

[12]Rocket Chat: `https://rocket.chat/`

Todavia, essa mesma flexibilidade trouxe consigo alguns desafios, que foram sanados parcialmente, e que merecem atenção constante da equipe. Por exemplo, como uma requisição potencialmente passa por diversos serviços, é necessário ter um *log* agregado, que identifique a qual requisição aquele *log* pertence. Isso pode ser obtido gerando um identificador externo único, que é colocado em todos os *logs* [Richardson 2019, Las-Casas et al. 2019]. A adoção de um registro de *log* agregado também permite que ferramentas específicas de análise possam verificar o caminho percorrido por requisições, auxiliando na caracterização da carga de trabalho experienciada pela aplicação. Além disso, podem ser extraídas informações como a latência dos serviços, e outras informações relevantes para detectar falhas em determinas transações e gargalos de desempenho.

O uso intenso de mensagens entre os serviços também requer alguns cuidados. O primeiro deles é que os serviços estão em constante evolução. Muitas vezes os parâmetros esperados para executar determinada função são alterados, e a contraparte não é avisada dessa alteração, causando erros de integração. Algumas estratégias podem ser utilizadas para mitigar isso. A primeira delas é ter um validador no elemento que envia as mensagens. Por exemplo, utilizando o Apache Kafka, é possível definir um contrato que deve ser implementado pelos produtores das mensagens. Assim, antes de chegar aos consumidores, é feita uma validação se a mensagem está aderente ao contrato. Se não estiver, a mensagem não é enviada, evitando erros do lado do consumidor [Confluent 2020].

Outra necessidade ao lidar com serviços assíncronos é manter um registro das mensagens que não foram processadas. Esse comportamento pode ocorrer por diversos motivos, como o fato de um serviço de terceiros estar inoperante durante a requisição. Neste caso, pode ocorrer da mensagem processada posteriormente, ou jamais será processada. Atualmente, essas mensagens ficam registradas no *Rocket.Chat* e seu contexto fica salvo. Porém, por não ser um canal de comunicação, caso a quantidade de mensagens transitadas em um curto período for muito elevado, é possível que algumas se percam. Para evitar isso, pode-se salvar essas mensagens em um local mais confiável, como em outra fila do *Redis*, ou em um banco de dados [Bribesois 2013]. Assim, essas mensagens são persistidas e podem ser analisadas pela equipe.

Algumas soluções proprietárias para *logging* e controle avançado sobre serviços de filas são oferecidas por provedores de infra-estrutura à nuvem computacional. . Porém, estas implicam em custo adicional, além de uma maior dependência ao provedor. Alternativamente, há diversas soluções de código aberto gratuitas, que podem ser instaladas e configuradas manualmente, dispondo de funções similares a desses produtos pagos, como o OpenTelemetry[13] e o Apache Kafka[14]. O contraponto à adoção destas soluções é a demanda de maior envolvimento de membros da equipe com configuração e manutenção de serviços de infraestrutura, o que pode ser um impeditivo para equipes menores.

Hoje, a arquitetura, conforme descrita na Seção 3.1, é mantida internamente, mas com boa parte da infraestrutura sendo provida por serviços da AWS. Terceirizar soluções de armazenamento e nodos virtuais à terceiros se mostra adequado para a aplicação apresentada, visto que é possível habilitar *backups* automáticos e histórico de alterações de forma facilitada. Caso algum arquivo seja alterado no Amazon S3, a versão anterior não

[13]OpenTelemetry: `https://opentelemetry.io/`
[14]Apache Kafka: `https://kafka.apache.org/`

é removida, mas sim mantida em um histórico, podendo ser acessada pelo painel de controle da AWS ou por chamadas de API. Porém, os serviços que não armazenam estado, ou armazenamento volátil, são todos criados utilizando Docker, evitando dependência de um provedor, sendo possível migrá-los para outro provedor, caso necessário, seja por motivos tecnológicos ou financeiros. Além disso, sempre que possível, utiliza-se ferramentas com código aberto, que possam ser instaladas na infra-estrutura atual, evitando custos com licenciamento e suporte. Deve-se analisar, portanto, qual o limite aceitável para a empresa estar atrelada a opções proprietárias, qual o custo de migração para outro provedor, e o impacto que essa decisão causará na equipe, adotando-se a estratégia que melhor se adequar à organização.

## 5. Considerações finais

Este artigo apresenta um estudo de caso de uma aplicação para vistoria imobiliária desenvolvida sobre uma arquitetura de microsserviços. A aplicação demonstra a necessidade de integração com aplicações de terceiros, além de um fluxo de trabalho complexo com a necessidade de integração entre diferentes tecnologias e dispositivos.

Além de introduzir um estudo de caso prático da indústria, este trabalho apresenta, em detalhes, a arquitetura de software proposta para implementação da solução, descrevendo as tecnologias utilizadas e decisões de projeto. Serviços de computação em nuvem, especialmente na modalidade IaaS são largamente utilizados para o desenvolvimento e implantação da aplicação.

Finalmente, o artigo apresenta as principais lições aprendidas e desafios enfrentados no desenvolvimento e manutenção da aplicação. O relato deste estudo de caso proporciona uma visão prática e realista do uso de tecnologias de computação em nuvem e arquiteturas de microsserviços. Dessa forma, serve como um importante guia para o desenvolvimento de aplicações seguinte estes modelos atuais e com uma crescente tendência no desenvolvimento de serviços para a Internet.

## References


Adhikari, V. K., Guo, Y., Hao, F., Varvello, M., Hilt, V., Steiner, M., and Zhang, Z.-L. (2012). Unreeling netflix: Understanding and improving multi-cdn movie delivery. In *2012 Proceedings IEEE INFOCOM*, pages 1620–1628. IEEE.

Bernstein, P. A., Cseri, I., Dani, N., Ellis, N., Kalhan, A., Kakivaya, G., Lomet, D. B., Manne, R., Novik, L., and Talius, T. (2011). Adapting microsoft sql server for cloud computing. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1255–1263. IEEE.

Bribesois, A. (2013). Posion queues are a must! `https://alexandrebrisebois.wordpress.com/2013/08/14/poison-queues-are-a-must/`.

Bukoski, E., Moyles, B., and McGarr, M. (2016). How we build code at netflix. `https://netflixtechblog.com/how-we-build-code-at-netflix-c5d9bd727f15`.

Casalicchio, E. (2017). Autonomic orchestration of containers: Problem definition and research challenges. *ValueTools 2016 - 10th EAI International Conference on Performance Evaluation Methodologies and Tools*, pages 287–290.

Casalicchio, E. and Iannucci, S. (2020). The state-of-the-art in container technologies: Application, orchestration and security. *Concurrency Computation*, (March 2018):1–17.

Confluent (2020). Schema registry tutorial. https://docs.confluent.io/current/schema-registry/schema_registry_tutorial.html.

Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. (2015). An updated performance comparison of virtual machines and Linux containers. *ISPASS 2015 - IEEE International Symposium on Performance Analysis of Systems and Software*, 25482:171–172.

Fowler, S. J. (2017). *Microsserviços prontos para a produção*. novatec, São Paulo, 1 edition.

Gan, Y. and Delimitrou, C. (2018). The architectural implications of cloud microservices. *IEEE Computer Architecture Letters*, 17(2):155–158.

Garcia-Molina, H. and Salem, K. (1987). Sagas. *ACM Sigmod Record*, 16(3):249–259.

Haselböck, S., Weinreich, R., and Buchgeher, G. (2017). Decision guidance models for microservices: service discovery and fault tolerance. In *Proceedings of the Fifth European Conference on the Engineering of Computer-Based Systems*, pages 1–10.

Hoque, S., de Brito, M. S., Willner, A., Keil, O., and Magedanz, T. (2017). Towards container orchestration in fog computing infrastructures. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 294–299. IEEE.

Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., et al. (2019). Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*.

Karlesky, M., Williams, G., Bereza, W., and Fletcher, M. (2007). Mocking the embedded world: Test-driven development, continuous integration, and design patterns. In *Proc. Emb. Systems Conf, CA, USA*, pages 1518–1532.

Khan, A. (2017). Key Characteristics of a Container Orchestration Platform to Enable a Modern Application. *IEEE Cloud Computing*, 4(5):42–48.

Las-Casas, P., Papakerashvili, G., Anand, V., and Mace, J. (2019). Sifter: Scalable sampling for distributed traces, without feature engineering. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 312–324.

Limón, X., Guerra-Hernández, A., Sánchez-García, A. J., and Arriaga, J. C. P. (2018). Sagamas: a software framework for distributed transactions in the microservice architecture. In *2018 6th International Conference in Software Engineering Research and Innovation (CONISOFT)*, pages 50–58. IEEE.

Liu, D. and Zhao, L. (2014). The research and implementation of cloud computing platform based on docker. *2014 11th International Computer Conference on Wavelet Active Media Technology and Information Processing, ICCWAMTIP 2014*, pages 475–478.

Maenhaut, P. J., Volckaert, B., Ongenae, V., and De Turck, F. (2019). *Resource Management in a Containerized Cloud: Status and Challenges*, volume 28. Springer US.

Mercl, L. and Pavlik, J. (2019). The comparison of container orchestrators. In *Third International Congress on Information and Communication Technology*, pages 677–685. Springer.

Peinl, R., Holzschuher, F., and Pfitzer, F. (2016). Docker cluster management for the cloud-survey results and own solution. *Journal of Grid Computing*, 14(2):265–282.

Polo, M., Tendero, S., and Piattini, M. (2007). Integrating techniques and tools for testing automation. *Software Testing, Verification and Reliability*, 17(1):3–39.

Richardson, C. (2019). Pattern: Distributed tracing. `https://microservices.io/patterns/observability/distributed-tracing.html`.

Rosen, R. (2014). Linux containers and the future cloud. *Linux J*, 240(4):86–95.

Rufino, J., Alam, M., Ferreira, J., Rehman, A., and Tsang, K. F. (2017). Orchestration of containerized microservices for IIoT using Docker. *Proceedings of the IEEE International Conference on Industrial Technology*, pages 1532–1536.