Migrating a System for Event Participation Assistance to the Microservices Architecture

Victor Guerra Veloso
Universidade Federal de Viçosa
Florestal, Minas Gerais, Brasil
victorgvbh@gmail.com

Elena Augusta Araujo
Universidade Federal de Lavras
Lavras, Minas Gerais, Brasil
elena.araujo@posgrad.ufla.br

Thais R. de M. Braga Silva
Universidade Federal de Viçosa
Florestal, Minas Gerais, Brasil
thaisrmb@gmail.com

ABSTRACT

Monolithic architecture systems, which are systems composed of a single logical and executable unit, are, in general, hard to scale once any modification made in a small part of the system requires it to be rebuilt and redeployed as a whole. One way to ease this aspect, which is even more problematic in bigger information systems, is to follow modern low cost architecture proposals, as the one that advises the microservices usage. However, migrating legacy monolithic systems specifically to the microservice scenario is a challenging task, with proposals and background still incipient within the literature. This work consists of a migration proposal to the microservice architecture of a monolithic system for event participation assistance and the comparison of the software quality between the resulting product and the old version. For the migration results study and evaluation it was applied the metrics lines of code, comments rate, time of features conclusion and rework rate, those points to a maintainability degradation caused by the monolith complexity, which doesn't happen so steeply in the microservice architecture. After all, it is concluded that the microservice architecture adoption contributed to increased team expectation, development efficiency, good practices usage inducement, test development ease and speed of feature release to production (Time to market).

KEYWORDS

Arquitetura de Sistemas, Microsserviços, Eventos

ACM Reference Format:
Victor Guerra Veloso, Elena Augusta Araujo, and Thais R. de M. Braga Silva. 2019. Migrating a System for Event Participation Assistance to the Microservices Architecture. In *SBSI '19: Simpósio Brasileiro de Sistemas de Informação, May 20–24, 2019, Aracaju, Sergipe.* ACM, New York, NY, USA, 7 pages.

1 INTRODUÇÃO

Ao longo dos últimos anos, mudanças progressivas vêm ocorrendo na maneira tradicional em que as empresas estruturam seus recursos da tecnologia da informação, guiadas pela necessidade de desenvolver aplicações distribuídas e escaláveis [8, 15]. Na maneira tradicional, os sistemas de software são implementados por arquiteturas compostas por uma única unidade lógica executável, denominada arquiteturas monolíticas [33]. Nessa arquitetura, todas as funcionalidades são integradas, fazendo com que qualquer alteração realizada em uma pequena parte da aplicação exija que essa seja reconstruída e reimplantada [29]. Escalar sistemas de software monolíticos torna-se um grande desafio quando algumas funcionalidades são muito requisitadas. Nesses casos, todo o software deve ser escalado, acarretando em funcionalidades pouco requisitadas consumindo grande quantidade de recursos [33].

Com o objetivo de contornar os problemas enfrentados com a implantação de sistemas monolíticos em ambientes distribuídos, abstrações que utilizam serviços como elementos principais de implementação, tal como a Arquitetura de Microsserviços, têm sido propostos [15, 27, 33].

Fowler e Lewis definem o estilo arquitetural de microsserviços como uma abordagem para desenvolver uma única aplicação como uma *suite* de serviços independentes, cada um executando em seu próprio processo e se comunicando por meio de mecanismos leves através de uma API(*Application Programming Interface*) que utiliza o protocolo HTTP (*Hypertext Transfer Protocol*) [15]. Nessa arquitetura, uma aplicação monolítica deve ser dividida em serviços que implementam funcionalidades específicas, garantindo assim baixo acoplamento e alta coesão, além de permitir agilidade, flexibilidade, escalabilidade e reusabilidade [9, 16, 30]. Desta forma, a arquitetura de microsserviços compreende uma abordagem poderosa para manutenibilidade de Sistemas da Informação, uma vez que novos microsserviços podem ser desenvolvidos e integrados com impacto reduzido no restante da aplicação, quando surgem novas necessidades no negócio [4].

Diversas instituições que mantinham sistemas monolíticos, estão considerando a adoção da arquitetura de microsserviços [25, 34]. Segundo Taibi et al.[31] as principais motivações estão associadas a redução do alto custo de manutenção e o aumento da escalabilidade de seus serviços. Por outro lado, Luz et al.[20] apresenta a falta de um critério de decomposição como um problema crítico. Isso se agrava ainda mais quando considera a arquitetura monolítica uma etapa prévia e necessária de amadurecimento da equipe e do projeto no desenvolvimento para um determinado domínio [14].

O Sistema de Apoio à Participação em Eventos é um sistema desenvolvido para oferecer apoio à participação em eventos, tanto para organizadores como para participantes. Ele é composto por um website, por meio do qual eventos são criados e gerenciados, e um aplicativo, utilizado durante um evento para obtenção de informações e envio de *feedback*. As principais funcionalidades desse sistema voltadas para o organizador de um evento são: cadastro, criação de evento, inserção de atividades, questionários, notícias e patrocinadores para um evento e visualização de opiniões, notas

SBSI '19, May 20–24, 2019, Aracaju, Sergipe

e respostas à questionários, enviadas pelos participantes de um evento. Por outro lado, para os participantes, os principais serviços são: cadastro, seleção de um evento, visualização de programação e montagem de agenda para um evento, visualização de patrocinadores e notícias inseridas pelo organizador do evento, envio do respostas à questionários de um evento e envio de notas sobre atividades de um evento. No que diz respeito à esse últimos serviço, cada participante pode enviar, para cada atividade, uma nota sobre sua qualidade, bem como sobre o conforto térmico e o sonoro percebidos durante a mesma. O Sistema de Apoio à Participação em Eventos é um sistema desenvolvido por uma equipe de professores e alunos de um núcleo de pesquisa em sistemas pervasivos e distribuídos de uma instituição pública de ensino superior. As principais tecnologias envolvidas com o projeto do sistema são Ionic, Django e MySQL, tendo sido o mesmo desenvolvido inicialmente com arquitetura monolítica.

É fácil observar que o Sistema de Apoio à Participação em Eventos é um sistema aplicável a cenários bastante distintos, uma vez que existem eventos de porte e características bastante diferentes. Além disso, o número de eventos ocorrendo simultaneamente em um período é também bastante variável. Sendo assim, é corriqueira a necessidade de rearranjo das capacidades para diferentes serviços oferecidos, o que não é um trabalho fácil quando existe baixa coesão e, principalmente, alto acoplamento. Soma-se à essa dificuldade, o fato de que a equipe de trabalho é frequentemente alterada, pois é formada por estudantes. Assim sendo, a migração da arquitetura deste sistema para o modelo de microsserviços pode ser vista como interessante, tanto do ponto de vista de melhoria do processo de desenvolvimento, como para a qualidade do produto em si. No entanto, realizar a migração de um sistema pronto e em produção não é uma tarefa simples, considerando ainda que a literatura da área sobre qual estratégia seguir e quais os impactos causados mediante diversas métricas é ainda incipiente.

Nesse contexto, este trabalho possui como objetivo descrever o processo de migração de um sistema de apoio à participação em eventos que possui arquitetura monolítica, para uma versão de microsserviços. Dessa maneira, esse processo apresenta os benefícios e impactos da migração em termos de qualidade de software, como também permite a análise específica de vantagens e desvantagens para o caso do sistema em questão.

O restante deste artigo está dividido da seguinte forma: a Seção 2 apresenta os principais trabalhos relacionados encontrados na literatura. As Seções 3 e 4 trazem, respectivamente, os detalhes sobre o trabalho de migração para arquitetura de microsserviços realizado para um sistema de apoio à participação em eventos, bem como o impacto deste observado por meio de métricas de qualidade de software. Por fim, as conclusões e alguns possíveis trabalhos futuros podem ser encontrados na Seção 5.

2 TRABALHOS RELACIONADOS

Trabalhos que apresentam experiências de migração entre arquiteturas monolíticas e de microsserviços, bem como aqueles que apresentam ferramentas, *frameworks* ou estratégias de migração, podem ser considerados relacionados ao trabalho descrito neste artigo. A seguir estão descritos os principais trabalhos da literatura neste recorte, bem como é apresentada Tabela 1, que condensa as semelhanças e diferenças entre eles, com destaque para as diferenças trazidas pelo atual trabalho.

Fan and Ma descrevem a migração de um sistema de gerência de artigos condensados monolítico para a arquitetura de microsserviços, como forma de ilustrar seu processo de migração [13]. Esse trabalho descreve a utilização de conceitos de *Domain-Driven Design* (DDD), como contextos delimitados para identificação dos serviços. Ainda, esse utilizou-se de tecnologias, que, conforme a cultura DevOps, automatizam todo o processo de montagem, testes e implantação por meio da utilização de containers Docker. Adotou-se uma arquitetura de comunicação coreografada onde um serviço fornece a sincronização de dados e uma interface para sistemas de *front-end*. Os demais serviços possuem cada um seu próprio banco de dados, característica ressaltada pelo autor como desejável de aplicações baseadas em microsserviços.

No trabalho [2], Balalaie et al. propôs a migração da aplicação SSaaS (Server Side as a Service) para a arquitetura de microsserviços. Sua motivação era alcançar um nível de escalabilidade superior aquele já encontrado no sistema legado. Já em [13], DDD e DevOps foram utilizados para identificação de serviços e automatização de processos mecânicos, respectivamente. Destaca-se a adoção de containers Docker orquestrados por Kubernetes, o orquestrador de containers *open-source* desenvolvido e mantido pela Google, juntamente à *framework* Spring Cloud com alguns componentes do projeto Netflix OSS [26].

Referência	Estratégia				Tecnologias			Métricas QA[a]
	DevOps	DDD	Persistência	Coordenação	Orquestrador de containers	Netflix OSS	Docker	
[11]	-	-	-	-	-	-	-	Sim
[10]	Não	Não	Segregada	Coreografia	Não	Não	Não	Não
[2]	Sim	Sim	Segregada	Orquestração	Kubernetes	Sim	Sim	Não
[13]	Sim	Sim	Segregada	Coreografia	Não	Não	Sim	Não
[23]	Não	Não	Monolítica	Orquestração	Não	Sim	Não	Não
Este trabalho	Sim	Sim	Segregada[b]	Coreografia	Docker[c]	Não	Sim	Sim

[a]Métricas de qualidade de software.
[b]Segregada e Poliglota.
[c]Docker Engine (swarm mode) [12].

Tabela 1: Comparativo dos trabalhos relacionados

Ao longo da migração para a arquitetura de microsserviços, estratégias podem ser adotadas para tornar o processo gradual mais fluido. Autores como Medeiros et al. e Balalaie et al. em [23] e [2], utilizaram um componente externo para redirecionar as requisições realizadas a uma funcionalidade do sistema monolítico em migração. Outra abordagem é apresentada por Fan and Ma em [13], que consiste em criar e manter interfaces da linguagem de programação adotada para, somente quando desejável, alterá-las para o protocolo de comunicação por rede.

Em [11], cinco trabalhos de migração foram analisados quanto a sua conformidade a dez princípios da arquitetura de microsserviço. Estes princípios, em seguida, são correlacionados a um conjunto de seis métricas, de forma que torna-se possível alcançar conclusões objetivas quanto a qualidade da implementação dos microsserviços. As métricas selecionadas são aplicáveis a softwares, cujos modelos arquiteturais são baseados em serviços, e portanto não são aplicáveis a sistemas monolíticos. Ao fazer o estudo dirigido à métricas da qualidade da migração do Sistema de Apoio à Participação em Eventos, a título de comparação elegeu-se métricas compatíveis com ambos modelos arquiteturais [3].

Com base nos valores descritos na Tabela 1, é possível identificar as características de cada trabalho relacionado. Nota-se que a estratégia de persistência adotada neste trabalho pode ser identificada não apenas como segregada, mas também como poliglota[1]. Além disso, o novo orquestrador de containers da Docker Engine[12] proveu ferramentas alternativas a algumas daquelas encontradas no projeto Netflix OSS [26] ou no orquestrador de *containers* Kubernetes, como *Load Balancer* e *Service Discovery*.

3 ESTRATÉGIA DE MIGRAÇÃO: DE MONOLÍTICO A MICROSSERVIÇOS

3.1 O processo de migração arquitetural SPReaD

O processo SPReaD (*Service-oriented process for Reengineering and Devops*) foi proposto por Justino em [19] ao migrar um sistema monolítico para a arquitetura de microsserviços. Na Figura 1 é possível visualizar as etapas e fases do processo, divido entre as etapas de *Comunicação*, *Planejamento*, *Modelagem*, *Construção* e *Implantação*. A etapa de *Comunicação* consiste em identificar a dívida técnica e os requisitos gerais do sistema legado. Na etapa de *Planejamento*, uma instância do processo é criada, sendo definidas *milestones* para realização dos cálculos dos riscos e recursos necessários durante o processo de migração. A etapa de *Modelagem* é subdividida nas fases de análise a qual utiliza o DDD para identificar os serviços e criar documentos de análise; e de *design* – responsável pela elaboração dos diagramas da arquitetura alvo e estabelecimento dos contratos de cada serviço. Na etapa seguinte, de *Construção*, são elaborados os documentos para implementação de APIs de comunicação e de componentes do negócio, sendo responsável também pela implementação dos testes automatizados que serão executados e analisados na etapa seguinte. Por fim, a etapa de *Implantação* consiste na utilização de técnicas e ferramentas da cultura DevOps,

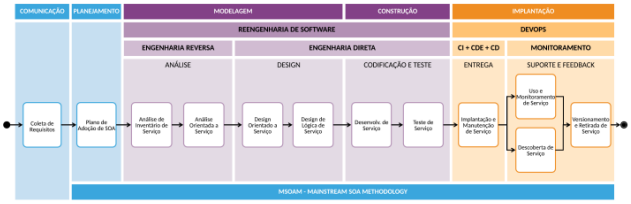

Figura 1: Visão geral do processo SPReaD
Fonte: Retirado de [19]

como CI/CDE/CD[2] e monitoramento de serviços, para automatizar o processo de empacotamento da aplicação e geração de *logs*.

3.2 Visão geral da migração

3.2.1 A arquitetura monolítica. O sistema de origem é composto por três entidades arquitetônicas e um banco de dados *MySQL*, como pode ser visto na Figura 2. A comunicação entre esses módulos consiste na manipulação de dados persistidos de forma centralizada, o que não só prejudica a escalabilidade do sistema, mas também contribui para o aumento da complexidade de manutenção desse. O *WebService*, desenvolvido na linguagem *PHP*, atua como uma interface de acesso ao banco para o aplicativo móvel. Esse, que é desenvolvido utilizando a *framework Ionic*, propicia ao usuário o favoritar e avaliar as atividades, receber notícias, responder questionários e publicar comentários sobre o evento. A plataforma web para administradores de eventos, desenvolvida de forma monolítica em *Python* com a *framework Django*, gerencia eventos e manipula o acesso aos *feedbacks* dos participantes.

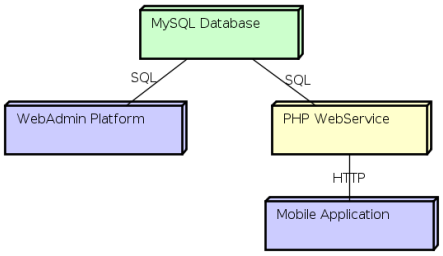

Figura 2: Diagrama de implantação da arquitetura monolítica.

3.2.2 Migrando o Sistema de Apoio à Participação em Eventos. Na etapa de *Comunicação*, identificou-se a demanda por segurança de acesso a informação do usuário da aplicação móvel, dado que os métodos de autenticação e autorização são adotados apenas na plataforma web para administradores de eventos. Assim, evidenciou-se a necessidade de refatoração da arquitetura e adoção de um modelo de comunicação que favoreça a extensão da aplicação.

Na etapa de *Planejamento*, a primeira versão do processo de migração foi elaborada e metas foram definidas juntamente à equipe responsável pelo sistema.

[1]Termo utilizado em [21] para se referir a utilização de tecnologias distintas de persistência para cada serviço

[2]Continuous Integration/Continuous Delivery/Continuous Deployment – ou Integração Contínua/Entrega Contínua/Implantação Contínua

Na subetapa de análise, a documentação do sistema monolítico foi readequada para o contexto de implantação de microsserviços. Elaborou-se o diagrama de casos de uso, o diagrama de implantação, conforme apresenta a Figura 2, e o diagrama de classes, gerado pela ferramenta Pylint Pyreverse [5]. Ainda, nesta subetapa, refatorou-se o Diagrama Entidade-Relacionamento, depreciando relações caracterizadas como redundantes e pouco coesas, possibilitando assim, a identificação de contextos delimitados, como descrito na Figura 3. Considerando a dimensão e o acoplamento dos dados dentro de

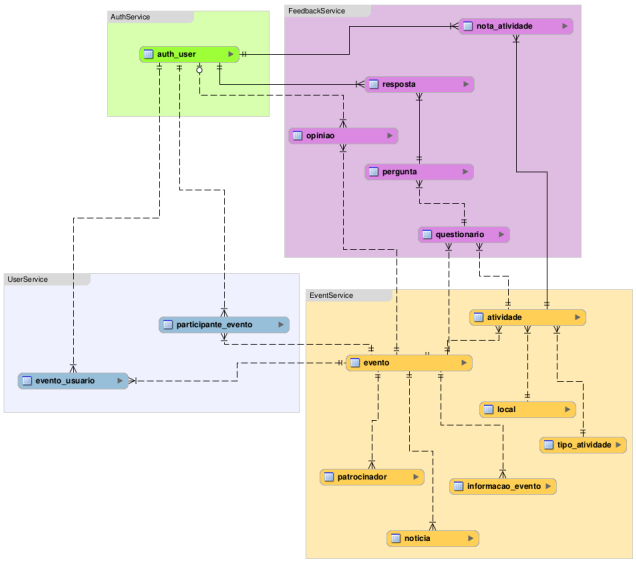

Figura 3: Diagrama Entidade Relacionamento indicando separação de serviços.

um contexto delimitado, na subetapa de *design* foram mapeados quatro microsserviços, como apresentado na Figura 4. Os serviços propostos são:

(1) *Evento*: serviço que agrupa todas as funcionalidades do interesse do administrador, como gerência de eventos, atividades e dados associados a patrocinadores, localização e notícias;
(2) *Feedback*: responsável pelos casos de uso associados aos participantes no que tange a gerência de *feedback* sobre atividades e eventos, ou seja, gerencia opiniões, avaliações e questionários;
(3) *Usuário*: representa o envolvimento de um usuário como participante e/ou administrador em um evento, como também as atividades favoritas daquele respectivo usuário;
(4) *Autenticação*: responsável por cadastro e login de usuários, como autenticação e autorização para as operações dos demais serviços.

A etapa de *Construção* consistiu na migração dos componentes de negócio já descritos na linguagem Python, para adequá-los ao padrão ORM (*Object-Relational Mapping*) definido no *microframework* Flask[7] adotado neste trabalho. A implementação da API e testes de cada microsserviço se deram mediante a seleção das tecnologias a serem adotadas (demonstradas na Tabela 2). A última etapa, *Implantação*, consistiu na execução automática das ferramentas e

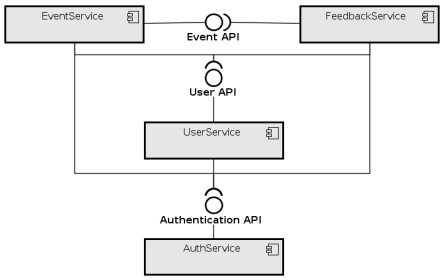

Figura 4: Diagrama de componentes da arquitetura proposta.

práticas do Pipeline DevOps à medida que *commits* eram submetidos ao sistema de controle de versão. Ao invés de monitorar os serviços, como previsto pelo processo SPReaD, foi nesta etapa que as métricas de qualidade de software foram calculadas e os gráficos plotados para estudo e comparação da arquitetura resultante da migração.

	Autenticação	Usuário	Evento	Feedback
Linguagem	Node.Js	Python	Python	Python
Banco de dados	Cassandra	MongoDB	MySQL	MySQL
Frameworks	Loopback	Flask	Flask	Flask

Tabela 2: Tecnologias aplicadas no processo de migração.

4 MEDIÇÃO DA QUALIDADE DA MIGRAÇÃO ARQUITETURAL

Após executar a estratégia de migração descrita na seção anterior, foram realizadas medições de qualidade de software nas versões arquiteturais monolítica e de microsserviço do Sistema de Apoio à Participação em Eventos, a fim de verificar se a nova versão apresenta as características de qualidade desejável. Dessa maneira, foram selecionadas as métricas linhas de código [3], taxa de comentários [24], abrangência de testes [22], tempo de conclusão de *features* [18] e taxa de retrabalho [6].

Os resultados obtidos, bem como a comparação entre ambas as versões são apresentadas nas próximas seções, destacando-se assim as vantagens e compromissos assumidos com a migração realizada.

4.1 Linhas de código

Uma métrica simples, comumente utilizada como indicador de complexidade de software. Seu cálculo consistiu na contagem de quebras de linha no código-fonte de cada serviço implementado e do sistema monolítico base. Seu resultado pode ser apurado desconsiderando as linhas com comentários, conforme apresentado na Seção 4.2.

Segundo o gráfico da Figura 5, nota-se uma considerável diferença entre a extensão do código da arquitetura monolítica e o resultado da migração. Isso é justificado pela remodelagem completa feita na estrutura do código, seleção de tecnologias adequadas para cada contexto delimitado e a maturidade do planejamento prévio à reimplementação.

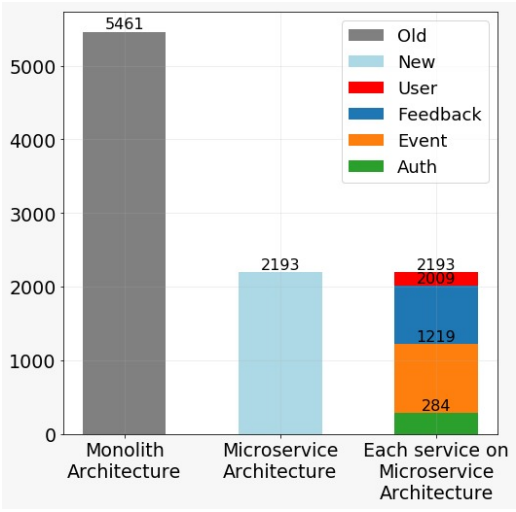

Figura 5: Gráfico de barras do número de linhas.

A arquitetura de microsserviços tem como um de seus pilares o aumento da manutenibilidade ao promover a quebra de um grande sistema em microsserviços. Uma equipe que vá realizar a manutenção a um destes serviços não precisará conhecer o código dos demais serviços, logo evidencia-se a importância de se conhecer o número de linhas de código de cada microsserviço, presente na Figura 5.

4.2 Taxa de comentários

Foi realizada a análise de linhas de código comentadas nos arquivos de código-fonte, descritos em sua totalidade nas linguagens Python e JavaScript. Para isso foi desenvolvido um *parser* para identificar os padrões de comentários de ambas as linguagens. Como resultados, na versão monolítica, o *parser* identificou uma taxa de comentários de apenas 5% (103 linhas de código comentadas, dentre as 2193 linhas). Já na versão com a arquitetura de microsserviços, foi reportada uma taxa de 8% de comentários (413 linhas comentadas dentre as 5461 referentes aos quatro serviços implementados). A Figura 6 apresenta a relação da porcentagem de comentários dos microsserviços implementados. Como pode ser observado, o serviço Evento possui a maior proporção de linhas comentadas e o serviço Usuário a menor.

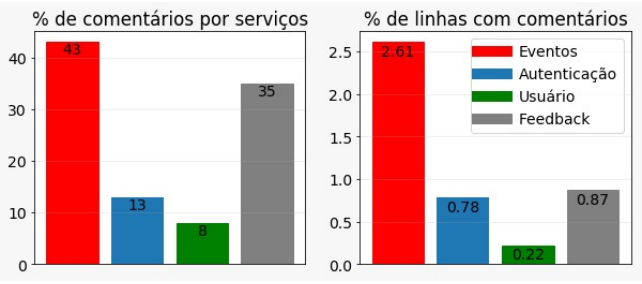

Figura 6: Gráfico da taxa de comentários detalhado.

Os resultados mostram que ambas implementações subutilizaram esse recurso. Isso pode acentuar a curva de aprendizado de novos contribuintes do projeto, principalmente se a documentação for pouco expressiva ou se ela estiver desatualizada, como era a realidade do sistema na arquitetura monolítica.

Ademais, é válido ressaltar que há no serviço de autenticação, uma considerável utilização de arquivos de configuração (correspondendo a 71,5% dos arquivos), no qual não foram considerados devido o padrão JSON não possuir suporte a comentários.

4.3 Abrangência de testes

Testes automatizados foram escritos para os microsserviços dos quais outros dependiam, com exceção do microsserviço de autenticação, uma vez que o seu código foi inteiramente gerado pela ferramenta Loopback CLI [17]. O cálculo da abrangência de testes consistiu na utilização da CLI do *framework* Pytest, uma suíte de execução de testes compatível com as principais bibliotecas de testes da linguagem Python.

Os resultados indicam que há 63% de abrangência de testes no microsserviço de eventos e 80% no microsserviço de usuário. Já na arquitetura monolítica testes não eram utilizados.

As principais consequência da utilização de testes no projeto foi o aumento da confiança no código e a facilidade no rastreio da origem de falhas, uma vez que a utilização de *mocking* possibilitou a execução dos testes em um serviço de maneira isolada e integrada aos demais serviços.

4.4 Duração de desenvolvimento de *features*

O intuito de calcular o tempo para se desenvolver uma *feature* é compreender a produtividade do processo de desenvolvimento e identificar qualquer degradação da produtividade ao longo do projeto. Para calcular essa métrica foi considerado tarefas presentes no software de gestão Trello[32], onde equipes estão associadas à quadros compostos por listas de *cards*. Devido a utilização desta ferramenta estruturada de forma semelhante ao quadro Kanban[1], bastou medir o tempo decorrido desde a criação de um *card* até o momento que este foi movido para lista de finalizados.

O gráfico utilizado para representar os resultados dessa métrica foi o diagrama de caixa ou "*boxplot*", como pode ser visto na Figura 7. Essa escolha foi devido a disparidade entre os desvios padrões dos valores capturados (166,12 da arquitetura monolítica e 22 da arquitetura de microsserviços), o que combinou bem com a capacidade desse gráfico em representar a variação dos valores, ressaltando a maior produtividade no desenvolvimento da arquitetura de microsserviços e a estabilidade do tempo de desenvolvimento de cada *feature* em comparação à arquitetura monolítica.

4.5 Taxa de retrabalho

Existem várias formas de calcular o retrabalho, mas a validade de cada método depende da forma com que a equipe de desenvolvimento trabalhou com *bugs*, *features* incompletas ou refatoração de código. No caso do sistema tratado neste trabalho, todo retrabalho era identificado pela utilização de um prefixo-chave na mensagem do *commit* e bastou contabilizar a frequência destes *commits*. Por

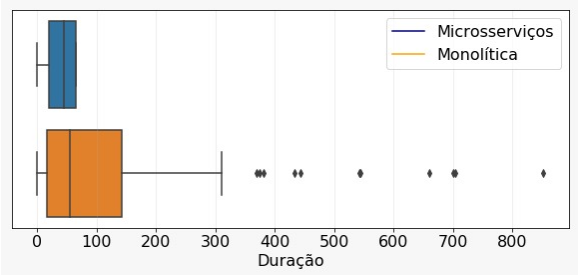

Figura 7: Boxplot do tempo de desenvolvimento de *features*.

outro lado, no processo de migração foi utilizado integração contínua, sendo o retrabalho identificado com base na frequência de *commits* recusados.

Diante dos valores demonstrados na Figura 8, é identificado que houve mais retrabalho durante a migração do que a implementação na arquitetura monolítica. Isso, contudo, pode ser justificado pela diferença entre as políticas de submissão de *commits* adotadas: durante o desenvolvimento do sistema monolítico, testes manuais eram praticados localmente concomitantemente à implementação e só depois disso um *commit* era feito; já no caso da migração para a arquitetura de microsserviços, testes automatizados eram escritos juntamente à implementação, mas eram executados em contêineres por servidores remotos como parte da integração contínua, ou seja após o commit a uma *branch* separada.

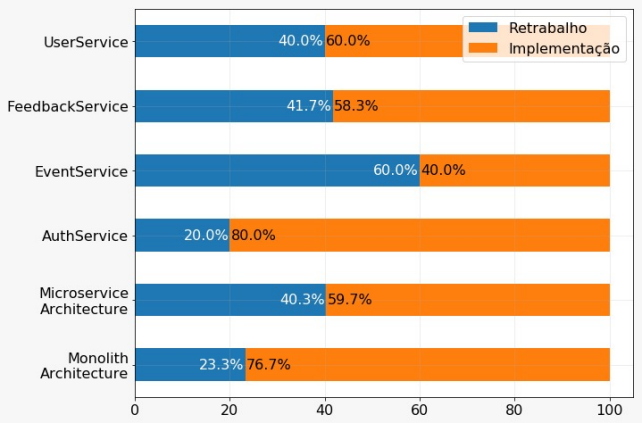

Figura 8: Gráfico da taxa de retrabalho.

5 CONCLUSÃO E TRABALHOS FUTUROS

Tradicionalmente, os sistemas, em geral, eram desenvolvidos com arquiteturas monolítica. Porém, devido às novas necessidades de escalabilidade das aplicações, estilos de arquitetura baseados em serviços foram propostos, sendo a mais recente a arquitetura de microsserviços.

Como em diversos trabalhos, este propõe a migração para a arquitetura de microsserviços objetivando melhorar a manutenibilidade e a escalabilidade de um sistema originalmente monolítico. Contudo o Sistema de Apoio à Participação em Eventos, desenvolvido em um núcleo de pesquisa de uma instituição pública de ensino superior por professores e alunos de iniciação científica, possui demandas restritas quanto a utilização dos servidores locais. Demandas como essa que acarretaram na utilização do novo orquestrador de containers acoplado ao Docker Engine, o swarm-mode [12].

Com base em diversos estudos e reuniões com a equipe responsável pelo sistema, foram identificadas oportunidades de divisão do mesmo em quatro serviços, compreendendo os serviços de autenticação, eventos, usuário e *feedback*. Nessa nova arquitetura há a possibilidade de desenvolvimento utilizando tecnologias distintas, visando a utilização daquela que for mais adequada para cada subdomínio (contexto delimitado) abrangido por um serviço. Neste projeto esta característica permitiu a adoção da linguagem Python e Javascript e bancos de dados MySQL, MongoDB e Cassandra.

A partir de metologias de DevOps ganhou-se agilidade no tempo de implantação e confiabilidade graças a execução de testes. Como maneira de verificar a nova arquitetura proposta, foram realizadas medições de qualidade de software aplicadas nas versões monolíticas e de microsserviços para o sistema de apoio à eventos. As métricas selecionadas foram linhas de código, taxa de comentários, abrangência de testes, tempo médio de desenvolvimento de *features* e taxa de retrabalho. Conclui-se que a nova versão vai auxiliar os novos alunos desenvolvedores da aplicação a entender melhor o código e a escalar os serviços mais requisitados durante as diversas etapas em que a aplicação é utilizada.

Como trabalho futuro pode-se promover a utilização de novas métricas de qualidade de software, como duplicação de código, medida do acoplamento entre os componentes de software e métricas relacionadas ao desempenho da aplicação. Além disso, também será interessante comparar a atual estratégia de comunicação, síncrona e por meio de HTTP, a uma estratégia orientada à eventos por meio da utilização de *brokers Publish-Subscribe* e filas de mensagens.

REFERÊNCIAS

[1] D.J. Anderson. 2010. *Kanban: Successful Evolutionary Change for Your Technology Business.* Blue Hole Press. https://books.google.com.br/books?id=RJ0VUkfUWZkC

[2] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. 2016. Migrating to Cloud-Native Architectures Using Microservices: An Experience Report. In *Advances in Service-Oriented and Cloud Computing*, Antonio Celesti and Philipp Leitner (Eds.). Springer International Publishing, Cham, 201–215.

[3] CARLOS FREUD ALVES BATISTA. 2007. *Métricas de Segurança de Software.* Ph.D. Dissertation. PUC-Rio.

[4] Jonathan Lincoln Brilhante, Rostand Costa, and Tiago Maritan. 2019. Projecting Microservices: A Study over Asynchronous Queues to Achieve Reactive Design. *iSys-Revista Brasileira de Sistemas de Informação* 12, 2 (2019), 117–153.

[5] Nicolas Chauvat, Stephen Medina, jcristau, and Shlomi Fish. 2018. Pylint - star your python code. Retrieved October 10, 2019 from https://www.pylint.org/

[6] T. Cheng, S. Jansen, and M. Remmers. 2009. Controlling and monitoring agile software development in three dutch product software companies. In *2009 ICSE Workshop on Software Development Governance.* 29–35. https://doi.org/10.1109/SDG.2009.5071334

[7] Davidism, Mitsuhiko, Adamchainz, and Etothemanders. 2019. Flask - a flexible and popular web development framework. Retrieved October 10, 2019 from https://palletsprojects.com/p/flask/

[8] Eduardo Santana de Almeida, Alexandre Alvaro, Daniel Lucrédio, Vinicius Cardoso Garcia, and Silvio Romero de Lemos Meira. 2004. Rise project: Towards a robust framework for software reuse. In *6th International Conference on Information Reuse and Integration (IRI).* 48–53.

[9] Mianxiong Dong, Kaoru Ota, and Anfeng Liu. 2015. Preserving source-location privacy through redundant fog loop for wireless sensor networks. In *14th International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing (IUCC).* 1835–1842.

[10] Luiz Chiaradia e Douglas Macedo e Moisés Dutra. 2018. Uma proposta de arquitetura de microsserviços aplicada em um sistema de CRM social. *Encontros Bibli: revista eletrônica de biblioteconomia e ciência da informação* 23, 53 (2018), 147–159. https://doi.org/10.5007/1518-2924.2018v23n53p147

[11] Thomas Engel, Melanie Langermeier, Bernhard Bauer, and Alexander Hofmann. 2018. Evaluation of Microservice Architectures: A Metric and Tool-Based Approach. In *Information Systems in the Big Data Era*, Jan Mendling and Haralambos Mouratidis (Eds.). Springer International Publishing, Cham, 74–89.

[12] Docker Core Engineering. 2016. Run Docker Engine in swarm mode. Retrieved October 10, 2019 from https://docs.docker.com/engine/swarm/swarm-mode/

[13] C. Fan and S. Ma. 2017. Migrating Monolithic Mobile Application to Microservice Architecture: An Experiment Report. In *2017 IEEE International Conference on AI Mobile Services (AIMS)*. 109–112. https://doi.org/10.1109/AIMS.2017.23

[14] Martin Fowler. 2015. MonolithFirst. Retrieved September 28, 2019 from https://www.martinfowler.com/bliki/MonolithFirst.html

[15] Martin Fowler and James Lewis. 2014. Microservices: a definition of this new architectural term. Retrieved September 28, 2019 from https://martinfowler.com/articles/microservices.html.

[16] Paolo Di Francesco. 2017. Architecting Microservices. In *1th International Conference on Software Architecture Workshops (ICSAW)*. 224–229.

[17] IBM. 2019. LoopBack 3.x. Retrieved September 28, 2019 from https://loopback.io/doc/en/lb3/index.html

[18] C. R. Jakobsen and T. Poppendieck. 2011. Lean as a Scrum Troubleshooter. In *2011 Agile Conference*. 168–174. https://doi.org/10.1109/AGILE.2011.11

[19] Yan de Lima Justino. 2018. *Do monolito aos microsserviços: um relato de migração de sistemas legados da Secretaria de Estado da Tributação do Rio Grande do Norte*. Master's thesis. Brasil.

[20] Welder Luz, Everton Agilar, Marcos César de Oliveira, Carlos Eduardo R. de Melo, Gustavo Pinto, and Rodrigo Bonifácio. 2018. An Experience Report on the Adoption of Microservices in Three Brazilian Government Institutions. In *Proceedings of the XXXII Brazilian Symposium on Software Engineering (SBES '18)*. ACM, New York, NY, USA, 32–41. https://doi.org/10.1145/3266237.3266262

[21] Leonardo Guerreiro Azevedo Luís Henrique Neves Villaça, Antônio Francisco Pimenta Jr. 2018. Construindo Aplicações Distribuídas com Microsserviços. In *SBSI'18: Proceedings of the XIV Brazilian Symposium on Information Systems*. ACM, Caxias do Sul, Brazil.

[22] Y. K. Malaiya, M. N. Li, J. M. Bieman, and R. Karcich. 2002. Software reliability growth with test coverage. *IEEE Transactions on Reliability* 51, 4 (Dec 2002), 420–426. https://doi.org/10.1109/TR.2002.804489

[23] Otávio Medeiros, Américo Sampaio, and Augusto Arraes. 2018. Uso de AOP na Migração de Aplicações Monolíticas para Microservices. In *Anais do XVI Workshop em Clouds e Aplicações (WCGA - SBRC 2018)*, Vol. 16. SBC, Porto Alegre, RS, Brasil. https://portaldeconteudo.sbc.org.br/index.php/wcga/article/view/2382

[24] Karine Mordal, Nicolas Anquetil, Jannik Laval, Alexander Serebrenik, Bogdan Vasilescu, and Stéphane Ducasse. 2013. Software quality metrics aggregation in industry. *Journal of Software: Evolution and Process* 25, 10 (2013), 1117–1135. https://doi.org/10.1002/smr.1558 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1558

[25] Ola Mustafa, Jorge Marx Gómez, Mohamad Hamed, and Hergen Pargmann. 2018. GranMicro: A black-box based approach for optimizing microservices based applications. In *From Science to Society*. Springer, 283–294.

[26] Netflix. [n. d.]. Netflix Open Source Software Center. Retrieved October 10, 2019 from https://netflix.github.io/

[27] Sam Newman. 2015. *Building microservices: designing fine-grained systems*. "O'Reilly Media".

[28] Joyce Aline Oliveira and Jose J.L.D. Junior. 2016. A Three-dimensional View of Reuse in Service Oriented Architecture. In *Proceedings of the XII Brazilian Symposium on Information Systems on Brazilian Symposium on Information Systems: Information Systems in the Cloud Computing Era - Volume 1 (SBSI 2016)*. Brazilian Computer Society, Porto Alegre, Brazil, Brazil, Article 54, 8 pages. http://dl.acm.org/citation.cfm?id=3021955.3022024

[29] Dmitry I. Savchenko, Gleb I. Radchenko, and Ossi Taipale. 2015. Microservices validation: Mjolnirr platform case study. In *38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 235–240.

[30] Alan Sill. 2016. The design and architecture of microservices. *IEEE Cloud Computing* 3, 5 (2016), 76–80.

[31] D. Taibi, V. Lenarduzzi, and C. Pahl. 2017. Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. *IEEE Cloud Computing* 4, 5 (Sep. 2017), 22–32. https://doi.org/10.1109/MCC.2017.4250931

[32] Trello. [n. d.]. Simples à primeira vista, mas com muitas surpresas. Retrieved October 10, 2019 from https://trello.com/tour

[33] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. 2015. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *10th Computing Colombian Conference (10CCC)*. 583–590.

[34] Pujianto Yugopuspito, Frans Panduwinata, and Sutrisno Sutrisno. 2017. Microservices architecture: case on the migration of reservation-based parking system. In *2017 IEEE 17th International Conference on Communication Technology (ICCT)*. IEEE, 1827–1831.