Submission date: 12/11/2018
Resubmission date: 11/02/2019
Camera ready submission date: 10/04/2019

1st round notification: 14/01/2019
2nd round notification: 25/03/2019
Available online: 10/04/2019
Publication date: 03/05/2019

Section: Systems and Prototypes

Projetando Microsserviços Reativos: Um Estudo de Refatoração de Aplicações Monolíticas sobre a Ótica de Orquestração e Elasticidade de Fluxos de Dados Internos Title: Designing Reactive Microservices: A Study of Refactoring Monolithic Applications on the Optics of Orchestration and Elasticity of Internal Data Flows

Jonathan Brilhante[1], Rostand Costa[1], Tiago Maritan[1]

[1]Universidade Federal da Paraíba
Centro de Informática
Lab. de Aplicações de Vídeo Digital
João Pessoa, Paraíba – Brasil

jonathan.brilhante@lavid.ufpb.br, rostand@lavid.ufpb.br,
maritan@lavid.ufpb.br

Abstract. *Microservices (MS) are a new approach in order to increase scalability and flexibility in large applications. One of the biggest challenges with the MS approach is how to properly develop a single service in terms of its scope, efficiency and reliability. This study proposes an architectural model to internally structure a microservice, inspired by concepts of reactive programming. The proposed model is based on the internal coordination of the microcomponents of a microservice through message queues, for orchestration of the internal data flows, and through the use of adapters, to allow eventual reuse and / or interoperation of legacy source code of applications or other microservices monolithic. A comparative study between the traditional approach and the proposed architecture was carried out with the objective of measuring the impacts of this strategy in terms of performance, elasticity and resilience. The results obtained in two real-world case studies of system refactoring into microservices have demonstrated that the combined use of message queues, wrapers, pool of workers, and orchestration of internal data streams have provided not only ease in decomposing monolithic applications but also relevant gains in time response and fault tolerance.*

Resumo. *Microsserviços (MS) são uma nova abordagem projetada com o intuito de incrementar a escalabilidade e flexibilidade em grandes aplicações. Um dos maiores desafios com a abordagem de MS está no desenvolvimento adequado de um único serviço em termos do seu escopo, eficiência e confiabilidade. Neste trabalho é proposto um modelo arquitetural para estruturar internamente*

Cite as: Brilhante, J. Rostand, C. & Maritan, T. (2019). Designing Reactive Microservices: A Study of Refactoring Monolithic Applications on the Optics of Orchestration and Elasticity of Internal Data Flows (Projetando Microsserviços Reativos: Um Estudo de Refatoração de Aplicações Monolíticas sobre a Ótica de Orquestração e Elasticidade de Fluxos de Dados Internos).
iSys: Revista Brasileira de Sistemas de Informação (Brazilian Journal of Information Systems), 12(2), 117-153.

um microsserviço, inspirada em conceitos da programação reativa. Tal modelo se baseia na coordenação interna dos microcomponentes de um microsserviço através de filas de mensagens, para orquestração dos fluxos de dados internos, e através do uso de adaptadores, para permitir o eventual reuso e/ou interoperação de código fonte legado de aplicações ou outros microsserviços monolíticos. Um estudo comparativo entre a abordagem tradicional e a arquitetura proposta foi executado com o objetivo de medir os impactos desta estratégia em termos de desempenho, elasticidade e resiliência. Os resultados obtidos em dois estudos de casos reais de refatoração de sistemas em microsserviços demonstraram que o uso combinado de filas de mensagens, wrapers, pool de workers e orquestração dos fluxos de dados internos proporcionaram tanto facilidade na decomposição de aplicações monolíticas quanto ganhos relevantes em tempo de resposta e tolerância a falhas.

1. Introdução

Atualmente, o surgimento de novas abordagens para a construção de aplicações vem trazendo uma série de benefícios com relação ao modelo arquitetural mais tradicional, denominado modelo monolítico [Richardson 2015]. O modelo de microsserviços começa a emergir nesse cenário como uma alternativa que propõe transformar um sistema único (monolítico) em um conjunto de pequenos serviços, ou microsserviços (MS), que cooperam e podem ser desenvolvidos, implantados e gerenciados de forma independente [Newman 2015].

Cada microsserviço é organizado para endereçar uma das necessidades do negócio, e pode ser ativado de acordo com um *workflow* próprio de cada operação. Este tipo de organização e comunicação permite que os microsserviços de uma mesma aplicação possam ser codificados em linguagens de programação distintas e possam utilizar plataformas que se adequem melhor as suas especificidades [Fowler and Lewis 2014].

Assim sendo, tem-se na arquitetura de microsserviços uma abordagem poderosa para manutenibilidade de Sistemas da Informação (SI), uma vez que novos microsserviços podem ser desenvolvidos e integrados com impacto reduzido no restante da aplicação, quando surgem novas necessidades no negócio. Não apenas a manutenção, como toda logística para o provisionamento do SI se beneficia desta arquitetura, uma vez que estes possam ser implantados em múltiplas plataformas e/ou empresas, flexibilizando contratos e otimizando custos [Saarimäki et al. 2019].

Entretanto, o desenvolvimento de novas aplicações baseadas em microsserviços, ou a migração de uma aplicação existente para microsserviços, não é simples, e muitas questões surgem no momento de decompor suas funcionalidades em novos serviços. Dentre elas, pode-se destacar:

- **Como estruturar um microsserviço?**
- **Qual deve ser o tamanho e escopo de cada microsserviço?**
- **Como refatorar um serviço monolítico para uma arquitetura de microsserviços?**

- **Como garantir que este novo microsserviço não repita os problemas de uma arquitetura monolítica?**

A decomposição da aplicação em microsserviços é um dos principais desafios para a construção de um sistema baseado na arquitetura de MS [Balalaie et al. 2015][Kecskemeti et al. 2016]. Apenas com uma estruturação adequada dos microsserviços integrantes e uma boa definição das interfaces de comunicação, é possível alcançar um baixo índice de acoplamento e um alto grau de coesão, fundamentais para uma boa orquestração e gerenciamento do sistema [Daniel and Pernici 2006].

Com o objetivo de permitir uma melhor estruturação e projeto de arquiteturas de serviços Web, uma abordagem emergente é a proposta no *Manifesto Reativo* [Bonér et al. 2014]. Este manifesto tem como proposta desenvolver sistemas de informação mais robustos, mais resistentes, mais flexíveis e melhor posicionados para sustentar as demandas modernas. Essas transformações na forma de construir os sistemas de informação estão acontecendo por causa das mudanças nos seus requisitos, as quais vêm ocorrendo nos últimos anos. Há apenas alguns anos, as grandes aplicações podiam conter uma(s) dezena(s) de servidores; demorar alguns segundos para responder as requisições; conter manutenções que podiam demorar algumas horas para serem desenvolvidas e implantadas, e um tráfego de alguns gigabytes de dados.

Atualmente, existem aplicações em produção em todos os lugares, desde aplicativos móveis até aplicações na nuvem, com *clusters* rodando em milhares de processadores com múltiplos núcleos. Além disso, os usuários geralmente esperam respostas em poucos milissegundos, que as aplicações estejam disponíveis em tempo integral, e com tráfego de dados relativamente maior. Para endereçar essas mudanças nos requisitos, o Manifesto Reativo organiza e apresenta as diretrizes necessárias para a construção de serviços que permitam um provisionamento dinâmico e em larga escala, e vem sendo um guia para o desenvolvimento de padrões arquiteturais em MS.

De acordo com o Manifesto, os sistemas reativos devem ser responsivos, resilientes, elásticos e orientados a mensagens. Além disso, eles devem ser mais tolerantes a falhas, e quando as elas ocorrerem devem ser tratáveis, trazendo impactos menores para o sistema [Bonér et al. 2014]. Partindo da premissa que os sistemas de informação criados para serem reativos são mais flexíveis, desacoplados e escaláveis, este trabalho se propõe a avaliar se a aplicação do paradigma reativo também na estruturação interna dos microsserviços pode ajudara a trazer esses mesmos benefícios para cada componente isolado do sistema.

Assim, neste trabalho, é investigado o uso de técnicas de decomposição e reestruturação de aplicações monolíticas na estruturação interna de microsserviços, adotando como diretrizes as normas e padrões de serviços reativos. A principal questão de pesquisa abordada é: "Qual a efetividade da aplicação dos princípios reativos na organização interna de microsserviços sobre a sua resiliência e elasticidade?". Neste sentido algumas hipóteses foram levantadas: i) O uso de filas de mensagens na orquestração dos fluxos de dados internos de um microsserviço melhora a sua tolerância a falhas; ii) o uso de *wrapers* baseados em clientes para filas de mensagens habilitam o aproveitamento de código legado e na inter-operacionalização de componentes de microsserviços escritos em

linguagens diferentes; iii) o acoplamento de *pools* de *workers* a filas de mensagens aumenta a elasticidade lateral de microsserviços; iv) microsserviços reativos permitem um melhor aproveitamento do *hardware* disponível. Com o objetivo de realizar um estudo comparativo entre a abordagem proposta e os métodos tradicionais, um estudo de caso foi planejado e realizado utilizando sistemas reais. Os resultados obtidos mostram que alguns dos desafios para a adoção da arquitetura de MS, tais como, a complexidade e o desempenho do sistema, podem ser tratados com a abordagem proposta.

O restante do documento está organizado da seguinte forma. Na Seção 2 são apresentados alguns trabalhos relacionados com a proposta deste trabalho. Na Seção 3, são apresentados os principais conceitos relacionados aos microsserviços, os maiores desafios das pesquisas neste ramo, bem como os principais padrões arquiteturais para criação de MS. Em seguida, na Seção 4, são listadas as diretrizes dos sistemas reativos e como a abordagem proposta utiliza tais conceitos. Na Seção 5, é apresentado o protocolo dos estudos de caso que foram utilizados para avaliar a metodologia proposta. Os resultados obtidos e a sua respectiva análise e discussão são apresentados nas Seções 6 e 7, respectivamente. Por fim, as considerações finais e algumas indicações de trabalhos futuros são apresentados na Seção 8.

2. Trabalhos Relacionados

A preocupação em como projetar um sistema estruturado em microsserviços existe há alguns anos. Balalaie *et al.* [Balalaie et al. 2015] iniciaram a discussão de como projetar um microsserviço. Seu trabalho apresentou um conjunto de desafios e questões que devem ser ponderadas durante a modelagem de um MS, tais como: "como decompor uma arquitetura de design orientado por domínio (*Domain-Driven Design*) para microsserviços?"ou "o que fazer com a dependência de código entre módulos?".

Johanson *et al.* [Johanson et al. 2016] e Dragoni *et al.* [Dragoni et al. 2016] apresentaram uma proposta semelhante a de Balalaie *et al.* [Balalaie et al. 2015], mas a metodologia escolhida por eles para tratar o armazenamento dos dados era diferente. Arne, em sua aplicação de suporte a pesquisas oceânicas (OCEANTEA), decidiu por incluir no escopo de cada microsserviço o próprio controle sobre os dados. O sistema FX Core System, de suporte bancário, foi decomposto com base no design dirigido a domínios pré-existente. Dessa forma, a comunicação passou a ser gerenciada por um *middleware* de fila de mensagens, neste caso o *RabbitMQ*, e o armazenamento passou a ser mais um destes serviços. Infelizmente, em ambos os casos, nenhuma análise existiu sobre o impacto que tais decisões acarretaram sobre a aplicação. A maior contribuição destes estudos de caso esteve no detalhamento técnico sobre o processo de *deployment* e reuso de bibliotecas.

Gouigoux e Tamzalit [Gouigoux and Tamzalit 2017] apresentam um conjunto de estratégias aprendidas e exploradas no trabalho de refatoramento de sistemas junto a empresa MGDIS. Para eles, os microsserviços possuem um forte relacionamento com a tecnologia de conteinerização, em especial a implementação Docker, que contém um ferramental importante para reduzir os custos de *deployment*.

Nesse trabalho, os autores sugerem definir a granularidade de um serviço pelo balanço entre o custo necessário para a implantação e o dispêndio com a validação dos

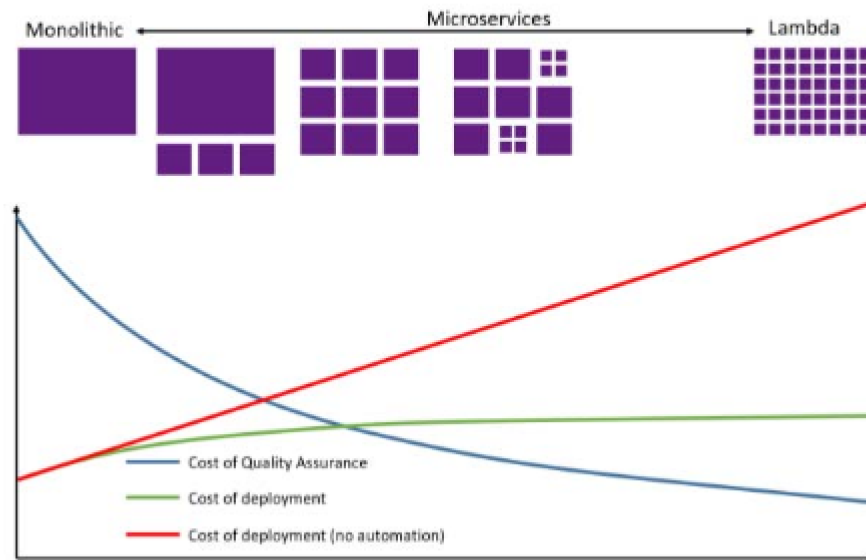


Figura 1. Gráfico para Granularidade de Microsserviços
Fonte: GOUIX, 2017.

novos microsserviços, como ilustrado pelo gráfico da Figura 2. Em outras palavras, esta atividade faz parte de um processo difícil de tentativa e erro, onde quanto maior o custo para validar novos microsserviços ou quanto menor o custo para novos *deployments*, mais granular deve ser o sistema final, sendo esta a principal contribuição da pesquisa de Gouigoux e Tamzalit.

Uma tentativa de formalizar um modelo para decomposição de microsserviços foi proposta por Levcovitz *et al.* [Levcovitz et al. 2016]. De acordo com os autores, os sistemas monolíticos são divididos em função de suas partes [Sill 2016]. Muito comumente, esta divisão é estruturada em 3 camadas: Apresentação, Lógica de Negócio e Armazenamento. Na Figura 2 é apresentado um modelo genérico de uma arquitetura monolítica segundo Levcovitz. Cada uma dessas camadas podem ser enxergadas internamente como um conjunto de funções ou microcomponentes.

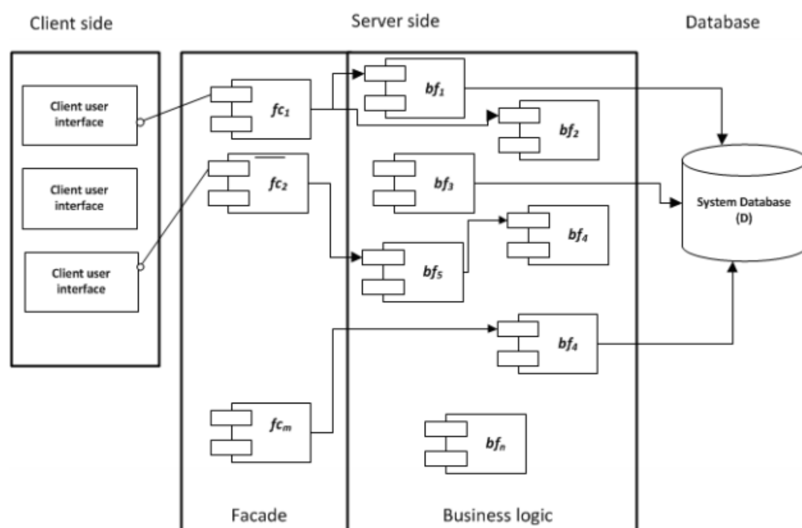


Figura 2. Representação Genérica de Aplicações Monolíticas. Fonte: Levcovitz, arXiv:1605.03175, 2016 [Levcovitz et al. 2016]

Levcovitz define uma metodologia para extração de microsserviços de sistemas monolíticos e formaliza um passo a passo baseado nas práticas mais comuns do mercado. Nesta abordagem, cada par composto por "fachada de comunicação"e "rotina da lógica de negócio"pode vir a compor um novo microsserviço baseado na dependência entre eles.

Um novo conjunto de objetos de persistência (como, por exemplo, as tabelas do banco de dados) pode ser incorporado a este microsserviço em potencial, assim como outras rotinas de negócio, quando estas estiverem relacionadas com o recorte da aplicação. Por exemplo, na Figura 2, a fachada "fc1"e os componentes de negócio "bf1"e "bf2"formam um subconjunto candidato à refatoração em um novo microsserviço.

A Tabela 1 apresenta um resumo de algumas características dos trabalhos apresentados nesta seção. Conforme pode ser observado na Tabela 1, uma das principais limitações desses trabalhos é que apenas um deles discute a arquitetura interna de um microsserviço.

Através desta compilação de trabalhos, é possível observar que existem poucos estudos sobre microsserviços, o que dificulta a sua adoção como um padrão estrutural. Nos últimos anos, contudo, tem crescido o número de trabalhos que procuram entender, de fato, o que "consiste"um microsserviço e quais são os seus limites. Embora a tarefa de visualizar se um serviço é ou não modularizável em microsserviços seja mais simples, ainda é difícil identificar ou definir qual é o melhor processo para se fazer essa modularização. A maioria dos trabalhos abordam estudos de casos para aplicar isso em um sistema ou em um conjunto específico de sistemas. Diferentemente deles, por outro lado, este trabalho foca numa nova metodologia para organização interna de microsserviços com base nas diretrizes da programação reativa.

Tabela 1. Resumo dos Trabalhos Analisados

Aplicação	Primeiro Autor	Ano Publicado	Estratégia de Decomposição/ Migração	Discute a Arquitetura Interna de um Microsserviço	Tecnologias Utilizadas
Server-Side as a Server	Armin Balalaie	2015	Domain-Driven Design	Não	Docker e RESTful
Backtory	Armin Balalaie	2015	Domain-Driven Design (DevOPs)	Não	Docker, RESTful e Spring
OCEANTEA	Johanson Arne	2016	Domain-Driven Design (com bibliotecas internas)	Não	Docker e RESTful
FX Core System	Dragoni Nicola	2016	Domain-Driven Design	Não	Docker, RabbitMQ e Remote Procedure Calls
Banco Brasileiro	Levcovitz	2016	Componentes de 3-camadas	Sim	Docker e RabbitMQ
MGDIS	Jean Gouigoux	2017	Custo x Validação	Não	Docker

3. Decompondo Aplicações Monolíticas em Microsserviços

Frequentemente as aplicações são organizadas em vários subprocessos de negócio. Como visto anteriormente, nem sempre estes apresentam a autonomia funcional necessária para constituir um microsserviço, uma vez que possuem dependências e/ou escopos limitados. No decorrer deste trabalho, será apresentada uma abordagem inovadora para organização destes micro-componentes em uma nova entidade na arquitetura interna de um microsserviço.

3.1. Definindo o Escopo de Microsserviços

Um dos principais desafios para a refatoração de uma aplicação monolítica está em como extrair corretamente um microsserviço. Parte desta dificuldade reside em ter um perfeito entendimento das características que distinguem um microsserviço de um componente de *software*. Basicamente, a principal diferença entre eles está no escopo e na forma de distribuição. Enquanto um componente, pacote ou biblioteca de software tem um enfoque em prover um conjunto isolado de funções para outras entidades internas da aplicação, um microsserviço tem como escopo a operacionalização integral de uma atividade ou rotina de negócio [Newman 2015].

Neste sentido, a principal propriedade de um microsserviço, e que o classifica como tal, é a atomicidade da sua funcionalidade. Esta precisa ser única, singular e autônoma, de forma que seja modular e independente de outros elementos externos para produzir as suas saídas e prover suas interfaces [Dragoni et al. 2016].

Nem sempre a melhor estratégia para a delimitação do escopo é clara ou definitiva. Definir as fronteiras de um microsserviço é algo que depende do projetista, da natureza da aplicação e até mesmo da configuração e capacidades das equipes de desenvolvimento. De forma geral, 3 diretrizes podem guiar a definição de um bom microsserviço [Thönes 2015]:

- **Fraco Acoplamento**: Um microsserviço ideal deve ter dependência mínima com

outros serviços. A comunicação com as eventuais dependências necessárias deve ser feita unicamente através de interfaces públicas (API REST, eventos, troca de mensagem) e tais interfaces não devem expor nenhum processo interno.
- **Alta Coesão**: Funções associadas ao mesmo escopo devem ser agrupadas e disponibilizadas em um mesmo microsserviço. Com isto, minimiza-se a troca de mensagens desnecessárias entre microsserviços.
- **Único Contexto**: Um microsserviço deve cobrir os limites de apenas um contexto. Estes contextos delimitam detalhes internos de domínio e negócio.

3.2. Construindo Aplicações com Microsserviços

Assim como ocorre na definição do escopo de um microsserviço, a definição do modelo arquitetural de uma aplicação baseada em microsserviços também traz alguns desafios. As estratégias mais utilizadas atualmente foram inspiradas por *design patterns* [Gupta 2015] já adotados em *webservices* e são apresentadas a seguir:

- *Aggregator*: Este padrão é o mais simples e intuitivo. Nele, um agregador invoca os microsserviços baseado na operação requerida. Este componente central age como um controlador do conjunto de microsserviços da aplicação, os quais são independentes entre si. Cabe ao agregador invocar e recuperar os resultados de cada um dos microsserviços envolvidos e por fim disponibilizar o resultado final para o usuário.
- *Proxy*: Diferentemente do *Aggregator*, no padrão *Proxy* não é feito nenhum processamento no componente que concentra as requisições e ativa os microsserviços. Qualquer lógica de negócio deve ser tratada internamente nos microsserviços. Dessa forma, embora também exista um componente centralizador, o mesmo atua apenas como um *proxy*, escolhendo e repassando as requisições para os microsserviços associados. Os microsserviços neste padrão também são independentes um dos outros.
- *Chained*: Neste padrão, o cliente acessa de forma síncrona apenas um microsserviço de entrada *A*, o qual, para concluir sua tarefa, ativará um segundo microsserviço *B*, que por sua vez invoca (se necessário) um novo microsserviço *C* e assim por diante. Essas conexões sequenciais dão a ideia de uma cadeia ou corrente, simplificando a resposta ao cliente.
- *Branch*: Na arquitetura em *Branch*, a requisição pode ser respondida por um ou mais microsserviços, sendo que a escolha depende do contexto e de critérios preestabelecidos. A grande diferença do padrão *Branch* em comparação ao *Aggregator* está no fato de que os microsserviços não precisam ser independentes e únicos. Em outras palavras, microsserviços em cadeia (*Chained*) podem ser acionados pelo componente central para processar uma determinada requisição.
- *Shared Data*: Este padrão estipula algumas estratégias de compartilhamento de dados persistentes que precisam extrapolar o domínio interno de um microsserviço. De forma geral, a adoção dessa abordagem deve ser evitada para preservar a ideia de atomicidade e independência dos microsserviços. Entretanto, em alguns contextos como a refatoração de aplicações monolíticas, algumas lógicas de negócio quando distribuídas em diferentes microsserviços podem continuar necessitando acessar a um mesmo banco de dados. Quando adotado, este padrão

remete a mesma ideia de microsserviços em cadeia, pela forte dependência que acaba gerando entre eles.

- *Asynchronous Messaging*: O padrão *Asynchronous Messaging* emerge como uma opção para as arquiteturas RESTful. Nesta arquitetura a comunicação é intermediada por uma fila que repassa as requisições ou chamadas remotas de processos. Desta forma, ao invés de uma cadeia de comunicações bloqueantes entre os microsserviços, passa a existir serviços mais independentes que não tem conhecimento direto um do outro. Tal abordagem facilita a escalabilidade e manutenção dos sistemas, mas em geral adiciona um componente extra para a aplicação: a fila de mensagens (*messaging queue*).

Na prática, um ou mais dos padrões citados podem ser combinados e/ou adaptados para a construção de uma aplicação baseada em microsserviços.

3.3. Desafios para a Adoção de Microsserviços

O uso de microsserviços apresenta diversas vantagens na construção de aplicações, sobretudo em termos de escalabilidade, isolamento, manutenibilidade, entre outros [Esposito et al. 2016]. Entretanto, como qualquer outra decisão de projeto, deve ter a sua utilização avaliada caso a caso, pois a adoção de serviços granulares autônomos e independentes para a construção de aplicações traz vários benefícios, mas também algumas potenciais desvantagens.

Uma primeira possível desvantagem no uso da arquitetura baseada em microsserviços está relacionada ao **desempenho** [Fazio et al. 2016]. Uma das razões é que a troca de mensagens por rede é consideravelmente mais lenta e custosa do que chamadas diretas em memória. Deste modo, sistemas com enfoque máximo em desempenho precisam ter a sua construção ou migração para microsserviços avaliada com cautela.

A adoção de microsserviços também pode trazer impactos negativos na **disponibilidade** e na **confiabilidade** das aplicações [Dragoni et al. 2016]. Embora a natureza independente e autônoma destes possa levar a crer que a disponibilidade do sistema tende a crescer, isto não é sempre verdade. Como o funcionamento do sistema depende de que cada serviço autônomo execute corretamente, a disponibilidade total é um produto do valor individual de cada um destes serviços. Desta forma, o argumento de que serviços menores tendem a acarretar uma densidade menor de erros não é sempre verdade, como demonstrado por [Hatton 1997] e [Compton and Withrow 1990]. Por outro lado, a redução do tamanho dos serviços (e, possivelmente, a sua complexidade) pode ter uma relevante influência na redução da propensão a falhas [El Emam et al. 2002]. Assim, é fundamental planejar os microsserviços da aplicação considerando, além das estratégias de negócio envolvidas, as métricas de confiabilidade do sistema.

Outra potencial desvantagem e que, possivelmente, afasta os programadores e projetistas dos microsserviços é um esperado aumento da **complexidade** do sistema [Hassan and Bahsoon 2016]. Tal incremento da complexidade pode acontecer tanto no âmbito da *codificação*, quanto na *implantação*, *testes* e *logging*.

Além disso, a **programação** de microsserviços exige conhecimentos específicos, como os paradigmas de comunicação inter-processos e a consequente curva de aprendi-

zado de cada uma das técnicas e ferramentas necessárias. Se em um sistema monolítico a comunicação é feita de forma direta entre os seus componentes, em uma arquitetura de microsserviço é necessária uma camada adicional de coordenação (via orquestração ou coreografia [Daniel and Pernici 2007, Daniel and Pernici 2006], por exemplo) que conduza o fluxo da tarefa pelos diversos microsserviços. Embora o grande poder de isolamento possa permitir escolhas livres de linguagens e padrões de projeto para cada microsserviço, também pode levar ao surgimento de equipes muito fragmentadas tecnicamente e exigir um esforço maior de integração e supervisão.

Em adição a tudo isto, a **implantação** de uma aplicação não é uma atividade trivial, mesmo no caso de sistemas monolíticos, e a implantação de inúmeros microsserviços tende a ser bem mais complexa [Villamizar et al. 2015]. Isso torna quase que obrigatório o uso de ferramentas de auxílio e virtualização, como *Docker Engine* [Anderson 2015] e o uso de abordagens de distribuição mais elaboradas e modernas, como por exemplo o *DevOps* [Bass et al. 2015].

Apesar de que os **testes** em microsserviços isolados podem ser mais simples e eficazes, garantir o funcionamento do conjunto como um todo adiciona etapas complementares no processo de validação da aplicação. A principal dificuldade está no fato de que identificar erros e comportamentos indesejados em microsserviços normalmente exige um esforço de análise de vários *logs* e *traces* em diferentes cenários e envolvendo múltiplos serviços independentes [Alshuqayran et al. 2016].

Com relação a **persistência**, a lógica de isolamento e autonomia dos microsserviços recomenda que os dados pertinentes a cada microsserviço sejam gerenciados internamente, incluindo a lógica de armazenamento e **logging**. Embora esse seja um ponto positivo, permitindo assim que cada serviço isolado possa utilizar diferentes bancos de dados, linguagens e padrões para o atendimento das necessidades específicas, ele também traz algumas novas preocupações. Dentre elas, está um aumento na complexidade da gerência na camada de persistência do projeto, uma vez que, consultas anteriormente feitas de forma direta aos dados precisarão ser intermediadas por outros microsserviços.

4. Construindo Microsserviços Reativos

Uma corrente que começa a ganhar força na implementação de serviços é a programação **reativa**, conforme descrito no Manifesto Reativo [Bonér et al. 2014]. Nesta vertente, um **serviço reativo**, ou seja, bem adaptado para as demandas correntes, deve priorizar a **responsividade**, a **resiliência**, a **elasticidade** e ser **dirigida a mensagens** [Atul Shukla 2014].

Por **responsividade** entende-se que o serviço “empurra"(*push*) a resposta para os clientes em tempo real, ao invés do contrário. Isto permite ao cliente estar livre de uma conexão bloqueante e síncrona, devendo então fazer uso de **trocas de mensagens** ao invés de requisições HTTP. Tais características combinadas, em tese, capacitam os serviços a escalarem de forma mais **elástica** em ambientes distribuídos, desde que tais sistemas sejam **resilientes**, ou seja, falhas em um componente não interrompam o serviço como um todo.

A programação reativa é bem convergente com a proposta de microsserviços, uma

vez que ela pode permitir a elasticidade dos seus componentes e fortalecer a sua atomicidade. Uma das principais formas para aplicar a abordagem reativa em microsserviços está na ideia de dividir para conquistar [Boner 2016]. Decompor um sistema monolítico em serviços distintos e isolados é a chave para alcançar a arquitetura reativa em microsserviços. Isolar os componentes (e também suas dependências e falhas) é o maior pré-requisito para resiliência, assim como para elasticidade [Gutierrez 2017]. Por sua vez, a melhor forma de alcançar tal isolamento é através de uma comunicação assíncrona e desacoplada, unindo perfeitamente a ideia de sistemas reativos a natureza dos microsserviços.

4.1. Uma Nova Abordagem para a Organização Interna de Microsserviços

Com inspiração na abordagem reativa e em boas práticas de refatoração de sistemas monolíticos, nesse trabalho é proposta uma nova forma de organização da arquitetura e da comunicação dos componentes internos de um microsserviço. Como apresentado anteriormente, um microsserviço é composto internamente por entidades (micro-componentes). A nossa premissa é que tais micro-componentes também podem ser organizados sob uma ótica de coreografia e/ou orquestração semelhante a do contexto global da aplicação.

A Figura 3 ilustra a adaptação de um sistema monolítico genérico em microsserviços com base em uma técnica arbitrária de fatoração.

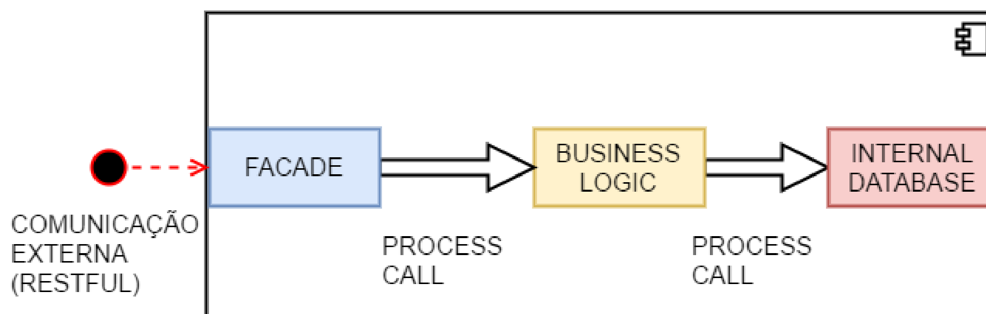


Figura 3. Arquitetura de Microsserviço “Monolítico"

Em linhas gerais, a forma mais intuitiva e simples para a concepção deste microsserviço é através de uma analogia direta com a adotada para a construção de uma aplicação monolítica. Neste sentido, toda a funcionalidade interna do microsserviço seria provida através de uma interface (*facade*) que encapsula e distribui uma lógica de negócio (*business logic*) e faz acesso a uma série de dados e tabelas (*internal database*) através de comunicação direta entre cada componente.

Para garantir as propriedades de um sistema reativo no contexto isolado de cada microsserviço que compõe a aplicação, a abordagem proposta neste trabalho envolve a utilização de uma coreografia interna, baseada em filas de mensagens, para desacoplar as ligações diretas de cada micro-componente do microsserviço. Nesta nova modelagem, como apresentada na Figura 4, um *middleware* orientado a mensagem (MOM) [Curry 2004] é adicionado para servir de canal de comunicação entre os micro-componentes, os quais baseiam a sua operação em mensagens, filas e rótulos do protocolo AMQP (do inglês *Advanced Message Queuing Protocol*) [Videla and Williams 2012].

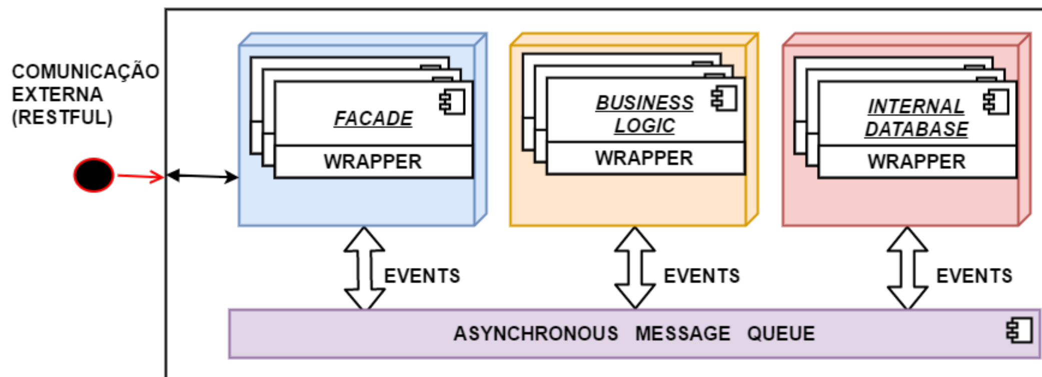


Figura 4. Arquitetura de Microsserviço Reativo

Uma vez que toda troca de mensagens passa por um canal separado, tais micro-componentes passam a agir como **workers**, que utilizam blocos de código padronizados (“wrappers") para facilitar a utilização das filas[1].

4.2. Algumas Implicações

A arquitetura proposta permite que o microsserviço tenha a possibilidade de expansão em termos de **escalabilidade lateral** [Namiot and Sneps-Sneppe 2014]. Em outras palavras, a escalabilidade do sistema pode ser feita também dentro do escopo interno de cada microsserviço, dividindo a demanda em um conjunto elástico de *workers*. Combinada com a replicação e balanceamento de carga dos microsserviços inteiros, esta abordagem permite que gargalos específicos do sistema possam ser tratados em uma granularidade menor que a de um microsserviço.

Além disso, a adição de uma coreografia interna baseada em filas consiste num dos entraves centrais quanto a adoção de microsserviços: a **curva de aprendizado** na migração e incorporação de trechos de sistemas monolíticos aos novos microsserviços. Na abordagem proposta, o código inteiro do trecho desejado pode ser reaproveitado sem grandes mudanças em termos de codificação. Neste caso, os micro-componentes candidatos devem ser identificados e agrupados em *workers* que processarão mensagens da filas de acordo com a coreografia definida. O código do *worker* pode ser encapsulado em um *wrapper* na linguagem original e integrado ao resto do microsserviço através do uso do componente cliente específico do *middleware* de filas de mensagens adotado.

Como reflexo da escalabilidade lateral interna, um dos possíveis benefícios da adoção desta abordagem pode ser um eventual incremento na performance do microsserviço como um todo quando executado em ambientes *multithread*. Se, por um lado, a troca de mensagens diretas possa ser um pouco mais rápida do que com a intermediação de filas de mensagens, uma abordagem de execução concorrente e independente traz ganhos em escalabilidade e resiliência. Isto se dá pelo fato que os diversos conjuntos de *workers* utilizam melhor os recursos computacionais e podem continuar operando mesmo

[1]De forma semelhante, outras estratégias de troca de mensagem (como a coreografia de fluxo de eventos) podem ser adotadas para tal contexto, mas dependeriam de uma estruturação arquitetural mais elaborada e de maior adaptação do código já existente.

que um deles falhe. Uma vez restaurado, o *pool* pode retomar os pedidos pendentes nas suas respectivas filas de entrada e reintegrar-se ao fluxo de execução com facilidade.

Um efeito colateral é que o microsserviço pode se tornar um pouco mais complexo, uma vez que a sua comunicação interna também passa a ser dirigida por mensagens. Entretanto, acredita-se que a adoção de um padrão arquitetural comum, encapsulado em um *framework* e bem documentado, permite a inibição de tal efeito.

5. Protocolo do Estudo

Para avaliação do efetividade da abordagem proposta, foram realizados dois estudos de caso de refatoração de sistemas monolíticos em microsserviços reativos. A descrição do contexto dos estudos de caso realizados, incluindo os ambientes de testes, os projetos de experimentos e as métricas de interesse adotadas, são descritas nas seções a seguir.

5.1. Estudo de Caso #1: Tradutor Automático PORTUGUÊS-LIBRAS

Os estudos de casos realizados no escopo deste trabalho foram baseados em um sistema real e em produção: o serviço de tradução automática de Português Brasileiro para a Língua Brasileira de Sinais (LIBRAS), chamado **Suíte VLibras** [Falcao et al. 2014].

A **Suíte VLibras** é o resultado de uma parceria entre o Ministério de Planejamento, Desenvolvimento e Gestão (MP), através da Secretaria de Tecnologia da Informação (STI), e a Universidade Federal da Paraíba (UFPB), junto ao Laboratório de Aplicações de Vídeo Digital (LAVID), e consiste de um conjunto de ferramentas para tradução automática de Português Brasileiro (texto, áudio e vídeo) para a Linguagem Brasileira de Sinais (LIBRAS), tornando computadores, dispositivos móveis e plataformas Web acessíveis para os surdos. Atualmente, o VLIBRAS é utilizado em vários sites públicos e privados, dentre eles os principais do Governo Brasileiro (brasil.gov.br), Câmara dos Deputados (camara.leg.br) e Senado Federal (senado.leg.br)[2].

O primeiro estudo de caso teve como base uma iteração localizada de refatoração do código da **Suíte VLibras** usando a abordagem proposta. Nesta primeira intervenção, o objetivo foi extrair o componente principal da Suíte VLibras, o **Tradutor**, e transformá-lo em um microsserviço reativo.

A subseções seguintes detalham as modificações, a ambiente de testes e o detalhamento dos experimentos realizados.

5.1.1. Transformando o Tradutor VLibras em um Microsserviço Reativo

O diagrama contido na Figura 5 ilustra a arquitetura de um novo microsserviço criado a partir da refatoração do código do componente **Tradutor** da Suíte VLibras. Modulado em um contêiner Docker [Anderson 2015], o novo microsserviço oferece uma API *Restful*, através da qual é feita a sua ativação pelos outros componentes da Suíte. Através

[2]Mais informações podem ser obtidas em http://www.vlibras.gov.br.

de uma requisição GET é possível submeter um texto para tradução automática de português para LIBRAS, o qual, posteriormente, será sinalizado visualmente por um avatar através das diversas interfaces da Suíte VLibras.

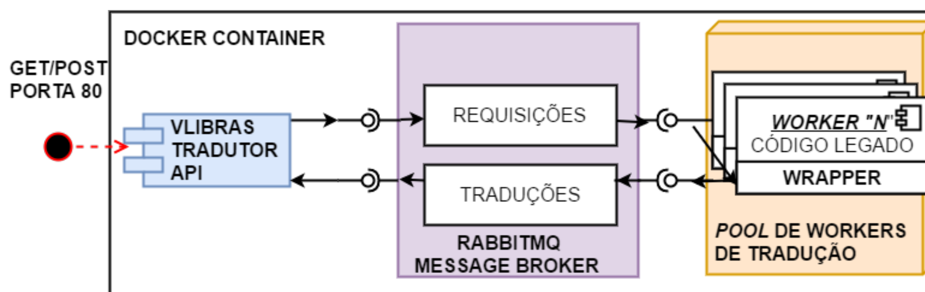


Figura 5. Microsserviço de Tradução Automática

Após receber uma requisição, a API enfileira a solicitação de tradução em uma fila de mensagens específica (**Requisições**) gerenciada pelo RabbitMQ [Videla and Williams 2012]. Por sua vez, um *pool* de *workers* de tradução fazem a conversão efetiva de português para LIBRAS, consomem requisições da fila **Requisições**, processam-nas e encaminham as respostas para a fila de saída, chamada **Traduções**.

Implementar um *worker* passa a ser uma tarefa mais simples nesta abordagem. Neste cenário, uma atividade de tradução demanda duas rotinas: uma para **extração** dos dados e outra para efetivar a **tradução** semântica. No bloco monolítico do VLibras original tais rotinas de negócio estavam altamente acopladas.

Devido a tais dependências, estes “micro-componentes"dificilmente poderiam se tornar microsserviços isolados. Em sua essência, tais rotinas fazem parte do mesmo domínio (Tradução) no serviço e não servem a outro aspecto do sistema que não seja traduzir uma sentença.

Entretanto, o processamento síncrono e sequencial das traduções no bloco monolítico afetava a eficiência do VLibras como um todo. Envolvendo estes trechos de codigo em um consumidor das filas de mensagens em *Python* foi possível possível desacoplar estas funções sem que fosse necessário realizar um grande *refactoring* na implementação atual do VLibras.

Assim, a nova organização possibilitou o atendimento concorrente e simultâneo de várias solicitações de tradução, sendo possível customizar o número de processos de tradução, ou seja, o tamanho do *pool* de *workers* de acordo com a demanda e a capacidade do *hardware*.

Essa capacidade foi explorada nos nossos testes que consideraram vários tamanhos para o *pool* de *workers* de tradução.

5.1.2. Ambiente de Testes

Para execução dos testes do primeiro estudo de caso foram utilizados dois computadores distintos: um deles foi usado como cliente do serviço de tradução e o outro como

servidor (*host*) do serviço de tradução. O objetivo de tal configuração foi isolar bem os dois ambientes e poder avaliar o impacto da aplicação da abordagem proposta no desempenho da Suíte VLibras, sobretudo pelo uso de um *pool* de *workers* e pela conteinerização do microsserviço de tradução.

A Tabela 2 apresenta as configurações de *hardware* em cada uma das máquinas utilizadas nos testes. A máquina com maior poder de processamento e memória foi alocada como servidor, enquanto a outra, com capacidade de processamento menor, foi usada como cliente para gerar as requisições e computar os tempos associados com cada tradução.

Tabela 2. Ambiente Computacional Usado nos Testes

	Processador	Threads	Mem	HD
Servidor	Intel i7-4790K/4GHz	8	16 GB	1TB
Cliente	Intel i7-4510U/2GHz	4	8 GB	1TB

Tanto a nova versão quanto a versão original da Suíte VLibras compartilharam o mesmo equipamento usado como servidor, estando ativas em rodadas distintas de execução separadas dos testes. O cliente de tradução usado foi o mesmo para as duas versões do tradutor.

Para cada execução dos testes, tanto a máquina cliente quanto o servidor foram reiniciados ao estado inicial com o objetivo de garantir a uniformidade e evitar que resíduos da execução anterior contaminassem os resultados. Cada configuração particular de teste foi repetida, pelo menos, 10 vezes e as médias obtidas foram estimadas com intervalos de confiança estabelecidos no nível de confiança de 95%.

5.1.3. Projeto de Experimentos

O principal foco do primeiro estudo de caso foi a análise do eventual ganho em desempenho que a abordagem proposta pode proporcionar, mesmo quando considerado apenas a extração de um único componente de uma aplicação monolítica.

Neste sentido, o projeto dos testes considerou as seguintes variáveis de controle: a) tamanho do *pool* de tradutores; b) tamanho do texto para tradução; c) quantidade de clientes simultâneos e d) quantidade de solicitações. As Tabelas 3, 4 e 5 ilustram os valores adotados para cada uma delas.

Tabela 3. Configuração do *pool* de *workers*

Pool	Tamanho
A	1 *worker* de tradução
B	20 *workers* de tradução

Tabela 4. Tipos de Entrada para Tradução

Descrição	Qtd Caracteres
Solicitação Média	1055
Solicitação Grande	1881

A variável dependente de interesse foi a *vazão média por minuto* (**VMPM**), representada pela taxa de solicitações de tradução efetivamente atendidas pelo servidor a cada

Tabela 5. Tipos de Demanda

Tipos	Cli Simul	Requisições P/Cliente	Duração
Carga Comum	10	10	-
Sobrecarga	50	10	-
Carga Contínua	20	-	15 min

minuto.

Para os testes, os clientes foram configurados, usando o **JMeter** [Halili 2008], para serem ativados simultaneamente na quantidade indicada para cada tipo de carga e realizarem um número pré-determinado de solicitações de forma sequencial - uma nova transação só é feita quando a anterior é respondida (ou um *timeout* ocorre). Tanto para a **Carga Comum** quanto para a **Sobrecarga**, o número de clientes simultâneos foi escolhido a partir de testes preliminares de forma a caracterizar um funcionamento intenso, mas suportável pelo *hardware* utilizado, no primeiro caso, e um funcionamento de estresse, suscetível à falhas, no segundo caso.

Uma carga específica, nomeada de **Carga Contínua**, foi usada para avaliar a vazão padrão do sistema em condições normais. Ela difere da **Carga Comum** apenas na duração, que no seu caso é limitada por um período de tempo (15 minutos) e não por uma quantidade fixa de solicitações.

5.2. Estudo de Caso #2: Geração de Vídeo Vlibras

O estudo de caso anterior considerou uma **orquestração simples**, em outras palavras, provisionado por um microsserviço isolado. Porém, em geral a motivação por trás da adoção de MS está em refatorar a granularidade do sistema afim de melhorar a elasticidade da solução. Neste caso, testes em uma **orquestração complexa** com vários MS interagindo reproduzem melhor o comportamento de um ambiente em produção.

Continuando o refatoramento da Suíte Vlibras, a segunda iteração desta atividade focou em remodelar a funcionalidade de geração de vídeos com legendas em Libras. Devido a natureza mais complexa do processo de geração de vídeos legendados em Libras, outros MS precisaram ser agregados para a orquestração “macro” e uma nova versão da Suíte VLibras foi produzida.

Esta seção detalha os testes feitos sobre o novo gerador de vídeos baseado em microsserviços reativos para comparação como o serviço monolítico original do VLibras.

Com o objetivo de avaliar o impacto da abordagem proposta nos microsserviços gerados na decomposição da geração de vídeo do VLibras, novos experimentos foram planejados para avaliar também as outras dimensões da abordagem reativa além do **Desempenho**: a **Elasticidade** e a **Resiliência**.

5.2.1. Transformando o Gerador de Vídeos VLibras em um Microsserviço Reativo com Armazenamento

A arquitetura do microsserviço anteriormente apresentado especifica uma orquestração simples, ou seja, uma orquestração de apenas um tipo de microsserviço. Por outro lado o processo para geração de vídeos legendados (com uma janela de legenda animada por um Avatar 3D) da Suíte VLibras possui quatro rotinas principais: extração de legendas, tradução de sentenças, geração de vídeo em Libras (animação) e a mixagem com o vídeo original. O diagrama da Figura 6 apresenta um exemplo de arquitetura de microsserviços reativos para cenários mais complexos de orquestração, como o de geração de vídeos legendados do VLibras.

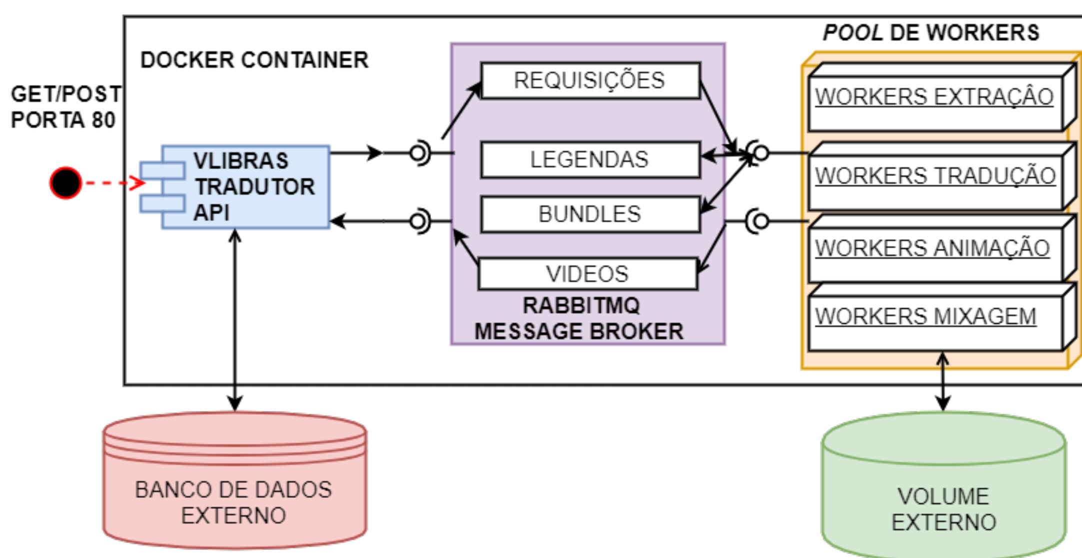


Figura 6. Microsserviço de Geração de Vídeos

Semelhante ao serviço Tradutor, o **Gerador de Vídeo** da Suíte Vlibras foi refatorado com base no código legado, utilizando um contêiner Docker. Internamente, este contêiner principal tem uma arquitetura semelhante ao serviço de tradução, com uma API *RESTful* que se comunica através de uma fila de mensagens acoplada a um *pool* de *workers* do Vlibras.

A primeira grande diferença dos dois serviços está na natureza destes *workers*. Enquanto no serviço de tradução existe apenas um tipo de *worker*, no serviço de geração de vídeo existem quatro tipos destintos, sendo estes: **extração** de legendas, **tradução** de legendas para glosa, **animação** das legendas e **mixagem** dos vídeos. Uma vez que estes componentes trabalham sobre um mesmo conjunto de dados e dicionário internos, não seria eficiente refatorar cada uma dessas operações em um microsserviço separado.

A segunda diferença entre as arquiteturas reativas está ligada ao armazenamento do estado das operações. O Tradutor Vlibras é um microsserviço *stateless*, ou seja, que não guarda nenhum estado sobre as traduções solicitadas. Uma vez processado o texto de entrada, o texto de resposta (em glosa) é encaminhado para o cliente. Novas solicitações do mesmo cliente serão tratados sem memória das transações anteriores.

O mesmo não é valido para operações de geração de vídeo. Além dos quatro processos envolverem a extração, tradução animação e mixagem de múltiplas sentenças para uma mesma solicitação, após a geração do vídeo legendado em Libras o mesmo precisa ser armazenado por um certo tempo até ser recuperado pelo cliente, dado ao alto custo de processamento e tempo para a sua produção após a solicitação e também ao eventual tempo gasto em *upload* e *download* pelo cliente. Neste sentido, foi necessário adicionar dois componentes de apoio.

Inicialmente, foi introduzida uma camada de persistência, a qual é ativado diretamente pelo *backend* da API, para o controle da orquestração do processo de geração de vídeo. A camada de pwersistência, implementada em um Banco de dados guarda os estados das operações, informações dos clientes e *path* dos vídeos originais e dos vídeos mixados. Os arquivos em si dos vídeos originais e gerados são armazenados por um **Volume Externo**. o volume externo age como uma memória compartilhada e controla acessos e alocações a disco por parte dos *workers* e contêineres de geração de vídeo.

5.2.2. Ambiente de Testes

Em um cenário real, um MS é em geral distribuído em provedores de nuvem (públicos ou privados) que armazenam e provisionam outros serviços distintos. Um ambiente isolado em um único servidor, como nos testes anteriores, pode não representar bem o comportamento esperado do serviço em produção.

Desta forma, para os testes mais complexos de elasticidade, um provedor de *cloud computing* foi escolhido para hospedar o serviço durante a execução dos testes. Uma vez que o microsserviço foi implementado utilizando a tecnologia Docker, foi necessário que o ambiente a ser escolhido provesse um *framework* com suporte a este tipo de contêiner e a opções de escalabilidade automática.

O provedor escolhido para os testes do segundo estudo de caso foi o **Google Cloud Platform**. A plataforma da Google provou-se a mais acessível considerando os requisitos necessários. Como ela disponibiliza um ferramental nativo com suporte a *Kubernetes* [Vohra 2017], tanto o uso do *framework* quanto do próprio serviço de nuvem apresentaram uma curva de aprendizagem pequena, além de um incentivo de gratuidade em uma quantidade limitada de horas de processamento. Além de um *cluster* de contêiners hospedado na nuvem, também foi utilizado um servidor físico (conforme descrito na Tabela 2) da infraestrutura da UFPB para a submissão dos vídeos e legendas originais para tradução.

A Tabela 6 apresenta as configurações utilizadas para configurar o Kubernetes Cluster. As máquinas virtuais alocadas foram do tipo "n1-standard-2"[3], com sistema operacional Ubuntu, 2 CPUs virtuais e 200 gigas de armazenamento. A cada execução dos testes o cluster alocado com os contêineres (também chamados de *Pods*) é completamente removido, para que a próxima execução parta do mesmo estado inicial, ou o mais próximo disto.

[3]Disponível em : https://cloud.google.com/compute/pricing?hl=pt-br

Tabela 6. Configurações do Cluster Kubernetes

Tipo	n1-standard-2
Região	us-central
Sistema Operacional	Ubuntu 17.04
CPUs	2 vCPUs
Memória	7,5 GB
Disco de Inicialização	200 GB
Escalonamento(VMs)	2 $\sim$ 4 VMs

Durante a execução, o *cluster* escalona suas máquinas virtuais de acordo com o consumo de processamento e *Pods* disponíveis, variando de no mínimo 2 máquinas virtuais para até 4, todas com as mesmas configurações. No âmbito dos contêineres, as operações de redirecionamento e balanceamento de carga foram gerenciadas automaticamente pelo orquestrador Kubernetes. Neste, a variação foi de no mínimo 1 a até 4 *Pods*. Já para o processo de escalonamento, o procedimento de “*scale-up*"ou “*scale-down*"foi configurado de maneira semi-automática.

5.2.3. Projeto de Experimentos

De forma semelhante ao que foi feito para a versão modificada do VLibras, a versão original, monolítica, também foi encapsulada em um contêiner *Docker*, tendo sua arquitetura também dívida em serviços separados: Um contêiner principal contendo o serviço **monolítico do Vlibras**, um contêiner contendo um serviço de **armazenamento de dados** (MongoDB) e um contêiner para o serviço de **armazenamento e sincronização de arquivos em rede**. Este MS sem refatoramento do código em micro-componentes, uso de filas e compilado de forma monolítica foi chamado de **microsserviço monolítico** e foi utilizado como referência na comparação com a arquitetura reativa proposta.

Além da adaptação feita sobre a arquitetura monolítica para migrá-la para um contexto de contêiner, também foi preciso implementar um novo controlador de testes especialmente projetado para orquestrações complexas. Este novo cliente de testes foi desenvolvido na linguagem de programação interpretada Python versão 2.7 e implementou todo o procedimento de envio e recebimento de vídeos, de forma customizável para a necessidade de cada cenário de teste a ser executado.

Vale ressaltar, entretanto, que devido a restrições físicas de rede e *hardware*, nem todos os vídeos gerados como resposta foram integralmente baixados. Para cenários de testes com um grande volume de vídeos, foi implementada uma operação de confirmação remota do status do vídeo, feita entre a API do microsserviço do VLibras e o cliente remoto.A duração da janela de envios dos vídeos foi definida em **1 hora**, e o tempo máximo de espera por uma resposta positiva também é igual a duração da janela de envios.

Desta forma, a duração máxima de cada repetição é de 2 horas de atividade e 40 minutos de *set-up* dos microsserviços na infraestrutura. Todas as medições foram feitas em intervalos de 1 minuto e armazenadas no cliente junto com os vídeos processados.

Segue abaixo um exemplo de um log gerado por este cliente.

6. Resultados Obtidos

6.1. Resultados: Estudo de Caso #1

A Tabela 7 apresenta os resultados para os experimentos sobre a vazão (VMPM) padrão obtida por cada configuração de servidor. As duas últimas colunas da tabela apresentam os valores na razão de “sucessos por minuto"na forma da média da vazão e também do intervalo de confiança apurado.

Tabela 7. VMPM Após Carga Contínua de 15 minutos

Tradutor	Entrada	Vazão Média	Intervalo de Confiança da Vazão
VLibras	Média	15,02	(14,92, 15,12)
	Longa	15,01	(14,97, 15,05)
1 Worker	Média	15,12	(15,09, 15,15)
	Longa	14,79	(14,78, 14,81)
20 *workers*	Média	33,09	(32,98, 33,19)
	Longa	33,06	(32,72, 33,40)

Os testes utilizando um *pool* com apenas 1 (um) *worker* na nova versão tiveram a finalidade principal de permitir uma comparação mais direta com a versão original do **VLibras**, sobretudo com relação ao efeito da intermediação do **RabittMQ** no desempenho. Como pode ser visto na Tabela 7, o custo do *overhead* da fatoração e da aplicação da abordagem proposta não trouxe um impacto significativo na vazão do **VLibras**. Ou seja, quando usando apenas um *worker*, foi observada apenas uma pequena perda de $1,5\%$ para entradas de tamanho grande e uma equivalência de vazão para entradas de tamanho médio.

Entretanto, quando aumenta-se o *pool* para 20 (vinte) *workers*, ficou bem evidente que a abordagem de microsserviços reativos pode trazer ganhos em desempenho e escalabilidade. A VMPM proporcionada pela nova versão foi mais que o dobro da vazão original do **VLibras** tanto para entradas médias quanto para entradas longas. Tal melhoria foi obtida usando não apenas o mesmo *hardware* mas também o mesmo código do componente tradutor.

Para avaliar o atendimento de outra propriedade que a abordagem reativa poderia trazer, a resiliência, alguns testes de carga foram feitos submetendo os servidores de tradução à uma simulação de pico de demanda. Os resultados estão detalhados nas Tabelas 8 e 9, as quais apresentam os resultados para cargas com 10 clientes e 50 clientes simultâneos, respectivamente.

O primeiro cenário, com apenas 10 clientes simultâneos, representou o limiar da capacidade de atendimento da versão original do **VLibras**, o qual já apresentou uma pequena taxa de erros, perdendo solicitações de tradução. O mesmo não aconteceu com a nova versão, tanto com 1 ou com 20 *workers*. Isso pode ser explicado pelo tratamento mais leve dado no atendimento inicial da solicitação, restrito à sua recepção e enfileiramento na fila de mensagens.

Por outro lado, o paralelismo proporcionado pelos 20 *workers* se refletiu na VMPM obtida nessa configuração, a qual foi cinco vezes maior que a obtida pela versão original do **VLibras**.

Tabela 8. Resultados para 10 Clientes Simultâneos

Tradutor	Entrada	Erro (%)	Vazão (p/min)
VLibras	Média	(0,64, 1,55)	(11,92, 14,12)
	Longa	(0,00, 0,59)	(13,69, 15,34)
1 Worker	Média	0	(15,05, 15,13)
	Longa	0	(14,69, 14,75)
20 *workers*	Média	0	(51,09, 51,29)
	Longa	0	(50,43, 50,69)

No cenário efetivo de sobrecarga, representado por 50 clientes solicitando traduções simultâneas, a capacidade do sistema foi propositadamente exaurida e todas as configurações do servidor de tradução apresentaram erros, por descarte ou *timeout*, no atendimento das solicitações. Novamente, o efeito do pico foi mais sentido pela versão original do **VLibras**, a qual só conseguiu processar corretamente cerca de 7% da demanda que recebeu.

Tabela 9. Resultados para 50 Clientes Simultâneos

Tradutor	Entrada	Erro (%)	Vazão (p/min)
VLibras	Média	(93,2%, 93,8%)	(15,20, 17,16)
	Longa	(93,2%, 93,7%)	(16,90, 18,89)
1 Worker	Média	(67,5%, 74,6%)	(24,87, 25,86)
	Longa	(70,5%, 75,1%)	(24,34, 25,43)
20 *workers*	Média	(32,7%, 34,4%)	(36,46, 37,85)
	Longa	(32,3%, 35,2%)	(26,61, 39,61)

Uma das possíveis razões para os resultados obtidos é o melhor aproveitamento do *hardware*. Neste sentido, os piores resultados da arquitetura proposta em cada configuração de *pool* (1 e 20 *workers*) foi comparado com o melhor resultado da versão original do *VLibras*. O parâmetro para esta comparação foi a utilização média dos recursos de CPU e Memória.

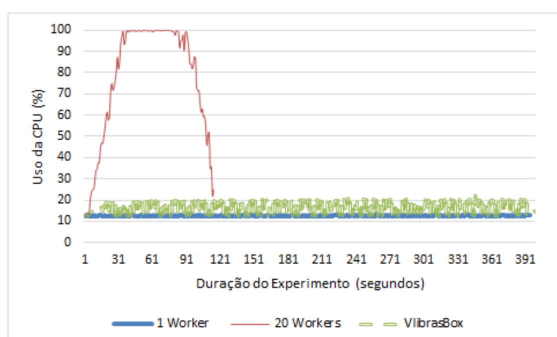


Figura 7. Consumo de CPU com 10 Clientes Simultâneos

Figura 8. Consumo de CPU com 50 Clientes Simultâneos

As Figuras 7 e 8 apresentam a curva percentual de uso do *hardware* no eixo temporal do experimento. Como pode ser visto, a configuração com apenas um *worker* de tradução (linha azul) obteve uma taxa de uso de ambos os componentes de *hardware* mais constante e ligeiramente inferior, mesmo obtendo resultados semelhantes aos do **VLibras** (linha verde).

A configuração com vinte *workers*, por sua vez, foi o único cenário a alcançar o uso máximo da CPU nos dois cenários de muitos clientes simultâneos (linha vermelha). Além disso, essa configuração demonstrou uma capacidade de recuperação plena após as falhas provocadas pelo excesso de demanda, retomando o nível máximo de utilização da CPU com rapidez.

6.2. Resultados: Estudo de Caso #2

No segundo estudo de caso, a orquestração interna dos serviços foi ajustada para os experimentos. O número de *workers* testados em cada um dos cenários foi ajustado para variar entre 1, 4 e 8, de acordo com a Tabela 10. Estes valores foram estipulados em decorrência dos testes exploratórios prévios, os quais, para comparação com o modelo padrão, também foram realizados com o "microsserviço" adaptado da Suíte Vlibras original.

Tabela 10. Distribuição das Requisições para Testes de Elasticidade

Configurações	*Pool* de *workers*
A	1 *Worker*
B	4 *workers*
C	8 *workers*
D	Vlibras Core

Para os testes definitivos foi utilizado apenas um tipo de entrada ("vídeo (formato mp4) + legenda'). Uma vez demonstrado nos testes exploratórios que o tamanho da entrada não teve impacto relevante para qualquer uma das abordagens, foram descartadas cenários variando este fator. Para todos os cenários tratados, o vídeo e a legenda se referem a uma contagem regressiva de 10 segundos.

6.2.1. Avaliação da Responsividade

Assim como a configuração interna dos *workers* e o tipo de entradas, as taxas de entrada de requisições aos serviços foram adaptadas para o novo ambiente.

Tabela 11. Distribuição das Requisições para Testes de Desempenho

Cenário	Intervalo entre Requisições	Taxa de Chegada
Carga Leve	12 minutos	5 vídeos/hora
Carga Baixa	8 minutos	7 vídeos/hora
Carga Normal	4 minutos	15 vídeos/hora
Carga Moderada	2 minutos	30 vídeos/hora
Sobrecarga	1 minuto	60 vídeos/hora
Stress	30 segundos	120 vídeos/hora

Uma vez que, por sua natureza, as demandas por geração de vídeo são mais complexas, foram adicionados quatro outros cenários além da “Sobrecarga"e “Carga Comum"dos experimentos realizados no primeiro estudo de caso. A Tabela 11 enquadra os novos cenários planejados e os cenários antigos adaptados para o contexto do microsserviço de geração de vídeo.

No cenário de “Stress"é esperado do serviço uma grande taxa de rejeiçào das requisições do cliente. Já para a “Carga Leve”, foi estipulada uma taxa de entrada que resulte em nenhuma rejeição, dado o pouco volume dos dados e grande espaço de tempo entre as requisições. Os cenários de carga “Baixa” e “Moderada” representam picos abaixo ou acima do comportamento padrão.

Como já visto anteriormente, sistemas reativos tem grande foco na resposta ao usuário. Neste contexto um sistema responsivo (**Responsive**), referente a métricas de performance e capacidade de responder de forma ágil *“on demand”* . Os resultados observados com os cenários de teste sobre desempenho da arquitetura proposta em um ambiente de nuvem reforçam os dados dos resultados preliminares como se segue.

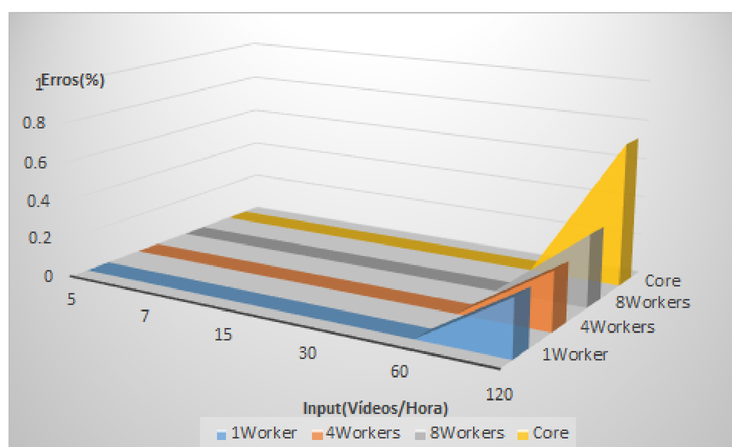


Figura 9. Erro Médio(%) - Desempenho

O gráfico de área apresentado na Figura 9 representa a comparação entre a média de erros de cada orquestração interna dos serviços em cada um dos cenários planejados em vídeos por hora (v/h). De imediato é notável o ganho de performance com a adoção da arquitetura distribuída com Kubernetes. Para os testes com a "carga leve" (5 v/h), "baixa" (7 v/h), "normal" (15 v/h), "moderada" (30 v/h) e "sobrecarga" (60 v/h), nenhuma das execuções ocasionou no impedimento do provisionamento do serviço de forma integral. Em outras palavras, duplicar a taxa de chegada nestas configurações não acarretou em falhas na computação dos vídeos e todas as requisições foram entregues.

Por outro lado, para uma carga de "stress" (120 v/h), foi possível verificar o surgimento de erros em todas as configurações do serviço. Assim como nos testes preliminares, os piores resultados encontrados foram para configuração tradicional do "Vlibras Core". Para este microsserviço "monolítico", a porcentagem de erros médios chegou à **72.8%** com o desvio padrão de 23.8%. Já para as configurações de microsserviços que adotaram a arquitetura proposta, a média de erros obtidas foram de menos da metade. O pior caso dos microsserviço reativos para estes testes foi a orquestração de 8 micro-componentes internos. Dado o uso excessivo dos recursos, tal distribuição não se mostrou a mais efetiva tendo o percentual médio de erros de **36.7%** (± 17.1).

Já para as configurações com 1 e 4 *workers*, os percentuais médios de erro foram de **28.8%** (± 09.2) e **29.8%** (± 10.5) respectivamente. Para as três configurações de microsserviços reativos, uma vez que seus limites são interpostos entre si, não se pode aferir estatisticamente uma diferença entre eles. Contudo, quando comparando estes casos com a versão do gerador de vídeo monolítico, fica comprovado o incremento na efetividade da arquitetura proposta.

Este percentual de erros médio, também pode ser visualizado em um número médio de requisições concluídas ao final do experimento. Ao número dos vídeos concluídos sobre a duração total do experimento têm-se a vazão geral do sistema ao final de 2 horas. A Figura 10 ilustra estes dados normalizados para a unidade de vídeos por hora.

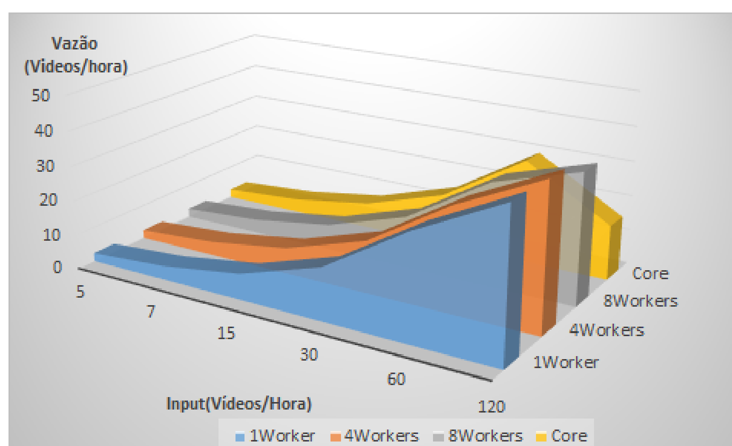


Figura 10. Vazão Média - Desempenho

De forma semelhante ao percentual de erro, todas as arquiteturas propostas tiveram resultados ideais para os cenários de 5, 7, 15, 30 e 60 vídeos por hora. Isto reflete

a ociosidade do sistema quando submetido a estas cargas, conseguindo assim responder todas requisições de forma quase que imediata. A vazão destes cenários foram 0.042, 0.067, 0.125 e 0.25 vídeos por minuto ou **2.5** , **4**, **7.5**, **15** e **30** vídeos por hora.

A grande diferença entre as propostas se deu quando sob stress. Para este cenário, a refatoração do Vlibras com a abordagem monolítica para microsserviços conclui, em média, apenas **15.72** (± 13.68) vídeos por hora ou aproximadamente **1 vídeo a cada 4 minutos**. Por outro lado, o pior desempenho dos microsserviços reativos, obtido pela orquestração de 8 *workers* alcançou a entrega de **37.2** ($\pm 10,08$) vídeos por hora. Estes resultados foram ainda melhores para as orquestrações de 1 e 4 *workers*, com médias de **42.18** (± 5.7) e **41.58** (± 5.76) vídeos por hora. Desta forma, para cenários de cargas massivas, a solução reativa quase triplicou a responsividade do sistema, alcançando cerca de **2,4 vídeos a cada 4 minutos**.

6.2.2. Avaliação da Elasticidade

Para avaliar a elasticidade do sistema é preciso avaliar, toda a capacidade de expansão e não apenas a sua configuração minima. Os cenários de testes de escalabilidade foram projetados levando em conta especificadamente o número de instâncias do microsserviço. Apesar de suporte nativo a um processo automático de escalonamento por parte do *Kubernets*, na prática este tipo de atividade acaba sendo seguida de uma revisão manual, em especial no encerramento de instâncias com carga ociosa.

Desta forma, para simplificar e automatizar a rotina de testes, os cenários planejados para estes tipos de testes tiveram um número de instâncias fixado em 4 *Pods*. Este acréscimo no número de instâncias foi feito apenas no MS "Vlibras". Os microsserviços de armazenamento de dados e arquivos, por não serem o objeto de estudo em questão, continuaram com a quantidade mínima de 1 *Pod*.

Uma vez que o ponto principal está em avaliar as métricas de desempenho em relação a um ambiente escalonado, a variação dos diferentes cenários passa a ser a quantidade de vídeos submetidos em paralelo. O intervalo da submissão destas requisições foi fixado em **1 minuto**.

Tabela 12. Distribuição das Requisições para Testes de Elasticidade

Cenário	Submissões Paralelas por Requisição	Taxa de Chegada (vídeo/hora)
Cliente Único	1	60 vídeos/hora
Carga Baixa	2	120 vídeos/hora
Carga Moderada	4	240 vídeos/hora
Stress	8	480 vídeos/hora

Na Tabela 12 é possível ver quais cenários foram planejados. Com base nos resultados experimentos de responsividade, quatro diferentes cenários foram projetados. Como comparativo, o cenário de "cliente único", envia um único vídeo para ser processado a cada intervalo das requisições. Assim sendo, este caso corresponde ao mesmo

input feito na “Sobrecarga” no contexto de responsividade e serviu para comparar a influência que os múltiplos *Pods* acarretaram ao sistema.

Seguindo então a mesma progressão anterior, a “Carga Baixa” e “Carga Moderada” têm o dobro e o quadruplo de vídeos submetidos por minuto. Estes casos tem como objetivo extrapolar os limites do stress suportado por um único *Pod*, mas têm em teoria o cargas aceitáveis para toda infraestrutura. O último dos cenários em questão se refere ao estado crítico do sistema. Para caso de “Stress”, foi planejada uma taxa de chegadas que provoque rejeição do sistema, mesmo considerando as múltiplas instâncias dos *Pods*.

Uma vez comprovado o impacto que a arquitetura reativa de microsserviços tem em um contexto isolado, foram planejados e executados experimentos para avaliar este mesmo impacto agora no contexto elástico. "Elasticidade"se refere ao potencial do serviço crescer, para atender uma maior demanda e/ou reduzir para economizar gastos.

Utilizando os resultados anteriores (Desempenho) como piso do processo de escalonamento de recursos é possível comparar o menor estado de recursos alocados do sistema, com a maior composição do serviço em uma orquestração complexa no ambiente apresentado. Desta forma, contrapondo os resultados logrados previamente com as medidas apresentadas a seguir, foi possível inferir sobre o impacto da orquestração interna para elasticidade de um microsserviço.

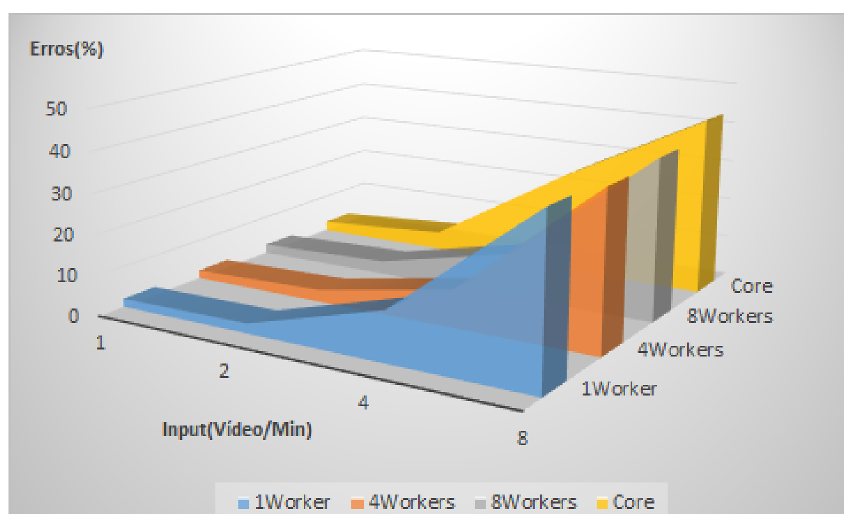


Figura 11. Tempo Médio de Resposta - Elasticidade

A Figura 11 explica a variação do tempo até a entrega do vídeo (**Tempo Médio de Resposta**). Reafirmando os resultados já alcançados, o tempo médio de resposta com o ambiente em 4 *Pods* também exibiu resultados favoráveis aos microsserviços reativos. Com o tempo médio de **1 vídeo a cada 2 minutos**, as configurações do *pool* de *workers* com 1 e 4 se mostraram novamente superiores para taxa de chegada de 1 vídeo por minuto. Neste mesmo cenário, o desempenho do Core Vlibras ficou em **1 vídeo a cada 2,5** minutos e o *pool* de 8 *workers* obteve resultados intermediários com **1 vídeo a cada 2,25 minutos**.

Todavia, uma vez que o desvio padrão destes intervalos sobrepõem-se, não é pos-

sível afirmar com certeza matemática uma diferença neste caso. O mesmo já não pode ser dito quando submetido a uma carga baixa (2 vídeos/minuto). Neste caso, o a utilização de apenas 1 *worker* obteve os melhores resultados, sendo capaz de processar **1 vídeo a cada 2,449** ($\pm 0,73$) **minutos**, enquanto para os set de 4 e 8 *workers*, o tempo médio de resposta ficou em **2,99** ($\pm 1,10$) e **3,30** ($\pm 1,20$) minutos respectivamente. Mas a maior diferença obtida foi no microsserviço monolítico. Com **4,65** ($\pm 2,01$) minutos para o processamento de um vídeo, ficou claro o ganho que os padrões reativos de programação têm sobre o sistema.

A fragilidade do microsserviço Vlibras sem refatoração também é evidenciada para taxas de chegada de 4 e 8 vídeos por minuto. Nestes casos, o Core Vlibras também apresentou os piores resultados, com tempo de **25,248** ($\pm 15,05$) minutos para uma carga moderada e **43,501** (± 25) quando sobre stress. O principal a se observar para estas cargas está na configuração do *pool* de *workers*. Se para os experimentos até então os melhores resultados foram observados para uma configuração de apenas 1 *worker* por tipo de microcomponente, para as cargas moderada e de stress a situação se inverte.

Diferente do ocorrido em um ambiente de orquestração simples (1 *Pod*), nestes testes, a configuração que obteve os melhores resultados foi o set de 4 *workers*, seguido pelo set de 8 *workers* como segunda melhor composição. Com um tempo de entrega médio de apenas **9,248** ($\pm 5,2$) minutos, a solução com 4 *workers* foi a única com uma média de tempo inferior a 10 minutos para uma carga moderada. As arquiteturas com 1 e 8 *workers* obtiveram médias de tempo semelhante, sendo respectivamente **12,242** (± 8) e **11,05** ($\pm 7,81$) aproximadamente. No cenário de stress, a diferença entre os tempos das configurações reativas não foi tão grande, estando em **40.11**, **38.524** e **39.59** para 1, 4 e 8 *workers* nesta ordem.

6.2.3. Avaliação da Resiliência

O último dos aspectos desejados com a programação reativa e um dos principais pontos da adoção de microsserviços está na resiliência. Contudo, avaliar sistemas com falhas ou faltas não é uma atividade trivial, em especial quando se trata de ambientes em nuvem[Brilhante et al. 2014]. Dado o escopo desta pesquisa, a abordagem adotada para os testes foi a prototipagem e injeção de falhas.

Nesta, um segundo contêiner do protótipo do microsserviço "Vlibras" foi desenvolvido para injetar falhas controladas internamente ao sistema. Este *script shell* faz chamadas diretas ao sistema e interrompe os processos em execução dos *workers*, ou do *core* no caso do "microsserviço monolítico". Durante o tempo em que a falha está sendo emulada no microsserviço, o injetor de falhas impede artificialmente a recuperação destes processos. Quando por fim o sistema deva retornar ao seu estado de normalidade, este controlador reinicia os processos que parou e continua a repetir este ciclo.

Desta forma, para este tipo de experimento as variáveis foram a **taxa de falha** e a **taxa de reparo**. Estas duas taxas juntas podem ser expressas em uma razão conhecida como **Disponibilidade** (Availability) [Malhotra and Trivedi 1995]. Como uma extensão dos testes de elasticidade, os testes de resiliência foram executados utilizando a mesma

configuração de **4 instâncias** de microsserviços "Vlibras" em uma taxa de chegada constante de **4 vídeos por minuto** em paralelo. A Tabela 13 denota os cenários projetados para cada disponibilidade esperada.

Tabela 13. Injeção de Falhas para Testes de Resiliência

Cenário	Taxa de Falhas (em min)	Taxa de Reparo (em min)	Disponibilidade Aproximada (porcentagem)
Falhas Mínimas	8	1	90
Falhas Graves	8	2	80
Estado Critico	8	4	50

Como de costume para este tipo de avaliação, em todos os 3 cenários planejados, tanto a taxa de falha quanto a taxa de reparo seguem uma distribuição exponencial [Balakrishnan 1996]. Então, por exemplo, o cenário de "Falhas Mínimas" foi submetido a carga de stress relativa a 8 vídeos por minuto, enquanto seus *workers* tinham, em média, 1 falha a cada 8 minutos e um reparo de uma eventual falha a cada minuto. A taxa de falhas adotada em questão tem como base a duração máxima das submissões (60 minutos). Desta forma, é esperado que com 90% de certeza haja ao menos uma falha durante toda a duração das requisições dos clientes. O último ponto a ser ressaltado é que devido a natureza da operação de geração de vídeo, o *worker* responsável por gerenciar as operações no *Unity Vídeo* não pode ser escalonado. Desta forma, prevendo um possível ponto único de falha, os mesmos cenários de testes apresentados na Tabela 13 foram executados com as falhas focadas neste gargalo. Com isto, espera-se avaliar qual o desempenho dos microsserviços reativos quando submetidos a restrições desta natureza.

A resiliência, a última das diretrizes reativas a ser analisadas neste trabalho, se trata da capacidade de um sistema de tolerar uma eventual falha e recuperar-se. Dada a orquestração interna de um microsserviço reativo, uma falha pode ser injetada em qualquer um dos seus micro-componentes. Especificamente para o objeto de estudo em questão (Gerador de Vídeo Vlibras), um destes *workers* não permite replicações.

Deste modo, os resultados a seguir foram divididos entre os casos com falhas geradas em qualquer um dos componentes (**Falhas Gerais**) e aqueles em que as falhas injetadas foram direcionadas ao componente único do MS (**Apenas Renderizador**). Os cenários projetados variam a disponibilidade (razão entre as falhas e reparos) em 90%, 80% e 50%. Análogo aos cenários anteriores, as métricas de interesse deste caso foram a vazão, tempo médio e a porcentagem de erros.

Os gráficos a seguir (Figura 12 e 13) explicitam a vazão média obtida nestes experimentos. Representado pela superfície amarela, a vazão média do Core Vlibras obteve os piores resultados, com uma vazão média de **0.838**, **0.612** e **0.593** para os cenários de "Falhas Minimas" (90%), "Falhas Graves" (80%) e "Estado Crítico" (50%), respectivamente. Mesmo considerando as falhas apenas no renderizador, a solução reativa com 1 *worker* já demonstrou melhores resultados, tendo uma vazão média de **1.007**, **0.657** e **0.688** nesta ordem. Porém a diferença fica ainda maior quando se considerando o aumento do número de *workers* no *pool* interno.

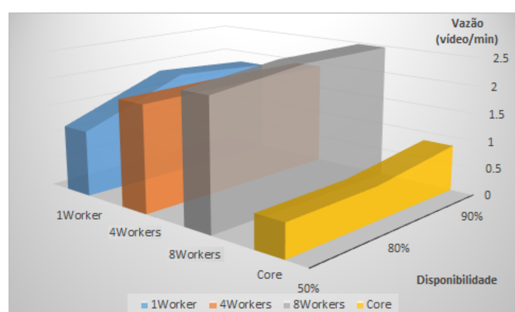


Figura 12. Vazão Média - Falhas Gerais

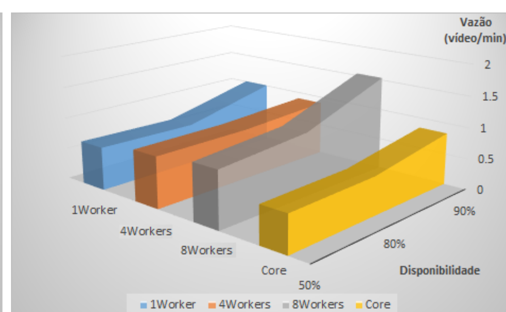


Figura 13. Vazão Média - Apenas Renderizador

Ainda que a falha injetada não atinja diretamente os micro-componentes replicados, o paralelismo das filas de mensagens acarretam em fluxo de execução mais ágil. Apesar de, sobre "Falhas Mínimas", a orquestração de 4 *workers* ter uma média (**0.895**) inferior a com 1 *worker*, a progressão da degeneração da vazão com o crescimento da taxa de reparo é bem menor, variando para aproximadamente **0.818** nos casos de falhas graves e críticas. Já a arquitetura com 8 *workers* teve os melhores desempenhos sobre falhas, conseguindo uma vazão média de **1.535** quando a 90% de disponibilidade e até **0.902** para 50%, superando de longe a arquitetura monolítica e a configuração com apenas 1 *worker*.

Este caso se repete também quando as falhas simuladas foram distribuídas por todos micro-componentes e não apenas no renderizador (Figura 12). Neste cenário, o impacto das falhas sobre a vazão do sistema caiu drasticamente, com a orquestração interna de 1 *worker* ainda detendo os menos promissores resultados (**1.777**, **1.892** e **1.183** para 90%, 80% e 50% respectivamente). Já a configuração com 4 *workers* deteve as menores variações entre os cenários de falha, atingindo uma média de **1.927** para uma disponibilidade esperada de 90% e a média de **1.909** para 50%. Entretanto, os melhores mesmo considerando falhas em qualquer componente foram por parte do *pool* com 8 *workers*, que obteve uma média de **2.474** para 90% e 80%, decaindo para **2.282** no "Estado Crítico".

7. Análise e Discussão

A análise dos resultados obtidos em cada um dos casos levou em consideração duas métricas distintas. A primeira destas métricas se refere a vazão total do sistema, ou seja, o número médio de vídeos computados em uma fração de tempo qualquer. A segunda destas métricas computadas é percentual médio de erros encontrados. Esta Seção está dividida entre a análise dos resultados nos testes para cada um dos aspectos reativos: **Responsividade**, **Elasticidade** e **Resiliência**.

Para os experimentos de elasticidade e resiliência, por possuírem cenários de múltiplas *threads* clientes, também foi levantada uma métrica referente ao tempo médio para conclusão de uma requisição. Como múltiplas conexões esperam ativamente pelo retorno dos vídeos traduzidos, apenas a vazão do sistema não é capaz de representar o comportamento, como por exemplo caso aja *deadlock* ou interrupções a *threads* anteriores em decorrência de novos clientes.

De forma semelhante, nestes casos a vazão total ao longo do tempo pode não representar de forma mais assertiva o comportamento do sistema. Para estes casos, também foi levantada a vazão média do sistema ao decorrer do tempo, uma vez que devida a existência de clientes. A Figura 14 traz um gráfico de superfície com os resultados sobre o percentual médio de erros, o qual afeta diretamente as vazões obtidas.

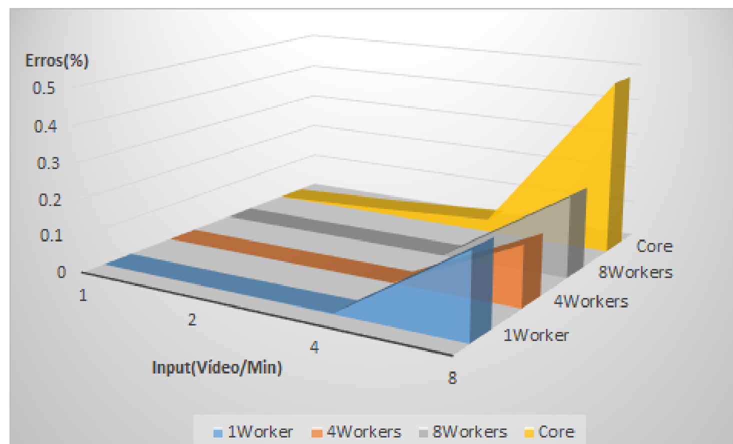


Figura 14. Erro Médio (%) - Elasticidade

Como esperado, a taxa de 1, 2 e 4 vídeo por minuto não foram suficiente para gerar erro em nenhuma das abordagens reativas. Já para o microsserviço monolítico Core, a porcentagem de falhas variou de 0 (carga baixa) a **3%** (carga moderada) com desvio padrão contendo o 0. Já para a taxa de chegada de 8 vídeos em paralelo, 4 *workers* demonstraram novamente os melhores resultados.

Com o melhor *trade-off* entre paralelismo interno e a utilização de recursos, a configuração com 4 *workers* obteve um percentual de apenas **16.5%** de falhas, contra os **22%** das configurações de 1 e 8 *workers*. O maior percentual de erros neste hipotético *"scale-up"* ficou por parte da abordagem tradicional, com **47.77%** de falhas dos vídeos recebidos.

Ainda assim, apenas minimizar a quantidade de erros não caracteriza em si uma melhoria da escalabilidade do sistema. É desejado também que a velocidade da entrega dos vídeos gerados não seja prejudicada com o incremento na capacidade de atendimento. Tendo isto em mente, a comparação entre a vazão das abordagens pode ser vista nos gráficos a seguir.

Os gráficos 15, 16, 17 e 18 mostram a vazão média ao decorrer do tempo para os cenários experimentados. A uma taxa de 1 vídeo por minuto (Cliente Único), a vazão média de todas as configurações testadas seguem uma função aproximadamente constante de média **2** vídeos por minuto. Apesar das configurações com 1 e 4 *workers* apresentarem uma menor oscilação, todos as configurações seguem uma distribuição semelhantes. De forma semelhante, este comportamento se repete para a "Carga Baixa", tendo uma média da vazão oscilando em torno de **4** vídeos por minuto, como visto na Figura 16. Isto se deve ao fato de que, por submetidos a uma taxa de chegada inferior ao limite que os sistemas podem responder (taxa de saída média), não são esperados grandes atrasos e/ou

Figura 15. Vazão Média - Cliente Único

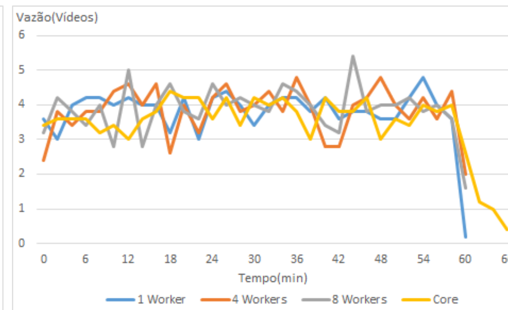


Figura 16. Vazão Média - Carga Baixa

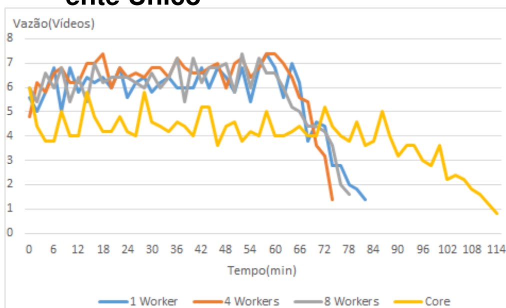


Figura 17. Vazão Média - Carga Moderada

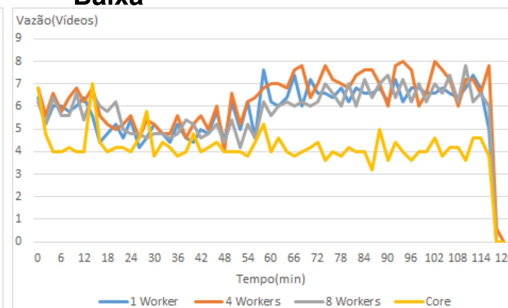


Figura 18. Vazão Média - Carga de Stress

perdas em filas.

Em contrapartida, a vazão média para as cargas "Moderada" e de "Stress" evidenciaram grandes diferenças entre a abordagem reativa da versão "monolítica". Em um cenário de "Stress", submetido a 8 vídeos por minuto, a abordagem tradicional oscilou de forma constante em **4,5** vídeos por minuto. Já para a proposta reativa, a vazão média cresceu após a primeira hora do teste, conseguindo alcançar uma média de aproximadamente **7** vídeos por minuto durante a segunda hora do teste, alcançando picos de até **8** vídeos para a melhor composição interna (4 *workers*).

Mas o caso que melhor demonstrou a superioridade da arquitetura reativa foi o cenário de "Carga Moderada"(4 vídeos/minuto). Atingindo a capacidade máxima dos recursos alocado na orquestração complexa, a abordagem de microsserviços conseguiu computar a bateria de testes em muito menos tempo. Neste cenário, a versão monolítica do Vlibras, limitada à uma vazão de até **5** vídeos/minuto, demorou quase toda duração do teste para finalizar os vídeos, durando em média **114** minutos. Por sua vez, para o *pool* reativo de micro-componentes, a mesma carga pode ser finalizada em até **74** (4 *workers*), **78** (8 *workers*) e **82** (1 *worker*) minutos.

Desta forma, todas as métricas (vazão, tempo de resposta e erro médio) levantadas corroboram com a utilização de práticas e designs reativos na arquitetura interna de um microsserviço. Uma vez que, além de um desempenho superior ou igual em orquestrações simples, a adição da escalabilidade interna dos micro-componentes incrementou a performance do processo de *"scale"* de todo microsserviço.

Figura 19. Erro Médio (%) - Falhas Gerais

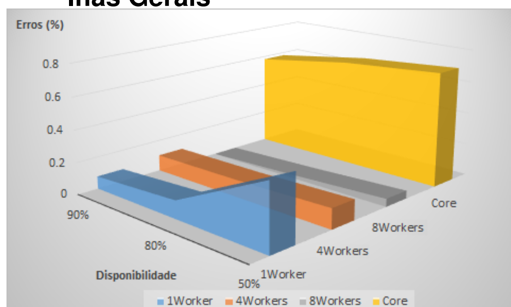
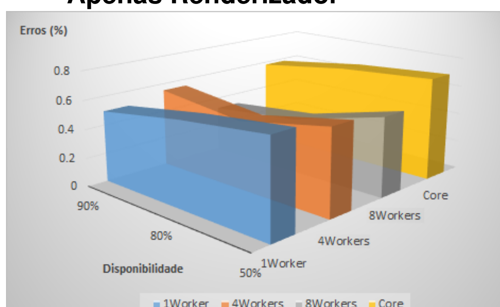


Figura 20. Erro Médio (%) - Apenas Renderizador



Reforçando as conclusões da análise da vazão média, a porcentagem média de erros (Figuras 19 e 20) também retratou um melhor funcionamento com MS reativos. No gráfico 19 é possível observar com clareza os benéficos de uma comunição orientada a mensagens. Com um percentual de erros variando de **58.1%** (falhas mínimas) à **71%** (falhas críticas), o MS monolítico teve um percentual de erros **15 vezes maior** do que a melhor orquestração reativa(8 *workers*, com apenas **4.9%** de falhas em um cenário critico. Mesmo para os *pools* contendo 1 e 4 *workers*, o percentual de erros foi bem menor.

A configuração com 4 *workers*, por exemplo, obteve um percentual de apenas **10.1 à 12.5%** de erros, estatisticamente não sendo impactado pela taxa de reparo das falhas injetadas. Com apenas 1 *worker* contudo, esta taxa de reparo impactou mais fortemente nos resultados. Para uma disponibilidade de 90%, a orquestração com 1 *worker* apresentou menos erros (**7.8%**) que o MS reativo com 4 destes. Para um percentual de 80% de disponibilidade, ambas as configurações (1 e 4) tiverem resultados semelhante estatisticamente (**11.2** e **10.7**), mas para uma taxa de reparo de 4 minutos (50% de *disponibilidade*) o *pool* sem replicações falhou em média **40.8%**, demonstrando que a escalabilidade interna de micro-componentes impactou positivamente na resiliência do sistema.

Mesmo com falhas focadas apenas no renderizador (Figura 20) este comportamento se repetiu. A orquestração com 8*workers* continuou com os melhores resultados obtidos, com **32.2%**, **38.4%** e **54.9%** em ordem de maior disponibilidade para menor. De forma semelhante a configuração com 4*workers* obteve um percentual de falhas quase constante entre os cenários, variando de **55.2%** à **59.4%** entre as falhas minimas e falhas críticas. Isto se deve, em geral, a não otimizado número de *workers* deste cenário. Ainda que haja paralelismo interno suficiente para diminuir os erros e aumentar a vazão do sistema, ainda assim não houve processamento paralelizado o suficiente para desengarrafar as filas de mensagens nos períodos que o renderizador funcionava normalmente. Como esperado também, com erros gerados apenas no renderizador os erros médios para 1 *worker* quase coincidiram com o Core Vlibras, manifestando **49.7%**, **59%** e **65.6%** de erro médio para as taxas de reparo de 1(90%), 2(80%) e 4(50%) minutos respectivamente.

Finalizando a avaliação do desempenho das arquiteturas, os recursos computacionais utilizados também foram medidos ao decorrer dos testes. Dada restrições de ambiente, o único recurso computacional disponível para medição neste caso foi o uso das

vCPU. Nesta seção será abordada o consumo destes recursos para a carga de stress, que apresentou as maiores discrepâncias de erro e vazão.

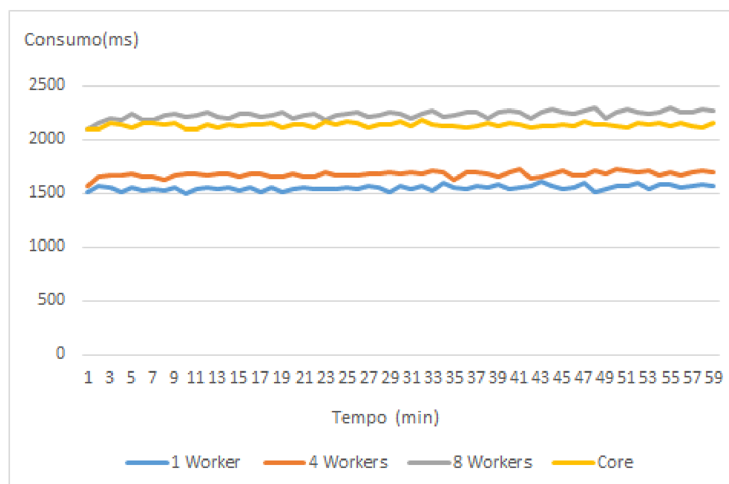


Figura 21. Consumo Médio da CPU - Desempenho

Os gráficos de linha na Figura 21, exibem os resultados ao longo do tempo da variação média da utilização das vCPUs, em microssegundos alocados. Por este motivo, o resultado da soma destes podem superar 1 segundo (ou 1000m), uma vez que o ambiente configurado contém ao todo 4 vCPUs. O comportamento demonstrado pelas curvas indica fortemente que a todos os cenários seguem a mesma função, deslocado apenas a média. Para a orquestração com 1 e 4 "*workers*" obteve-se uma média geral de **"1600"** microssegundos ou **40%** da capacidade das 4 vCPUs alocadas.

Por outro lado, o microsserviço monolítico obteve resultados ligeiramente melhores que a orquestração com 8 *workers*. Para o Core Vlibras, a média obtida foi de aproximadamente **2180**m (ou **54,5%**) em comparação aos aproximadamente **2240**m (ou **56%**). É notável então que a orquestração com 8 *workers* obteve os piores resultados entre as abordagens reativas, sendo a menos indicada para uma orquestração simples. Contudo, de forma geral, o projeto reativo sinalizou um grande incremento na utilização dos recursos computacionais, que em um ambiente real acarretaria por baratear o custo de provisionamento e tornando o processo de *scale* mais eficiente.

8. Considerações Finais

Neste trabalho foi proposta uma nova metodologia para refatoração ou desenvolvimento de microsserviços baseada em um conjunto de boas práticas de projeto para serviços web, denominada programação reativa. Com o intuito de minimizar o esforço na codificação e no projeto de microsserviços, desacoplar os módulos das macro-funcionalidades de um sistema monolítico exibiu bons resultados. Como principais contribuições deste trabalho, destacam-se: um conjunto de ações para se definir o escopo de um novo microsserviço; uma metodologia de como projetar ou refatorar novos microsserviços com filas de mensagens; a migração de uma aplicação monolítica (Vlibras) para MS; e uma abordagem para testes e avaliações de microsserviços.

O estudo partiu da seguinte questão de pesquisa: "Qual a efetividade da aplicação dos princípios reativos na organização interna de microsserviços sobre a sua resiliência e elasticidade?"e, através de dois estudos de caso, buscou avaliar, de forma quantitativa, os impactos da aplicação de tal abordagem nas três dimensões previstas na programação reativa: desempenho, resiliência e elasticidade. Os resultados foram avaliados em função de um conjunto de métricas objetivas, dentre elas o tempo de resposta, a vazão média e o erro médio. Os resultados obtidos permitiram mostrar que há indícios que a solução proposta é capaz de validar as hipóteses preliminares que foram levantadas:

- i) O uso de filas de mensagens na orquestração dos fluxos de dados internos de um microsserviço melhora a sua tolerância a falhas;
- ii) o uso de *wrapers* baseados em clientes para filas de mensagens habilitam o aproveitamento de código legado e na interoperacionalização de componentes de microsserviços escritos em linguagens diferentes;
- iii) o acoplamento de *pools* de *workers* a filas de mensagens aumentam a elasticidade lateral de microsserviços; e
- iv) microsserviços reativos permitem um melhor aproveitamento do *hardware* disponível.

Esta primeira fase da pesquisa não teve como objetivo propor uma metodologia conclusiva sobre o processo de refatoramento e projeto para MS, a qual ainda é pouco discutido na literatura e geralmente baseada na intuição e no conhecimento empírico da equipe de desenvolvimento. Entretanto, trata-se de uma análise de boas práticas e *design patterns* focados em microsserviços, em particular, quando as dimensões da programação reativa são consideradas.

Neste sentido, conforme corroborado pelos dois estudos de caso, a abordagem proposta mostrou-se promissora para a construção de microsserviços reativos durante a refatoração de sistemas monolíticos. Com pouca alteração na codificação original, foi possível extrair e orquestrar um novo microsserviço para uma aplicação legada utilizando troca de mensagens via filas de mensagens. Além disso, a arquitetura reativa interna do microsserviço mostrou-se a mais efetiva em termos de desempenho, resiliência e elasticidade.

Com relação ao desempenho do microsserviço, estava fora do escopo deste trabalho, sugerir uma configuração ideal para ser adotada no sistema utilizado como estudo de caso (Vlibras) ou qualquer outro ferramental. Como demonstrado nos resultados obtidos, o escalonamento interno de micro-componentes acarreta em um *trade-off*: por um lado, o menor número de recursos em *loop* acarreta em um ganho de desempenho, e uma orquestração pouco complexa; por outro lado, um número maior de *workers* melhora a tolerância a falhas do sistema, mais utiliza mais recursos computacionais. Desta forma, a melhor configuração entre o número de micro-componentes escalonados acaba também por depender da natureza da aplicação e do ambiente de provisionamento. Tal sintonia fina pode ser investigada em trabalhos futuros em busca de um ponto de equilíbrio. Assim como ocorre um *auto-scale* com microsserviços, uma ideia análoga pode ser aplicada para os *workers* de cada *pool* interno.

Outra proposta de trabalho futuro envolve a comparação da metodologia proposta com os métodos tradicionais, quando aplicados a um sistema inteiramente projetado como

microsserviço. Neste cenário, eliminando-se a complexidade de lidar com códigos legados e restrições de modelagem existentes em um sistema monolítico existente, talvez seja possível aperfeiçoar a metodologia proposta.

Por fim, um efeito colateral esperado é que o desenvolvimento de um microsserviço reativo pode ser um pouco mais complexo, uma vez que a sua comunicação interna também passa a ser dirigida por mensagens. Entretanto, acredita-se que a adoção de um padrão arquitetural comum, encapsulado em um *framework* e bem documentado, possa minimizar este problema. Realizar uma avaliação desse e de outros aspectos relacionados a adoção de de filas de mensagens para migração de sistemas legados é também uma das propostas de trabalho futuro.

Referências

Alshuqayran, N., Ali, N., and Evans, R. (2016). A systematic mapping study in microservice architecture. In *Service-Oriented Computing and Applications (SOCA)*, pages 44–51. IEEE.

Anderson, C. (2015). Docker. *IEEE Software*, 32(3).

Atul Shukla, R. T. (2014). Architecting reactive applications on aws.

Balakrishnan, K. (1996). *Exponential distribution: theory, methods and applications*. CRC press.

Balalaie, A., Heydarnoori, A., and Jamshidi, P. (2015). Microservices migration patterns. Technical report, Tech. Rep. TR-SUTCE-ASE-2015-01, Automated Software Engineering Group, Sharif University of Technology, Tehran, Iran.

Bass, L., Weber, I., and Zhu, L. (2015). *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional.

Boner, J. (2016). Reactive microservices architecture.

Bonér, J., Farley, D., Kuhn, R., and Thompson, M. (2014). The reactive manifesto.

Brilhante, J., Silva, B., Maciel, P., and Zimmermann, A. (2014). Eucabomber 2.0: A tool for dependability tests in eucalyptus cloud infrastructures considering vm life-cycle. In *Systems, Man and Cybernetics (SMC), 2014 IEEE International Conference on*, pages 2669–2674. IEEE.

Compton, B. T. and Withrow, C. (1990). Prediction and control of ada software defects. *Journal of Systems and Software*, 12(3):199–207.

Curry, E. (2004). Message-oriented middleware. *Middleware for communications*, pages 1–28.

Daniel, F. and Pernici, B. (2006). Insights into web service orchestration and choreography. *International Journal of E-Business Research (IJEBR)*, 2(1):58–77.

Daniel, F. and Pernici, B. (2007). Web service orchestration and choreography: Enabling business processes on the web. *E-Business Models, Services, and Communications-Advances in E-Business Research Series*, 2:251–274.

Dragoni, N., Giallorenzo, S., Lafuente, A., et al. (2016). Microservices: yesterday, today, and tomorrow. *arXiv preprint arXiv:1606.04036*.

El Emam, K., Benlarbi, S., Goel, N., et al. (2002). The optimal class size for object-oriented software. *IEEE Transactions on software engineering*, 28(5):494–509.

Esposito, C., Castiglione, A., and Choo, K.-K. R. (2016). Challenges in delivering software in the cloud as microservices. *IEEE Cloud Computing*, 3(5):10–14.

Falcao, E., Maritan, T., Duarte, A., and Pessoa, J. (2014). Deaf accessibility as a service: uma arquitetura elástica e tolerante a falhas para o sistema de traduç ao vlibras.

Fazio, M., Celesti, A., Ranjan, R., et al. (2016). Open issues in scheduling microservices in the cloud. *IEEE Cloud Computing*, pages 81–88.

Fowler, M. and Lewis, J. (2014). Microservices a definition of this new architectural term. *URL: http://martinfowler.com/articles/microservices.html*.

Gouigoux, J.-P. and Tamzalit, D. (2017). From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture. In *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*, pages 62–65. IEEE.

Gupta, A. (2015). Microservice design patterns. `http://blog.arungupta.me/microservice-design-patterns/`.

Gutierrez, F. (2017). Microservices. In *Spring Boot Messaging*, pages 179–192. Springer.

Halili, E. H. (2008). *Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites*. Packt Publishing Ltd.

Hassan, S. and Bahsoon, R. (2016). Microservices and their design trade-offs: A self-adaptive roadmap. In *Services Computing (SCC), 2016 IEEE International Conference on*, pages 813–818. IEEE.

Hatton, L. (1997). Reexamining the fault density component size connection. *IEEE software*, pages 89–97.

Johanson, A., Flögel, S., Dullo, C., and Hasselbring, W. (2016). Oceantea: Exploring ocean-derived climate data using microservices.

Kecskemeti, G., Marosi, A. C., and Kertesz, A. (2016). The entice approach to decompose monolithic services into microservices. In *High Performance Computing & Simulation (HPCS), 2016 International Conference on*, pages 591–596. IEEE.

Levcovitz, A., Terra, R., and Valente, M. T. (2016). Towards a technique for extracting microservices from monolithic enterprise systems. *arXiv preprint arXiv:1605.03175*.

Malhotra, M. and Trivedi, K. S. (1995). Dependability modeling using petri-nets. *IEEE Transactions on reliability*, 44(3):428–440.

Namiot, D. and Sneps-Sneppe, M. (2014). On micro-services architecture. *International Journal of Open Information Technologies*, 2(9):24–27.

Newman, S. (2015). *Building microservices*. "O'Reilly Media, Inc.".

Richardson, C. (2015). Introduction to microservices. *URL: https://www.nginx.com/blog/introduction-to-microservices*.

Saarimäki, N., Lomio, F., Lenarduzzi, V., and Taibi, D. (2019). Does migrate a monolithic system to microservices decreases the technical debt? *arXiv preprint arXiv:1902.06282*.

Sill, A. (2016). The design and architecture of microservices. *IEEE Cloud Computing*, pages 76–80.

Thönes, J. (2015). Microservices. *IEEE Software*, pages 116–116.

Videla, A. and Williams, J. J. (2012). *RabbitMQ in action: distributed messaging for everyone*. Manning.

Villamizar, M., Garcés, O., et al. (2015). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *Computing Colombian Conference (10CCC), 2015 10th*, pages 583–590. IEEE.

Vohra, D. (2017). Scheduling pods on nodes. In *Kubernetes Management Design Patterns*, pages 199–236. Springer.