FTT JOURNAL
of Engineering and Business

Aplicação de microsserviços para a concepção de ERPs: uma abordagem prática de padrões e antipadrões

Application of microsservice for the design of ERPs: a practical approach to standards and anti-patterns

Gabriel Lara Baptista (pro15969@cefsa.edu.br)
Mestre em Engenharia de Produção pela Universidade Nove de Julho (Uninove) e professor da Faculdade de Tecnologia Termomecanica (FTT).

Erick Tiossi (ericktiossi@gmail.com)
Graduado em Engenharia de Computação pela Faculdade de Tecnologia Termomecanica (FTT).

Laion Biazão (laionfb@yahoo.com.br)
Graduado em Engenharia de Computação pela Faculdade de Tecnologia Termomecanica (FTT).

Rafael Cardoso (rafael_eng31@outlook.com)
Graduado em Engenharia de Computação pela Faculdade de Tecnologia Termomecanica (FTT).

Rafael Galani (galani.rafael@gmail.com)
Graduado em Engenharia de Computação pela Faculdade deTecnologia Termomecanica (FTT).

Engenharia de Computação

FTT Journal of Engineering and Business.
• SÃO BERNARDO DO CAMPO, SP DEZ. 2020
• ISSN 2525-8729
Submissão: 24 ago 2020. Aceitação: 5 nov. 2020
Sistema de avaliação: às cegas dupla (double blind review).
FACULDADE DE TECNOLOGIA TERMOMECANICA, p. 79-92.

Resumo

O presente artigo tem como objetivo realizar a implementação de um sistema ERP utilizando a arquitetura de microsserviços e seus padrões de desenvolvimento, realizando a análise qualitativa em termos de desenvolvimento e performance do sistema em questão e observando quais são os padrões adotados e antipadrões existentes para a concepção desse modelo arquitetural. Levantaram-se os principais requisitos funcionais e não funcionais de um sistema ERP por meio de pesquisas bibliográficas em livros e artigos científicos da área de Sistemas de Informação. Além disso, também se efetuou a pesquisa dos históricos de aplicações já consolidadas no mercado, com o objetivo de detectar possíveis oportunidades de melhorias e, por fim, realizou-se a implantação do sistema em si num ambiente de nuvem; além disso, foram feitos testes computacionais para avaliar a sua performance. Por fim, foi alcançada uma aplicação baseada na arquitetura de microsserviços, escalável e com os melhores padrões no que diz respeito à arquitetura de microsserviços.

Palavras-chave: Microsserviços. ERP. Padrões de arquitetura de software. Computação em nuvem.

Abstract

This article aims to implement an ERP system using the microservices architecture and its respective development patterns and analyze the improvement of the system, in both development and performance matters, observing which are the patterns and anti-patterns adopted to this architectural model's conception. The main functional and non-functional requirements of an ERP system were retrieved by bibliographical research in books and scientific articles related to the Information Systems area and conversations with specialists and key-users. Also, research of previous applications already consolidated in the market were conducted, in order to detect potential improvement opportunities and, finally, the system was deployed in a cloud environment with computational tests being performed, aiming the evaluation of the achieved performance. Finally, the result was an application based in the microservices architecture, scalable and within the best practices related to the microservices architecture.

Keywords: Microservices. ERP. Patterns. Software architecture. Cloud computing.

Introdução

Os Sistemas Integrados de Gestão Empresarial (*ERP – Enterprise Resource Planning*) são sistemas transacionais que reúnem informações de todas as áreas de atuação da empresa, proporcionando sua integração (KARMAKAR, 1989; ZIPKIN, 2000; AROZO, 2003). É sabido que, a cada novo desenvolvimento de funcionalidades para o ERP, fica cada vez mais complexo alterar o que já foi desenvolvido e a manutenção do sistema torna-se cada vez mais complexa.

Maier e Rechtin (2000) definem projeto de Arquitetura de Sistemas como um processo criativo que propõe uma organização do sistema em questão, para que sejam atendidos tanto os requisitos funcionais como os não funcionais de um sistema. A Arquitetura de Sistemas é de suma importância no ecossistema de qualquer aplicação por estar diretamente relacionada ao seu desempenho, robustez, confiabilidade, capacidade de distribuição e manutenibilidade.

Com o passar do tempo, a Internet surgiu e trouxe uma nova realidade a seus usuários, deixando de ser uma rede que disponibiliza somente um simples acesso e obtenção de informações, tornando-se uma rede que nos envolve de diferentes formas ao longo do uso, geralmente de forma paralela (BERNARDO, 2013).

Com o aumento exponencial de usuários espalhados mundialmente (KEMP, 2019), surgem plataformas de escala global como YouTube, Facebook, Twitter, Instagram, e plataformas de entretenimento que comportam acessos diários de mais de 1 bilhão de usuários (YOUTUBE, 2019; FACEBOOK, 2019). Tal disponibilidade e capacidade de entrega jamais haviam sido demandadas pela indústria (FACEBOOK, 2019). Logo, tornaram-se necessárias novas soluções e abordagens, tanto na infraestrutura da plataforma em si quanto no seu processo de desenvolvimento, para que a qualidade e a homogeneidade possam ser garantidas no acesso do usuário (BORKAR; CAREY; LI, 2016).

De modo a atender tal demanda, a concepção arquitetural dessas plataformas sofreu um processo de reformulação. Segundo Tole (2013), nos últimos anos esse processo ocorreu drasticamente e diversas novas soluções arquiteturais foram propostas. Lewis e Fowler (2014) justificam tal reformulação em razão do cenário atual da indústria, com desafios cada vez maiores, bem como projetos e regras de negócio cada vez mais complexas.

Uma das arquiteturas com esse foco é baseada em microsserviços. Esta arquitetura pode ser definida como um estilo e um conjunto de técnicas e tecnologias utilizadas para que a aplicação em questão seja composta, em seu resultado, por um conjunto de serviços de baixo acoplamento entre si, autônomos, independentes e implantáveis de modo autônomo (JAMSHIDI *et al.*, 2018).

O presente trabalho visa propor, à luz das melhores práticas que estão sendo adotadas atualmente, a adoção da arquitetura de microsserviços para o desenvolvimento de ERPs, de modo que a escalabilidade, a manutenibilidade e o desempenho sejam priorizados a fim de manter a integridade de seus princípios básicos.

Após a introdução aqui apresentada, este artigo está estruturado com o objetivo de deixar claro em seu referencial os aspectos teóricos que motivaram a estrutura da arquitetura desenhada, bem como a decisão pelo uso de microsserviços. Em seguida, descreve o processo de decisão dos módulos que foram construídos e a lógica de negócio envolvida nestes módulos. A metodologia adotada ainda apresenta toda a tecnologia que foi utilizada para que fosse viável a entrega da prova, levando em consideração as vantagens da utilização de uma arquitetura em microsserviços. Os resultados apresentam os repositórios de código criados, o mapa arquitetural desenhado, as aplicações concebidas, o desempenho computacional medido durante os testes de carga e os custos

obtidos com a solução adotada. Oferecem, ainda, uma lista de lições aprendidas ao longo do desenvolvimento da pesquisa e a tabela de padrões com os antipadrões evitados. As considerações finais são apresentadas indicando que o objetivo proposto no parágrafo acima foi atingido, as limitações de pesquisa foram encontradas e deu-se destaque a algumas sugestões para estudos futuros.

Referencial teórico

Quando se fala em ERP, Sommerville (2016) propõe quatro módulos principais essenciais para considerar um sistema como um ERP: (1) compras; (2) cadeia de suprimentos; (3) logística e (4) relacionamento com o cliente.

Além disso, um bom ERP deve atender a requisitos não funcionais. Diversos autores descrevem qualidades esperadas de um sistema computacional, sendo que muitos deles baseiam-se na norma ISO 25030 (ISO, 2019) que regula tais aspectos, podendo-se citar os seguintes parâmetros de qualidade de um sistema: (a) segurança e proteção; (b) desempenho; (c) usabilidade; (d) flexibilidade; (e) interoperabilidade.

Outra característica interessante de um ERP é a Modularização de Sistemas. Segundo Gershenson, Prasad e Zhang (2003), um produto modular é aquele que é constituído por blocos correlatos de conhecimento, aglutinados, porém com uma divisão bem definida. Define-se por Modularização de Sistemas uma atividade projetada para dividir programas grandes em pedaços gerenciáveis e fornecer um mecanismo de biblioteca (LINDSEY; BOOM, 1978).

Jamshidi *et al.* (2019) afirma que, comparada a uma aplicação construída de maneira única e módulo único, uma aplicação modularizada permite que tal mudança seja feita de maneira mais sutil e menos onerosa, tornando mais simples o processo de mudança como um todo – desde o levantamento de partes e setores afetados até sua própria implementação, apresentando benefícios como manutenibilidade, capacidade de extensão individual, alta disponibilidade e outras características cada vez mais necessárias nos negócios atuais. De acordo com Lewis e Fowler (2014), com a utilização da arquitetura de microsserviços, uma solução pode ser escalada horizontalmente e verticalmente de forma simples; o ganho em produtividade e velocidade concedido aos desenvolvedores aumenta exponencialmente e tecnologias obsoletas podem ser facilmente substituídas sem causar efeitos colaterais ao restante da solução.

Entende-se serviço como algo que supre uma necessidade pública, portanto, à luz dessa definição, o sistema de serviços deveria resolver as necessidades de uma ou mais aplicações a clientes através da interação entre os serviços (CARNEIRO JR., 2016).

Apesar de não haver uma definição absoluta do termo, Lewis e Fowler (2014) definem arquitetura de microsserviços como uma maneira específica de projetar aplicativos de software como conjuntos de serviços implementados de forma independente. Uma arquitetura de microsserviços possui algumas vantagens que justificam a sua adoção, dependendo da análise do negócio, ou seja, se a aplicação necessita ser altamente escalável, com módulos do negócio bem definidos e independentes; ou ainda quando necessitar que seus *deploys* sejam feitos separadamente. Quando se tem ainda como premissa a utilização de diferentes linguagens de programação – para que o melhor de cada uma delas possa ser aproveitado no negócio – e uma alta necessidade de resiliência da aplicação, pode-se dizer que estes são indícios claros de que a aplicação deverá possuir uma arquitetura de microsserviços para atender todos esses requisitos.

Adotar um padrão que seja previamente testado e provado dentro do desenvolvimento de software faz com que o processo possa ser acelerado drasticamente (MARTIN, 2018). Além disso, um bom desenho de arquitetura deve ser considerado para possíveis cenários que não são claramente visíveis atualmente, mas que têm chance considerável de acontecer no futuro. Por outro lado, Hall (2017) categoriza o antipadrão como um padrão utilizado no processo de desenvolvimento do software que é pouco ou nada efetivo e acaba atrapalhando o processo.

Metodologia

Levantaram-se os principais requisitos funcionais e não funcionais de um sistema ERP por meio de pesquisas bibliográficas em livros e artigos científicos, além de possíveis gargalos e características de baixa performance presentes nos sistemas ERPs.

A identificação dos padrões e antipadrões em arquiteturas de microsserviços também foi feita através de pesquisas bibliográficas. Entretanto, para mostrar a viabilidade de cada um deles, provas de conceito foram conduzidas a fim de atestar a sua efetividade. Todas essas provas partiram do princípio de que a infraestrutura foi provisionada através de um provedor de computação em nuvem – Microsoft Azure -, abstraindo-se assim toda a necessidade do provisionamento da infraestrutura no presente trabalho.

Com o resultado da efetividade das provas de conceito, foram desenvolvidos os módulos de compras, da cadeia de suprimentos, de logística e de relacionamento com o cliente, módulos básicos dos ERPs, sendo que os comportamentos básicos de cada módulo foram respeitados.

Esses comportamentos trazem o cenário em que tudo começa quando uma venda é efetuada, o que consequentemente acaba gerando uma demanda de entrega na área de logística e, inevitavelmente, o produto tem a necessidade de ser diminuído em estoque, conforme sua venda é efetuada. Com isso, já é possível identificar a necessidade de compra, a fim de repor o produto novamente no estoque, operação que deve ser realizada pelo departamento de compras.

A implantação foi composta por um aplicativo móvel como exemplo, desenvolvido especificamente para a plataforma Android, e por um sistema web de *front-end*. Porém, o foco principal do estudo foi o desenvolvimento de um *backend* totalmente implementado utilizando as melhores práticas de uma arquitetura de microsserviços e as melhores tecnologias presentes na data de conclusão deste trabalho.

Ao longo do trabalho, foram levados em consideração o princípio e a vantagem da arquitetura de microsserviços em que existe a possiblidade de uso de diferentes linguagens de programação, frameworks e bancos de dados na aplicação. Para tanto, foram utilizadas as tecnologias ASP.Net Core, Node.js e Python, tendo como bases de dados o Redis e o Microsoft SQL Server. Além disso, os módulos foram desenvolvidos respeitando os padrões mencionados na literatura.

Vale ressaltar que, pelo escopo proposto, optou-se pela não implementação de um sistema de autenticação, tendo-se ciência de que esta operação seria fundamental para um ERP real, ficando salientado que este tópico fica reservado para um estudo futuro.

Lewis e Fowler (2011) também citam que um microsserviço pronto para ser colocado em um ambiente de produção necessita de uma documentação. A partir dessa observação, cada serviço que foi desenvolvido conta com uma documentação de seus respectivos métodos, acessível por meio da navegação até o caminho padrão de cada API. Nessas documentações, estão descritos os parâmetros e resultados obtidos em suas chamadas. As documentações foram geradas utilizando o

Swagger, um framework que facilita tanto no desenvolvimento quanto na geração de documentação para as APIs desenvolvidas.

O desenvolvimento de cada módulo do sistema foi realizado utilizando-se a plataforma Docker, onde é disposto um conjunto de produtos que visam habilitar a capacidade de realizar virtualizações em nível de sistema, empacotando códigos-fonte em estruturas chamadas de *containers*. Dentro de um sistema com diferentes módulos, cada *container* é isolado do outro, e cada um tem seus recursos e configurações individualmente parametrizados.

Após o desenvolvimento de todos os módulos, para integrar sua execução e definir um ambiente em comum para eles, optou-se por utilizar a ferramenta "Docker Compose", também presente na plataforma Docker. Essa ferramenta disponibiliza**,** por meio de interface de linha de comando, funcionalidades como construção de imagens, gestão dos *containers* (escala, interrupção, execução), podendo ser utilizada em diferentes ambientes: produção, testes, homologação e desenvolvimento.

Obtido o sucesso na execução do conjunto dos módulos dentro de ambientes locais, isto é, máquinas físicas usadas no desenvolvimento desses módulos, o próximo passo consiste em alavancar esse conjunto para um ambiente de execução controlada com maior monitoramento de recursos, performance e disponibilidade. Neste trabalho, optou-se pelo uso do Kubernetes por compatibilidade com a plataforma Docker, por ser uma plataforma de código aberto e pela alta relevância dentro da indústria e na comunidade de desenvolvimento. Pode-se definir o Kubernetes como uma plataforma de código aberto que fornece um sistema de orquestração de *containers*, com o objetivo de automatizar a implantação, a escala e operações realizadas nesses elementos, além de facilitar a integração com serviços de segurança, armazenamento e medição de resultados.

Além disso, Lewis e Fowler (2014) mencionam que eliminar todos os pontos de falhas e a identificação de cenários de falhas e catástrofes não assegura que os microsserviços sejam tolerantes a falhas e que estejam preparados para qualquer tipo de catástrofe. Para que um microsserviço seja atestado como tolerante a falhas ele precisa ser capaz de se recuperar com transparência, sem afetar a sua própria disponibilidade, a dos serviços vizinhos e a da solução como um todo.

Considerando a abordagem de Carneiro (2016), testes automatizados são fundamentais para o desenvolvimento de uma arquitetura de microsserviços. Para assegurar que essa solução está tolerante a falhas, foi imprescindível que o presente trabalho possuísse testes de unidade para cada microsserviço, teste de integração e testes de carga. Optou-se por realizar os testes de carga, já que a adoção por testes automatizados se dá pelo fato de que testes manuais não são capazes de testar todas as possibilidades de falhas de um projeto de microsserviços.

Para gerar os testes de carga no servidor de aplicação foram utilizados o cURL, Apache Benchmark e o JMeter. Todos eles têm por objetivo facilitar o processo de chamada de requisições simultâneas no ambiente a ser testado (APACHE, 2020; CURL, 2020; JMETER, 2020).

Além das ferramentas utilizadas para a realização do teste em si, foi utilizada uma ferramenta de monitoramento sob perspectiva de métricas, o Azure Insights, que auxilia na coleta de dados pela perspectiva do servidor. Este recurso pode ser definido como uma *feature* do Azure Monitor – é um serviço de gerenciamento de aplicativos feito para desenvolvedores profissionais de DevOps. Ele possibilita a visualização em tempo real dos aplicativos que estão em execução no servidor (BULLWINKLE *et al.*, 2020).

Análise de resultados

Considerando que o conceito de microsserviços sugere a independência no desenvolvimento entre cada serviço, os projetos foram conduzidos separadamente, tendo cada um dos sistemas um repositório distinto, conforme Tabela 1.

Tabela 1 – Repositório dos Serviços.

Serviço	Repositório
Estoque	https://github.com/rafael-bcardoso/ERP
Logística	https://github.com/rafael-bcardoso/ERP
Compras	https://github.com/LaionFB/ERP-PurchaseModule
CRM	https://github.com/rafaelgalani/ftt-microservices-erp-crm
FrontEnd	https://github.com/LaionFB/ERP-Frontend
Configurações do Kubernetes	https://github.com/LaionFB/ERP-Microservices

Fonte: Biazao et al (2020).

O mapa arquitetural proposto, apresentado na Figura 1, contém todas as boas práticas que foram desenhadas com objetivo de alinhar os times que desenvolveram cada microsserviço.

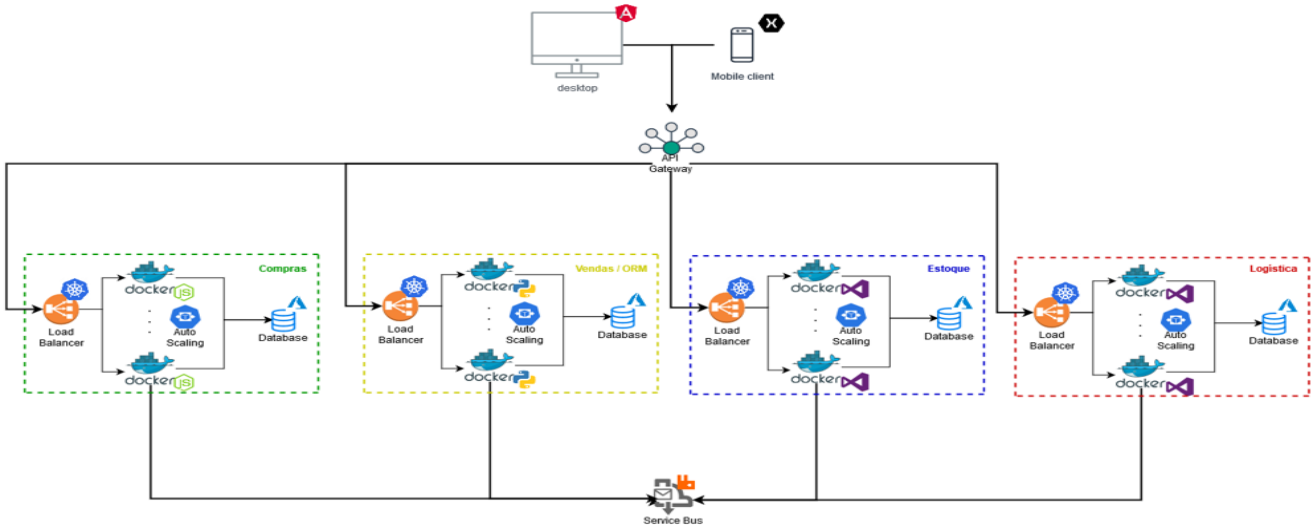

Figura 1 – Mapa Arquitetural.
Fonte: Biazao et al. (2020).

O mapa acima demonstra todo o fluxo de dados do sistema, que é composto por duas interfaces de usuário (*web* e *mobile*). As interfaces de usuário interagem com a aplicação em si primariamente através de um *API Gateway*, que foi feito utilizando-se a ferramenta *Ingress Controller* do

Kubernetes. Tal ferramenta recebe todas as requisições HTTP em um cluster *Kubernetes* e posteriormente realiza o encaminhamento para o serviço responsável por seu respectivo tratamento. O *Ingress Controller* provisiona, nativamente, o serviço de *Load Balance* das requisições, enviando-as para os serviços que estão com mais recursos disponíveis.

Cada serviço é composto por 1 a N *containers* Docker; cada *container* contém uma instância da aplicação que caracteriza o referido serviço. É válido salientar que o número de *containers* varia de acordo com a carga de dados sobre o serviço.

Isso significa dizer que, se o serviço estiver sobrecarregado, cabe ao *Kubernetes* criar *containers*, conforme parametrizado em seu arquivo de configuração (YAML), para que seja dividida a carga entre os *containers* através da ferramenta *Horizontal Pod Autoscaler*.

Pode-se definir um *Pod* como um grupo de um ou mais *containers*, com armazenamento e rede compartilhada e uma especificação de como executar os *containers* (KUBERNETES, 2020).

Vale lembrar ainda que o *container* não recebe a requisição diretamente do *API Gateway*: este a encaminha para o *Load Balancer* do serviço que, por sua vez, é responsável por encaminhá-la para a instância com menos carga num determinado momento. Cada um dos *containers* do mesmo serviço acessa a mesma instância de banco de dados, disponibilizada no ambiente da Microsoft Azure, ou seja, fora do ambiente Kubernetes. Os serviços não se comunicam diretamente entre si através de requisições HTTP; para isso, utilizam de um serviço de mensageria. Neste trabalho, optou-se por utilizar o *RabbitMQ*, onde são feitas as publicações e inscrições de eventos.

É importante ressaltar que, devido ao fato de a aplicação do projeto estar sendo construída em *containers*, isso permite que ela tenha a característica de independência de ambiente de execução. Logo, caso o ambiente escolhido tenha a disponibilidade de um serviço de Kubernetes e esse, por sua vez, permita a hospedagem de uma aplicação construída em *containers*, o processo de *deploy* da aplicação será suportado dentro desse ambiente. Portanto, outras plataformas como AWS, Google Cloud Platform, DigitalOcean – que também têm disponível o serviço de Kubernetes – poderiam ser usadas.

Um dos objetivos deste trabalho era verificar os ganhos em escalabilidade que um sistema construído em uma arquitetura de microsserviços consegue alcançar. Para avaliar esse aumento na escalabilidade da aplicação, foi desenvolvido um cenário de estresse, no qual a aplicação foi submetida a testes de carga com o intuito de avaliar os ganhos e perdas de desempenho quando uma determinada bateria de testes é executada. Este cenário de testes tem como principal objetivo avaliar a capacidade de escalonamento automático e o custo que esse procedimento gerou na arquitetura desenhada.

Para a realização dos testes, os microsserviços foram enviados para o *Azure Kubernetes Service* (AKS), para que pudesse ser realizada a avaliação do desempenho da solução quando submetidos a uma carga grande de requisições. O Azure Kubernetes Service possui a seguinte configuração: 2 máquinas virtuais Azure Standard B2s (nodes); 8 GB de memória RAM (total); 4 cores de processamento (total); 4 discos de armazenamento (total); Linux como sistema operacional. Estes recursos foram divididos entre os serviços conforme apresentado na Tabela 2:

Tabela 2 – Recursos dos Serviços

Serviço	Processamento	Memória RAM	Instâncias
Vendas – API Gateway Vendas – Ordens de Venda Vendas – Cotações	0,05 a 0,1 core para cada serviço	100 a 200 MB para cada serviço	1 a 8
Logística Estoque Compras	0,15 a 0,3 core para cada serviço	300 a 600 MB para cada serviço	1 a 8
Frontend – Angular	0,05 a 0,1 core	100 a 200 MB	1 a 4
Redis – Database	0,15 a 0,3 core	300 a 600 MB	1
RabbitMQ	0,25 a 0,5 core	512 MB a 1 GB	1

.

Fonte: Elaboração dos autores (2020).

No que diz respeito ao processamento e memória RAM, é reservado um valor mínimo de recurso para cada serviço, além de um valor máximo cujo serviço não poderá ultrapassar. Já a quantidade de instâncias é variada de acordo com a carga no serviço. Cria-se uma réplica do serviço quando essas instâncias atingem, em média, 75% de uso de algum dos recursos máximos. Por fim, o serviço de vendas foi dividido em três subsserviços, sendo reservado um terço dos recursos remanescentes para cada um deles.

Foram realizados 4 testes de carga no servidor de aplicação, simulando cenários comumente realizados em um ERP convencional: (a) 1 cliente enviando 1000 requisições para o servidor; (b) 2 clientes enviando 2000 requisições para o servidor; (c) 3 clientes enviando 3000 requisições para o servidor; e (d) 4 clientes enviando 4000 requisições para o servidor.

Os microsserviços utilizados nos testes foram os de logística, compras, CRM e estoque. Para tanto, foram utilizadas as seguintes funcionalidades da aplicação: (a) microsserviços de logística – listar todas as entregas que estão pendentes; (b) microsserviços de CRM – criar ordens de pedido; (c) microsserviços de compras – listar os pedidos de compras; e (d) microsserviços de estoque – listar todos os produtos em estoque.

Todos os testes foram realizados em um ambiente controlado e após sua execução o ambiente foi totalmente resetado, para que os próximos testes não fossem ficassem comprometidos. Realizaram-se os testes de carga com o envio de um total de 50.000 requisições para o servidor, de acordo com o objetivo de analisar o escalonamento automático da aplicação. Com o aumento da necessidade de recursos e conforme parametrização, o *Kubernetes*, por meio do *Horizontal Pod Autoscaler*, escalona os *Pods* de forma automática, já selecionando o Nó com mais recursos disponíveis para a criação do novo *Pod*.

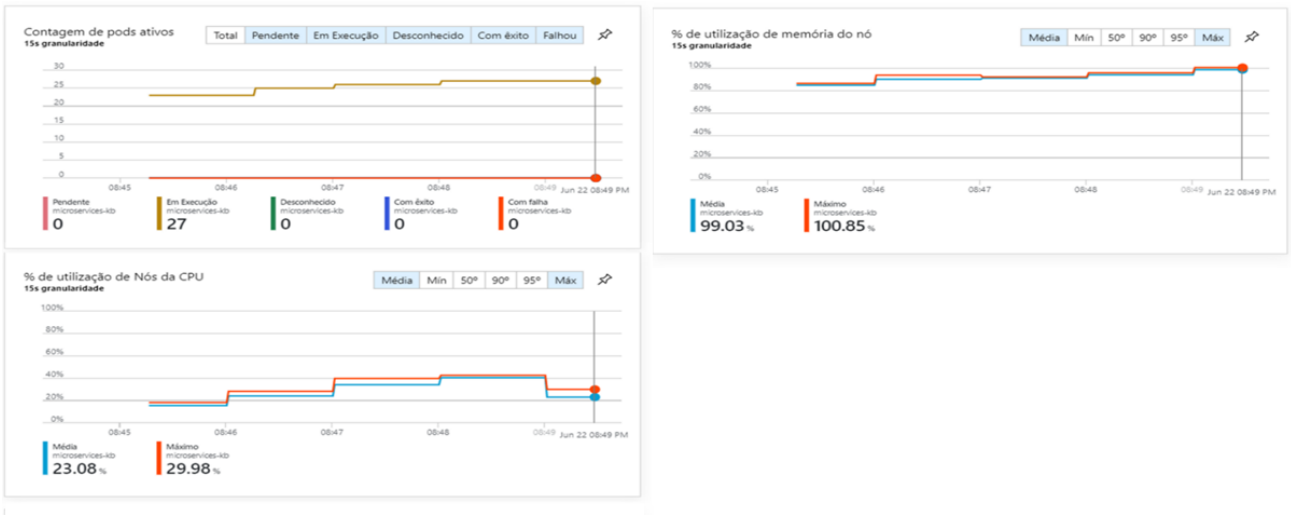

Figura 2 – Métricas da aplicação – Azure Insights.
Fonte: Elaboração dos autores (2020)

Como pode ser observado na Figura 2, os testes duraram cerca de 4 minutos; ao final desse período, apesar do baixo processamento, o consumo de memória RAM da aplicação chegou a 100% do total disponível no ambiente, sendo impossível a criação de novos Pods do serviço testado. Os testes foram conduzidos a partir dos computadores pessoais dos próprios desenvolvedores. Foram feitas 200 requisições por segundo durante o período de testes; nenhuma requisição foi rejeitada e o tempo médio de resposta foi de pouco menos de 120ms.

Os custos para o projeto foram estipulados pela própria plataforma Azure. Foram obtidas métricas usando como base o recurso *Cost Management*, disponível no Azure. A distribuição do custo aconteceu entre as máquinas virtuais necessárias para alocação do serviço de *Kubernetes* e as bases de dados. Como era de se esperar, o componente mais caro na solução foi o serviço de Kubernetes, pois ele é o grande responsável pelo gerenciamento, escalonamento e monitoramento de todos os demais serviços. Também é notório o valor gasto com armazenamento, principalmente pelo fato de cada serviço ter a necessidade de possuir sua própria base de dados.

O maior desafio ao se implementar este trabalho foi levantar as melhores práticas e tecnologias para o desenvolvimento de uma arquitetura de microsserviços. Devido ao fato de ser uma arquitetura utilizada por poucas companhias, principalmente por conta de sua complexidade, o levantamento das melhores práticas e padrões tornou-se ao longo dos dias um trabalho árduo e custoso para a equipe. No entanto, a partir do momento em que foram levantadas as melhores abordagens e os melhores padrões de desenvolvimento, o trabalho tornou-se mais simples, pois a técnica de adotar o que foi pesquisado facilitou a escolha dos melhores caminhos de implementação e a visualização de como seria o futuro desenvolvimento da arquitetura da aplicação.

Com o desenvolvimento do *backend*, onde está a implementação mais massiva da arquitetura, houve a possibilidade de se utilizar diversos tipos de tecnologias diferentes que comumente não se são encontradas nas aplicações convencionais.

O *deploy* e os testes com a arquitetura em produção também foram de grande valia para o conhecimento sobre as vantagens e desvantagens desse tipo de arquitetura de software, pois desafios que são desconhecidos em aplicações monolíticas tornaram-se visíveis neste tipo de arquitetura.

Ao longo do trabalho, foram levantados pontos críticos, sendo que estes serviram como lições aprendidas. Pode-se descrever os principais como sendo:

(1) Padronização das mensagens: adotar um padrão para as mensagens foi fundamental para o sucesso do presente trabalho. Como a arquitetura é constituída por diversos serviços, que por sua vez são escritos em diferentes linguagens, definir qual deveria ser o padrão de mensagem que os serviços deveriam possuir foi essencial, pois no início do projeto, devido a uma falta de padronização das mensagens, foram enfrentados diferentes problemas de integração entre os serviços. Portanto, pode-se inferir que, para o sucesso de uma arquitetura de microsserviços, faz-se necessária a definição de qual será o padrão das mensagens. Cada mensagem, portanto, continha uma informação relacionada ao evento em si a que estava vinculada: data de geração e versão do conteúdo interno. Além disso, o conteúdo era enviado no formato *JSON*, comumente utilizado para esse tipo de aplicação.

(2) Variáveis de ambiente: para o sucesso de uma arquitetura de microsserviços, se faz necessário que cada serviço fique totalmente customizável, ou seja, todas as suas dependências para outros serviços devem ser passadas para ele por meio de variáveis de ambiente. Este requisito da arquitetura já foi levantado anteriormente, de forma que seja possível migrar do ambiente de desenvolvimento para os demais ambientes apenas com a mudança das variáveis de ambiente (HEROKU, 2020).

(3) Serviços de apoio: alguns recursos de que a arquitetura necessita devem estar em um ambiente separado de onde se encontram os serviços. Bancos de dados, filas para mensagerias e gerenciador de arquivos são recursos essenciais para a aplicação, e nela devem ser acoplados como recursos extras, de forma que a independência deles seja preservada.

(4) Segregação correta das responsabilidades entre os serviços: numa primeira abordagem, desenvolveu-se o serviço de CRM utilizando-se o de mensageria para a comunicação interna de seus módulos. Tal abordagem mostrou-se ineficiente; conforme as requisições do teste de carga eram realizadas, o serviço de mensageria não seguia a proporção de escala do serviço. Com isso, a mensageria se tornava um gargalo na aplicação, já que tinha de lidar com mensagens trocadas entre os serviços e entre os seus módulos. Após essa constatação, o serviço foi refatorado, deixando de se utilizar a mensageria para seu funcionamento. Com isso, a escala do serviço pôde ser realizada com maior facilidade, sem que a mensageria se tornasse um gargalo.

Por fim, como produto da pesquisa, foram listados padrões e antipadrões. Alguns destes foram levantados em pesquisas bibliográficas, principalmente segundo Taibi, Lenarduzzi e Pahl, (2019), sobre arquitetura de microsserviços. Outros foram destacados no momento do desenvolvimento da arquitetura, de forma que os padrões que não estão presentes na bibliografia podem ser observados na Tabela 3, a partir do padrão “Mapa Arquitetural”.

Tabela 3 – Padrões e antipadrões

Padrão	Ação	Antipadrão evitado
Banco de dados privado	Cada serviço deve ter seu próprio banco de dados privado	Persistência de dados compartilhada
Uso de um *API Gateway*	Os serviços devem se comunicar através de um *software* de mensageria comum	Acesso direto aos serviços
Segregação	Os serviços devem ser segregados por negócio	Serviço segregado por camada
Escala	Os serviços devem ser disponibilizados em infraestrutura com escala automática	Serviço muito grande
Responsabilidade pelo serviço	Evitar que o mesmo time seja responsável por vários serviços	Time responsável por vários microsserviços
Mapa arquitetural	Os serviços devem ser regidos por um padrão arquitetural para facilitar a comunicação entre os times	Construção de serviços com o mesmo propósito
Uso de *Enterprise Service Bus* (ESB)	Os serviços devem se comunicar através de um software de mensageria comum.	Alto acoplamento de serviços
Padronização das mensagens	Os serviços deverão estabelecer um padrão para as mensagens que serão trocadas	Comunicação entre os serviços inconsistentes
Variáveis de ambiente	Parametrização e alteração dos ambientes de trabalho	Valores de configuração em código
Serviços de apoio	Desacoplamento de serviços periféricos	Acoplamento da aplicação
Segregação correta das responsabilidades entre os serviços	Cada serviço deve ter a sua responsabilidade bem definida	Excesso de responsabilidade no serviço de mensageria

Fonte: Adaptado de Taibi, Lenarduzzi e Pahl (2019).

Considerações finais

O objetivo do presente trabalho foi propor uma arquitetura de microsserviços para sistemas de ERPs de forma a levantar os melhores padrões e antipadrões desse tipo de arquitetura de software. Considerando todo o esforço mencionado acima, pode-se dizer que tal objetivo foi atingido.

É possível também afirmar que este tipo de arquitetura de software é uma forma viável de desenvolvimento de aplicações distribuídas que possuem requisitos de escalabilidade, manutenibilidade, independência entre os times e alta demanda, já previamente estabelecidos. Porém, é importante ressaltar que tal arquitetura depende de padrões mencionados neste trabalho e em outros com características semelhantes, além da necessidade de se levar em consideração o tamanho do projeto. Por se tratar de uma arquitetura projetada para a solução de problemas de larga escala, nem sempre sua adoção se mostra efetiva em projetos de pequeno porte, pois pode gerar uma alta complexidade para a concretização do projeto, o que prejudicará a performance da equipe de desenvolvimento sem trazer muitos benefícios nesse cenário em especificamente.

Como principais limitações para o presente trabalho, podem ser citadas a disponibilidade limitada dos recursos computacionais disponíveis dentro da plataforma escolhida, sendo esta limitação definida por questões financeiras. Também é válido destacar a pequena quantidade de regras de negócio da aplicação, o que aumenta diretamente a complexidade do projeto, pois trata-se de uma prova de conceito que procura validar os melhores padrões e antipadrões levantados nas pesquisas bibliográficas. Além disso, o tamanho limitado do time foi um dos grandes empecilhos para o desenvolvimento deste trabalho.

Como sugestões para futuros projetos, podemos propor o desenvolvimento da mesma aplicação utilizando a arquitetura *serverless*, ou ainda a construção de uma aplicação idêntica com recursos computacionais mais abrangentes, de forma que os problemas encontrados no presente trabalho possam vir a ser evitados.

Referências

APACHE. **AB - Apache HTTP server benchmarking tool.** Disponível em: https://httpd.apache.org/docs/2.4/programs/ab.html. Acesso em: 23 jun. 2020.

AROZO, R. Softwares de supply chain management: Definições, principais funcionalidades e implantação por empresas brasileiras. *In*: FIGUEIREDO, K.F.; FLEURY, P.F. & WANKE, P. **Logística e gerenciamento da cadeia de suprimentos: planejamento do fluxo de produtos e dos recursos.** Atlas: São Paulo, 2003.

BERNARDO, Felipe E. **A Evolução da Internet: uma perspectiva histórica**. Aslegis: Cadernos Aslegis, 2013.

BULLWINKLE, Michael. *et al.* **What is application insights?** Disponível em: https://docs.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview. Acesso em: 20/06/2020.

BORKAR, Vinayak R.; CAREY, Michael J.; LI, Chen. **Big Data Platforms. What's next?** New York: Association for Computing Machinery, 2016.

CURL. **curl.** Disponível em: https://curl.haxx.se/. Acesso em 23 jun. 2020.

DOCKER. **Get Started**. Disponível em: https://docs.docker.com/get-started/overview/. Acesso em 16 jun. 2020.

FACEBOOK. **Facebook Q2 2019 Results**. Disponível em: https://s21.q4cdn.com/399680738/files/doc_financials/2019/Q2/Q2-2019-Earnings-Presentation-07.24.2019.pdf. Acesso em: 02 nov. 2019.

GERSHENSON, J. K.; PRASAD, G. J.; ZHANG, Y. **Product modularity:** definitions and benefits**.** Abingdon: Taylor & Francis, 2003.

HALL, G. M. **Adaptive Code: agile coding with design patterns and SOLID principles.** Pearson Education. Redmond, 2017.

HEROKU. **The twelve factor app.** Disponível em: https://12factor.net/. Acesso em 21 jun. 2020.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. **ISO/IEC 25030:2019 - Systems and software engineering - Systems and software quality requirements and evaluation (SQuaRE) - Quality requirements framework**. Geneva: 2019.

JAMSHIDI, P. *et al.* **Microservices: the Journey so far and challenges ahead**. New York: IEEE, 2018.

JMETER. **Apache JMeter.** Disponível em: https://jmeter.apache.org/. Acesso em: 23 jun. 2020.

KARMAKAR, Uday. **Getting Control of Just-in-Time**. Massachusetts: Harvard Business Publishing, Harvard Business Review, 1989.

KEMP, S. **Digital 2019: q4 global digital statshot (2019)**. Disponível em: https://wearesocial.com/blog/2019/01/digital-2019-global-internet-use-accelerates. Acesso em: 02 nov. 2019.

KUBERNETES. **Pods**. Disponível em: https://kubernetes.io/docs/concepts/workloads/pods/pod/. Acesso em: 10 jul. 2020.

LEWIS, James; FOWLER, Martin. **Microservices - a definition of this new architectural term.** Disponível em: https://martinfowler.com/articles/microservices.html. Acesso em: 12 nov. 2019.

LINDSEY, C. H.; BOOM, H. J. **A Modules and Separate Compilation Facility for ALGOL 68**. Mountain View: ALGOL Bulletin, 1978.

MAIER, Mark W.; RECHTIN, Eberhardt. **The Art of Systems Engineering**. 2ª ed. Boca Ratón: CRC Press, 2000.

MARTIN, R. C. **Clean Architecture: a craftsman's guide to software structure and design**. Pearson Education. 2018.

SOMMERVILLE, I. **Engenharia de Software**. 10ª ed. Boston: Pearson Education, 2016.

TAIBI, D; LENARDUZZI, V; PAHL, C. **Microservices Anti-Patterns: a taxonomy.** Science and Engineering. Springer. 2019

TOLE, Alexandru Adrian. **Big Data Challenges.** Disponível em: http://www.dbjournal.ro/archive/13/13_4.pdf. Acesso em: 2 nov. 2019.

YOUTUBE. **YouTube para a imprensa**. Disponível em: https://www.youtube.com/intl/pt-BR/about/press/. Acesso em: 2 nov. 2019

ZIPKIN, Paul H. **Foundations of Inventory Management**, Boston: McGraw Hill, 2000.