UNIVERSIDADE LUSÓFONA

RODNEY NUNES PORTO

ANÁLISE COMPARATIVA DE ARQUITETURAS DE SOFTWARE PARA UM SISTEMA DE GESTÃO DE ALUNOS

Orientador: Profº Doutor José Luís Azevedo Quintino Rogado

Universidade Lusófona de Humanidades e Tecnologias

Escola de Comunicação, Arquitetura, Artes e Tecnologias da Informação

Departamento de Engenharia Informática e Sistemas de Informação

Lisboa

2021

RODNEY NUNES PORTO

ANÁLISE COMPARATIVA DE ARQUITETURAS DE SOFTWARE PARA UM SISTEMA DE GESTÃO DE ALUNOS

Dissertação defendida em provas públicas para a obtenção do Grau de Mestre no Curso de Mestrado em Engenharia Informática e Sistemas de Informação, perante o júri, com o Despacho de Nomeação Nº 275/2021, de 4 de outubro de 2021, com a seguinte composição:

Presidente : Prof. Doutor Paulo Jorge Tavares Guedes
Arguente: Prof. Doutor Pedro Hugo Queirós Alves,
Orientador: Prof. Doutor José Luís Azevedo Quintino Rogado

Universidade Lusófona de Humanidades e Tecnologias
Escola de Comunicação, Arquitetura, Artes e Tecnologias da Informação
Departamento de Engenharia Informática e Sistemas de Informação
Lisboa

2021

ÍNDICES

ÍNDICES 2

Dedicatórias 7

Agradecimentos 8

Resumo 9

Abstract 10

Résumé 11

Índice de Figuras 12

Índice de Tabelas 14

Capítulo 1 - Introdução 16

1.1 - Motivação 17

1.2 - Estrutura da dissertação 18

Capítulo 2 - Arquitetura de software 20

2.1 - Breve historial da arquitetura de software 20

2.2 - A importância da arquitetura de software 22

2.3 - Uma “boa” arquitetura de software 23

2.4- Sistema de gestão de alunos em contexto universitário 25

2.4.1 – Os componentes da gestão de alunos 25

2.5 – Arquitetura monolítica 27

2.5.1 - Modelo de arquitetura em camadas 27

Figura 1 - Arquitetura em 3 camadas 28

Figura 2 – Arquitetura em N camadas 29

2.5.2 - Arquitetura implementada 30

Figura 3 – Arquitetura monolítica implementada para gestão de alunos 31

Figura 4 - Diagrama de Classes da arquitetura monolítica 33

2.5.3 - Base de Dados Relacional - BDR 33

Tabela 1 - Representação da entidade CAO em uma tabela de uma BDR 35

Tabela 2 - Representação da entidade PESSOA em uma tabela de uma BDR 35

Tabela 3 - Resultado do SQL de exemplo para buscar informações sobre os cães 37

2.6 – Arquitetura de micro-serviços 38

2.6.1 – API Gateway 39

2.6.2 – Base de dados não relacional – NoSQL 40

2.6.3 – Service Discovery e Message Broker 42

Figura 5 – Exemplo de atuação do Message Broker 44

2.6.4- A arquitetura Implementada 44

Figura 6 – Arquitetura de micro-serviços implementada para gestão de alunos 45
Figura 7 – Comunicação síncrona e assíncrona entre os micro-serviços 46

Capítulo 3 – A Computação em Nuvem 48
3.1 – O início da Computação em Nuvem 48
3.2 – Tipos e serviços de Computação em Nuvem 49
3.2.1 - Tipos de Computação em Nuvem 49
3.2.2 - Tipos de serviços de Computação em Nuvem 49
3.3 – A importância da Computação em Nuvem 50

Capítulo 4 – Escalabilidade, manutenção e resiliência 53
4.1 – Escalabilidade de software 53
4.1.1 – Escalabilidade de arquitetura monolítica 53
Figura 8 – Escalabilidade da arquitetura monolítica 54
4.1.2 – Escalabilidade de arquitetura de micro-serviços 54
Figura 9 – Escalabilidade da arquitetura de micro-serviços 55
4.2 – Manutenção de software 56
4.2.1 – Manutenção da arquitetura monolítica 56
Figura 10 – Diagrama de Sequência para criar um novo aluno em arquitetura monolítica 57
4.2.2 – Manutenção da arquitetura de micro-serviços 58
Figura 11 – Diagrama de Sequência para criar um novo aluno em arquitetura de micro-serviços 59
Figura 12 – Comunicação assíncrona (via Publishers e Subscribers) e síncrona (via API Rest) 61
4.3 – Resiliência de software 61
Figura 13 – Fluxo de desenvolvimento de um software 62
4.3.1 – Resiliência da arquitetura monolítica 63
4.3.2 – Resiliência da arquitetura de micro-serviços 64

Capítulo 5 – Programação bloqueante e programação reativa 66
5.1 – A tecnologia Java e o Framework Spring 66
5.2 – Evolução da capacidade de processamento computacional 67
5.3 – Programação bloqueante 68
5.3.1 – Programação Não-bloqueante 69
5.4 – Programação reativa 70
5.4.1 – Manifesto reativo 71
Figura 14 – Características de um sistema segundo o Manifesto Reativo (Reactive Manifesto Org, 2021) 71
5.4.2 – Como funciona a programação reativa 73

Capítulo 6 – Tecnologias complementares para as arquiteturas 76
6.1 – Tecnologias complementares para arquitetura monolítica 76
6.1.1 – MySQL Server 76

6.2 – Tecnologias complementares para arquitetura de micro-serviços 77
6.2.1 – MongoDB 77
Figura 15 - Comparação entre NoSQL MongoDB e uma BDR. 78
6.2.2 - RabbitMQ 78
Figura 16 - Simples publish e subscribe uma mensagem no RabbitMQ 79
Figura 17 - Publish de uma mensagem em concorrência de subscribers no RabbitMQ 80
Figura 18 - Publish de uma mensagem para vários subscribers no RabbitMQ 80
6.2.3 - Docker e o Kubernetes 80
Figura 19 - Aplicações conteinerizadas com o Docker, (Docker., 01/07/2021) 81

Capítulo 7 – Testes das arquiteturas: monolítico x micro-serviços 84
7.1 – A API REST 84
7.1.1 – API REST 84
Figura 20 - Modelo de uma Web API REST sobre os protocolos HTTP ou HTTPS 85
7.1.2 – API REST desenvolvida 85
Tabela 4 - Exemplos de URL idênticas entre a APIs monolítica e de micro-serviços 87
7.2 – Apache JMeter 87
Figura 21 - Configuração de um Plano de Testes do Jmeter 89
Figura 22 - Exemplo de um Grupo de Threads do Jmeter 90
Figura 23 - API do sistema de gestão de alunos criado no JMeter 91
7.3 – Testes de Desempenho 93
7.3.1 - Ambiente de Computação em Nuvem 94
Figura 24 - Cluster do Google Kubernetes Engine da Google Cloud 94
Figura 25 - Configuração da instância do Cloud SQL da Google Cloud 95
7.3.2 - Teste 1: teste de sistema “Frio” 96
Figura 26 - Tempo em segundos de execução da API REST 97
Figura 27 - Execução da API por 10 threads sobre sistema monolítico 98
Figura 28 - Execução da API por 10 threads sobre sistema de micro-serviços 98
7.3.3 - Teste 2: Teste de desempenho sequencial 99
Figura 29 - Resultado da latência média por execução para 50 utilizadores 100
Figura 30 - Resultado de throughput médio por execução para 50 utilizadores 101
Figura 31 - Resultado de % de erros por requisição para 50 utilizadores 102
Figura 32 - Resultado médio da latência para 50 utilizadores 103
Figura 33 - Resultado médio do throughput para 50 utilizadores 103
Figura 34 - Resultado da latência média por execução para 250 utilizadores 104

Figura 35 - Resultado de throughput médio por execução para 250 utilizadores 105
Figura 36 - Resultado de % de erros por execução para 250 utilizadores 106
Figura 37 - Resultado médio da latência para 250 utilizadores 107
Figura 38 - Resultado médio do throughput para 250 utilizadores 107
Figura 39 - Resultado médio do % de erros para 250 utilizadores 108
7.3.4 - Teste 3 - Teste de desempenho com 3 micro-serviços desativados 109
Figura 40 - Resultado médio da latência para 250 utilizadores 110
Figura 41 - Resultado médio do throughput para 250 utilizadores 110
Figura 42 - Resultado médio do % de erros para 250 utilizadores 111
7.3.5 - Comportamento da computação em nuvem 112
Figura 43 – Uso de memória do micro-serviços disciplines no GKE 113
7.3.6 - Custos da computação em nuvem 113
Figura 44 – Custo total da prova de conceito na Google Cloud 114
Figura 45 – Custo da Base de Dados Relacional com a Cloud SQL na Google Cloud 115

Capítulo 8 - Conclusão e trabalhos futuros 116
8.1 - Conclusão Final 119
8.2 - Trabalhos Futuros 121
Figura 46 – Arquitetura de micro-serviços com o micro-serviço do EventStore do event sourcing 122

Apêndices 124
Apêndice I – Base de dados NoSQL de alunos 124
Apêndice II – Base de dados NoSQL de fichas financeiras 125
Apêndice III – Base de dados NoSQL de registros acadêmicos 126
Apêndice IV - Ampliação da figura 4 - Diagrama de Classes da arquitetura monolítica 126
Apêndice V - Ampliação da figura 8 - Diagrama de Sequência para criar um novo aluno em arquitetura monolítica 128
Apêndice VI - Ampliação da figura 9 - Diagrama de Sequência para criar um novo aluno em arquitetura de micro-serviços 129

Referências Bibliográficas 130

Glossário 135

Folha propositalmente em branco

Dedicatórias

Para Maria do Socorro Nunes Gonçalves e Enito Aparício Porto, por vossa dedicação para prover aos filhos educação formal e informal, mesmo que não tenham tido, e assim construir bons cidadãos para o Mundo.

Agradecimentos

Agradeço ao Papai do Céu por me oferecer o milagre da vida e permitir obter mais essa experiência em um caminho que apenas Ele conhece.
Agradeço a Hannah Nunes Yahiaoui, que tem sido uma parceira de vida, ao meu lado com o suporte de seu Amor.
Agradeço a Mara Martins e Franky Sanchez, amigos que compartilharam seu conhecimento.

Resumo

Esta dissertação apresenta uma evolução recente na área de desenvolvimento de software, que acompanha uma evolução simultânea nos componentes de hardware, com base na comparação de duas arquiteturas de software, realizada através do estudo de duas implementações de um sistema de gestão de alunos em contexto universitário, tendo como objetivo o de demonstrar como este modelo de negócio pode ser gerido de forma mais eficiente, tanto tecnológica como economicamente, através da otimização das suas características e aspectos arquiteturais.

Neste sentido, a melhoria de eficiência aqui referida baseia-se na comparação de duas arquiteturas distintas: uma baseada num modelo monolítico – ou modelo clássico; e outra baseada num modelo de micro-serviços – modelo mais recente, sendo apresentados o enquadramento teórico e a aplicação prática de ambas as arquiteturas, bem como as tecnologias e ferramentas utilizadas no processo de criação e desenvolvimento aplicacional, da realização de testes, e da comparação de desempenho.

Com base neste estudo comparativo, será apresentado em conclusão a vantagem que a arquitetura de software baseada em micro-serviços possibilita em termos de eficiência e custo/benefício, sendo igualmente fornecida uma abordagem que permite optar por uma destas arquiteturas, em função de diversos parâmetros e objetivos de negócio.

Palavras-chave: *arquitetura de software, micro-serviços, monolítico, testes de performance, gestão de alunos*

Abstract

This dissertation presents a recent evolution in software development, which accompanies a simultaneous evolution in hardware components, based on the comparison of two software architectures, carried out through the study of two implementations of a Student Management System in a university context, with the objective of demonstrating how this business model can be managed more efficiently, both technologically and economically, through the optimization of its characteristics and architectural aspects.

In this sense, the efficiency improvement referred to here is based on the comparison of two distinct architectures: one based on a monolithic model - or classical model; and the other based on a microservices model - a more recent model. The theoretical framework and the practical application of both architectures are presented, as well as the technologies and tools used in the process of creation and application development, of carrying out tests, and of performance comparison.

Based on this comparative study, the advantage of the microservices-based software architecture in terms of efficiency and cost/benefit will be presented in conclusion, and an approach will also be provided that allows the choice of one of these architectures, depending on various parameters and business objectives.

Keywords: *software architecture, microservices, monolithic, performance testing, student management*

Résumé

Cette thèse présente une évolution récente dans le domaine du développement de logiciel, qui accompagne une évolution simultanée dans les composants logiciels, basée sur la comparaison entre deux architectures logicielles, réalisée à travers l'étude de deux implémentations d'un système de gestion des étudiants dans un contexte universitaire, ayant pour objectif de démontrer comment ce modèle d'entreprise peut être géré plus efficacement, à la fois technologiquement et économiquement, à travers l'optimisation de ses caractéristiques et de ses aspects architecturaux.

En ce sens, l'amélioration de l'efficacité dont il est question ici repose sur la comparaison de deux architectures distinctes: l'une basée sur un modèle monolithique - ou modèle classique - et l'autre basée sur un modèle de micro-services - un modèle plus récent. Le cadre théorique et l'application pratique des deux architectures sont présentés, ainsi que les technologies et les outils utilisés dans le processus de création et de développement des applications, de réalisation des tests et de comparaison des performances.

Sur la base de cette étude comparative, il sera présenté en conclusion l'avantage que permet l'architecture logicielle basée sur les microservices en termes d'efficacité et de coût/bénéfice, en ayant également une approche qui permet d'opter pour l'une de ces architectures, en fonction de divers paramètres et objectifs commerciaux.

Mots clés: architecture logicielle, microservices, monolithique, tests de performance, gestion des étudiants

Índice de Figuras

Figura 1 - Arquitetura em 3 camadas
Figura 2 – Arquitetura em N camadas
Figura 3 – Arquitetura monolítica sem a camada de apresentação
Figura 4 - Diagrama de Classes da arquitetura monolítica
Figura 5 – Exemplo de atuação do message broker
Figura 6 – Arquitetura de micro-serviços implementada para gestão de alunos
Figura 7 – Comunicação síncrona e assíncrona entre os micro-serviços
Figura 8 – Escalabilidade da arquitetura monolítica
Figura 9 – Escalabilidade da arquitetura de micro-serviços
Figura 10 – Diagrama de Sequência para criar um novo aluno em arquitetura monolítica
Figura 11 – Diagrama de Sequência para criar um novo aluno em arquitetura de micro-serviços
Figura 12 – Comunicação assíncrona (via Publishers e Subscribers) e síncrona (via API Rest)
Figura 13 – Fluxo de desenvolvimento de um software
Figura 14 – Características de um sistema segundo o Manifesto reativo (The Reactive Manifesto, 2021)
Figura 15 - Comparação entre NoSQL MongoDB e uma BDR
Figura 16 - Simples publish e subscribe de uma mensagem no RabbitMQ
Figura 17 - Publish de uma mensagem em concorrência de subscribers no RabbitMQ
Figura 18 - Publish de uma mensagem para vários subscribers no RabbitMQ
Figura 19 - Aplicações conteinerizadas com o Docker, (Docker, 01/07/2021)
Figura 20 - Modelo de uma web API REST sobre os protocolos HTTP ou HTTPS
Figura 21 - Configuração de um plano de testes do Jmeter
Figura 22 - Exemplo de um grupo de threads do Jmeter
Figura 23 - API do sistema de gestão de alunos criado no JMeter
Figura 24 - Cluster do Google Kubernetes Engine da Google Cloud
Figura 25 - Configuração da instância do Cloud SQL da Google Cloud
Figura 26 - Tempo em segundos de execução da API REST
Figura 27 - Execução da API por 10 threads sobre sistema monolítico
Figura 28 - Execução da API por 10 threads sobre sistema de micro-serviços
Figura 29 - Resultado da latência média por execução para 50 utilizadores

Figura 30 - Resultado de throughput médio por execução para 50 utilizadores
Figura 31 - Resultado de % de erros por requisição para 50 utilizadores
Figura 32 - Resultado médio da latência para 50 utilizadores
Figura 33 - Resultado médio do throughput para 50 utilizadores
Figura 34 - Resultado da latência média por execução para 250 utilizadores
Figura 35 - Resultado de throughput médio por execução para 250 utilizadores
Figura 36 - Resultado de % de erros por execução para 250 utilizadores
Figura 37 - Resultado médio da latência para 250 utilizadores
Figura 38 - Resultado médio do throughput para 250 utilizadores
Figura 39 - Resultado médio do % de erros para 250 utilizadores
Figura 40 - Resultado médio da latência para 250 utilizadores
Figura 41 - Resultado médio do throughput para 250 utilizadores
Figura 42 - Resultado médio do % de erros para 250 utilizadores
Figura 43 – Uso de memória do micro-serviços disciplines no GKE
Figura 44 – Custo total da prova de conceito na Google Cloud
Figura 45 – Custo da Base de Dados Relacional com a Cloud SQL na Google Cloud
Figura 46 – Arquitetura de micro-serviços com o micro-serviço do Event Store do Event Sourcing

Índice de Tabelas

Tabela 1 - Representação da entidade CAO em uma tabela de uma BDR

Tabela 2 - Representação da entidade PESSOA em uma tabela de uma BDR

Tabela 3 - Projeção do resultado da SQL de exemplo

Tabela 4 - Exemplos de URL idênticas entre a API monolítica e de micro-serviços

Folha propositalmente em branco

Capítulo 1 - Introdução

É notório que a tecnologia, através de sua evolução nos segmentos de hardware e software, é um dos pilares nos dias atuais para as empresas em busca de uma boa relação da eficiência custo/benefício. Seja sobre as empresas aumentarem seus investimentos ou reduzirem seus custos para, em ambos os casos, melhorarem o benefício do retorno financeiro. E reduzir custos tem-se tornado ainda mais presente nas decisões organizacionais baseando-se, algumas vezes, na própria evolução da tecnologia como argumento para a implementação de novas, ou adaptação de já existentes, regras e/ou processos de negócio.

Com a intenção de demonstrar como a tecnologia pode ser realmente uma importante e relevante parte das decisões de investimento organizacionais, será apresentada neste trabalho uma comparação entre a arquitetura de software monolítico e arquitetura de software de micro-serviços, a fim de identificar qual destas pode ser mais eficiente, independentemente do cenário de negócio e área de atuação de uma empresa. Neste trabalho é utilizado como caso de estudo um sistema de gestão de alunos em contexto universitário, que será comparado em duas implementações, tendo por base os dois modelos de arquitectura referidos.

Neste trabalho, a apresentação destas duas arquiteturas de software tem como objectivo demonstrar como a evolução tecnológica pode realmente ajudar uma organização na tomada de decisão em busca de uma melhor eficiência da relação custo/benefício no momento de criar uma nova, ou otimizar uma já existente, solução ou processo de negócio, ou ao menos, avaliar a substituição de uma arquitetura de software mais antiga por uma arquitetura de modelo mais recente para otimizar a eficiência de um negócio já em produtividade operacional.

É preciso para isso contextualizar ambas arquiteturas de software de acordo com o software desenvolvido para atender às necessidades de negócio e além destes, também contextualizar ambas arquiteturas de acordo com o “ecossistema de software” que as compõem e que atendem, ou definem, as suas características.

Antes, ainda, é feita uma introdução sobre o que é uma arquitetura de software para que seja entendido o porquê de sua importância.

Um pouco além das arquiteturas de software, também será apresentado uma abordagem da programação bloqueante, o modelo clássico implementado com software monolítico, e a programação reativa, modelo evolutivo recomendado para software micro-serviço. A comparação destes dois paradigmas de programação será feita utilizando a mesma tecnologia de desenvolvimento e programação, a linguagem Java da Oracle, e seus recursos de *Libraries* e *Frameworks*. Além de cada uma das aplicações, também será feita uma breve apresentação dos componentes que integram o “ecossistema de software” de cada arquitetura, como por exemplo as bases de dados e os sistemas de mensageria utilizados. E para finalizar, serão descritas as ferramentas utilizadas na execução de testes e análise de desempenho e de recursos utilizados, permitindo uma comparação completa da eficiência de ambas as arquiteturas.

1.1 - Motivação

Após alguns anos de experiência profissional como estagiário de programação, programador e analista de sistemas baseado em arquiteturas monolíticas, em 2018 surgiu a primeira oportunidade em trabalhar no desenvolvimento de um sistema que funcionaria sob uma arquitetura de micro-serviços. Entretanto, por decisões das equipes de arquitetura e de infraestrutura, o sistema em desenvolvimento não seguiu propriamente uma arquitetura de micro-serviços, tornando-se limitado a uma arquitetura de software híbrida de monolítico e de micro-serviços com programação bloqueante.
Em 2019 surgiu então a oportunidade de trabalhar, em uma outra empresa, no desenvolvimento de um sistema que seguiria totalmente uma arquitetura de micro-serviços, incluindo o desenvolvimento com a recomendada programação reativa. Entretanto, e mais uma vez, por decisões das equipes de arquitetura e de infraestrutura, o desenvolvimento do sistema que seguia totalmente sob a arquitetura de micro-serviços foi descontinuado, e foi tomada a opção de o reconstrui-lo como um sistema limitado a uma arquitetura híbrida de monolítico e de micro-serviços, sem o recurso à programação reativa.

Sendo assim, a motivação para este trabalho de dissertação surgiu pelo fato de tentar perceber melhor, e demonstrar, os benefícios da arquitetura de micro-serviços que já se encontra consolidada há alguns anos na comunidade tecnológica e em grandes organizações de diferentes áreas de atuação, em todo o mundo. Mas, entretanto, a arquitetura de software de micro-serviços ainda encontra resistência em equipes de arquitetura e de infraestrutura de muitas organizações, independentemente da área de

atuação e tamanho destas organizações, quando esta arquitetura poderia, e faria sentido, ser adotada para muitos projetos de software. Obviamente devem ser consideradas e avaliadas as necessidades do modelo de negócio a ser suportado para o uso de qualquer tecnologia e arquitetura.

O fato de ter vivido de perto a experiência das decisões tomadas por diferentes equipes de arquitetura e de infraestrutura, em diferentes organizações, que levaram a abandonar uma arquitetura de software específica, supostamente mais atrativa, também foi um fator motivacional relativamente ao assunto desta dissertação, pois como veremos adiante, a arquitetura de software possui um alto nível de importância tanto a nível tecnológico como econômico para os projetos de software nas organizações.

Outras motivações foram o estudo um pouco mais profundo do tema abordado e a criação de um trabalho acadêmico que possa de alguma forma contribuir para discussão na comunidade tecnológica sobre a análise, avaliação e decisão de adotar, e porque adotar, uma arquitetura de software específica como solução uma necessidade de negócio.

1.2 - Estrutura da dissertação

Este trabalho dissertação está estruturado em 9 capítulos que seguem, após a Introdução, a contextualização de quatro sub temas que podem ser classificados como: as arquiteturas de software; o modelo de negócio suportado pelas arquiteturas de software; as tecnologias que compõem as arquiteturas de software; e testes para a comparação entre as arquiteturas de software.

Mais detalhadamente, os capítulos desta dissertação são:
Capítulo 1 - Introdução : uma introdução sobre o tema principal abordado, e ainda a motivação e a estrutura da dissertação.

Capítulo 2 - A arquitetura de software : uma introdução especificamente sobre a arquitetura de software em geral, descrição do modelo de negócio que será suportado pelas arquiteturas e também a descrição detalhada das arquiteturas de software abordadas.

Capítulo 3 – A Computação em Nuvem : descrição de alguns aspectos de computação em nuvem, largamente utilizada em arquitetura de micro-serviços e também tem se tornado solução para softwares monolíticos.

Capítulo 4 – Escalabilidade, manutenção e resiliência : descrição das principais características de uma arquitetura de software, exemplificando sobre ambas as arquiteturas estudadas na dissertação.

Capítulo 5 – Programação bloqueante e programação reativa : descrição das técnicas e metodologias de programação diferentes que podem ser adotadas no desenvolvimento de software.

Capítulo 6 – Tecnologias complementares para as arquiteturas : descrição das tecnologias e software utilizados para suportar ambas as arquiteturas de software estudadas.

Capítulo 7 – Análise de performance: monolítico x micro-serviços : apresentação dos testes realizados para comparação de desempenho entre ambas as arquiteturas de software estudadas.

Capítulo 8 - Conclusão e trabalhos futuros : apresentação do resultado final a que o estudo realizado nesta dissertação nos permitiu atingir, e como esta poderia ser complementada com trabalhos futuros.

Capítulo 2 - Arquitetura de software

Antes de descrever as arquiteturas de software monolítico e de micro-serviços é importante introduzir o conceito genérico de arquitetura de software, para contextualizar ambos os tipos de arquiteturas e também apresentar a motivação que levou à escolha do tema desta dissertação.

2.1 - Breve historial da arquitetura de software

É comumente reconhecido pela comunidade tecnológica que historicamente o conceito de arquitetura de software pode ter começado a ser abordado na década de 1960.
Um dos trabalhos que corrobora esse reconhecimento é o artigo (Dijkstra, 1968) no qual são abordados vários temas além da multiprogramação, como hierarquia e estrutura de sistemas e alocação de armazenamento.
Na década de 1970, outros cientistas e pesquisadores deram prosseguimento à construção desse conceito, como (Parnas, 1979), abordando a modularidade e a extensibilidade de software, e também o artigo (Spooner, 1971), que já abordava o termo arquitetura de software.
Na década de 1980, os estudos, pesquisas e experiências prosseguiram para definir melhor, ou para ainda criar uma definição, sobre o conceito base da arquitetura de software como o conhecemos atualmente, não ficando mais somente restrita como parte interna dos conceitos de engenharia de software. Alguns trabalhos que podem ser referenciados com importância nessa altura são os de (Shaw, 1989).
Já no início da década de 1990, pouco depois de Shaw, foi publicado o artigo (Perry & Wolf, 1992), que também contribuiu com ainda mais fundamentos (alicerces) para a específica disciplina de arquitetura de software.

Nos 30 anos passados entre as décadas de 1960 e 1990, houve uma grande produção de trabalhos na área da arquitetura de software, mas ainda assim para Baragry e Reed, “Despite the volume of research since those papers were published, it is surprising that the software development community has failed to agree on exactly what we mean by the ‘software architecture’ level of design” (Baragry & Reed, 1999).

Segundo (Imran et al., 2016), as primeiras tentativas, até a década de 1980, para criar uma definição, exemplificação e aplicação conceitual e prática de uma "arquitetura de software", não foram inicialmente muito precisas nem muito organizadas, sendo muitas vezes apenas desenhadas e diagramadas, a fim de caracterizar uma arquitetura de software. Já na década de 1990, ainda segundo estes autores, o termo arquitetura de software passou então, após anos de melhorias, a ser mais reconhecido e assim, foi mais aprofundado o esforço da comunidade científica em fundamentar e documentar melhor esta disciplina na área da tecnologia informática (Imran et al., 2016).

Ainda nos estudos, pesquisas e artigos ao longo destas décadas, a arquitetura de software foi muitas vezes contextualizada junto com a engenharia de software e talvez seja esse o motivo pelo qual não tenha sido fácil defini-la e conceitualizá-la.

Mesmo nos dias atuais ainda se discute na comunidade tecnológica sobre o que é, ou representa, uma arquitetura de software, embora entretanto a disciplina também já seja mais reconhecida e aceite de forma mais ampla dentro da comunidade, como um conceito fundamental da engenharia de software, de tal forma que é possível separá-las um pouco mais claramente no âmbito do desenvolvimento de software. Portanto, seguindo o consenso atual da comunidade tecnológica, com o qual o autor desta dissertação também concorda, podem ambas disciplinas serem separadamente definidas, mesmo que ainda contendo alguma interseção de seus conceitos e práticas.

Assim, a engenharia de software é a área que se relaciona, um pouco mais, com a gestão técnica de um projeto de software atuando sobre as soluções e os processos que atendem aos aspectos estratégicos para obter melhores níveis de qualidade e de produtividade relativamente aos requisitos funcionais.

Enquanto que, a arquitetura de software é a área que se relaciona de fato com a implementação técnica da solução planejada, sendo utilizada para definir os componentes do software a ser desenvolvido e mantido, bem como sua relação com componentes externos, como software terceiro ou legado quando existirem, que também irão compor a arquitetura. Atua sobre a garantia de que o software e os componentes externos atendam as métricas de qualidade, aos requisitos funcionais e também aos requisitos não-funcionais, tais como segurança, riscos e custos da solução técnica sobre o ciclo de desenvolvimento do software, tipos de tecnologias e ferramentas, escalabilidade, desempenho e usabilidade, tipos de servidores, etc.

2.2 - A importância da arquitetura de software

Para o sucesso de um projeto de software é importante que todas as equipes envolvidas, de negócio e de tecnologia, consigam ter e partilhar uma visão macro sobre “de onde partir”, “como prosseguir” e “onde chegar”. E nessa “caminhada” encontra-se a importância da arquitetura de software, para que seja definida e especificada uma orientação técnica, segundo sua definição e atuação anteriormente descritas, sobre como é expectável que sejam realizadas as fases de desenvolvido, operacionalização e manutenção do software..

Esta orientação técnica guiará, orientará, tanto os desenvolvedores quanto os próprios arquitetos, na direção em que deve seguir o desenvolvimento do software, e a infraestrutura, no objetivo de se preparar para suportar o software, a fim de ser evitado o desperdício de tempo, e de dinheiro claro. Ou seja, a existência de uma arquitetura planejada evitará a criação de um projeto que poderá ser tecnicamente desorganizado quanto à codificação, criação de módulos e camadas de desenvolvimento, ou construção de ambientes de servidores que não atenderão as necessidades de desenvolvimento ou operação. Guiará atividades de pesquisa de soluções técnicas mais assertivas, no que respeita aos papéis e responsabilidades dos componentes do software desenvolvido e dos componentes externos além de uma mais fácil, ou menos dolorosa, manutenção evolutiva e corretiva. Possibilitará até mesmo uma otimização das reuniões, seja em quantidade ou efetividade, sejam elas de caráter técnico ou de negócio.

Esta importância da arquitetura de software como uma orientação, mesmo que técnica, não fica restrita somente às equipes técnicas. Ela também é importante para a equipe de negócios conseguir planejar sua solução, não somente pensando sobre o momento atual, como também prevendo a sua evolução futura, uma vez que a evolução de um modelo de negócio é importante para torná-lo, ou mantê-lo, competitivo perante sua concorrência. E a arquitetura de software pode ser utilizada, como já dito anteriormente, como base e orientação para decisões de negócio.

> “Applications lacking a formal architecture are generally tightly coupled, brittle, difficult to change, and without a clear vision or direction. As a result, it is very difficult to determine the architectural characteristics of the application without fully understanding the inner-workings of every component and module in the system. Basic questions about deployment and maintenance are hard to answer: Does the architecture scale? What

> are the performance characteristics of the application? How easily does the application respond to change? What are the deployment characteristics of the application? How responsive is the architecture?" (Richards, 2015)

Nesta citação de Richards são colocadas importantes questões, associadas ao planejamento e desenvolvimento de um projeto de software, que podem ser respondidas justamente através da definição de uma arquitetura de software.

2.3 - Uma "boa" arquitetura de software

Ter a certeza de que uma arquitetura de software é "boa" ou não é um pouco subjetivo, pois sua qualidade dependerá não somente de questões técnicas, mas também, e talvez principalmente, de questões de negócio, como referido no parágrafo anterior. Para auxiliar à obtenção de uma resposta sobre a qualidade de uma arquitetura de software, algumas questões podem ser formuladas tais como:
a arquitetura de software:

- Atende a necessidade de negócio expectável pelos "*Stakeholders*"?
- É compreendida e dominada, ou quase, pelos "*Stakeholders*"?
- É flexível para manutenção evolutiva relativamente às necessidades de negócio?
- É compreendida e dominada, ou quase, pela equipe técnica em geral?

Outras questões de carácter mais técnico como por exemplo: escalabilidade, manutenção e resiliência, que serão descritas em capítulo adiante neste trabalho, também são úteis e devem ser levantadas e respondidas, mas as 4 questões listadas acima são, de uma forma geral, o suficiente para que as equipas de negócio e técnicas envolvidas no projecto tenham uma resposta mais "concreta e satisfatória" para analisar a construção de uma "boa" arquitetura de software. As 2 primeiras perguntas podem ser inclusive as questões determinantes para definir a qualidade, e o futuro sucesso ou fracasso da arquitetura definida e do software em si, uma vez a que a mesma deveria atender, mais do que a requisitos não-funcionais técnicos, principalmente a requisitos funcionais de negócio e de domínio dos "*Stakeholders*".

Para auxiliar na concepção e construção de uma arquitetura de software "boa", ou ideal, para um projeto de software, as características mais importantes da arquitetura deveriam, atualmente, ser consideradas com base na infraestrutura que irá suportar e manter sua

operacionalização, sejam em ambientes de desenvolvimento, de testes ou de produção. Dessa forma, a quarta questão, das 4 levantadas acima, envolve não somente equipes de desenvolvimento e de arquitetura mas envolve também, e principalmente, a equipe de infraestrutura, ou atualmente também designada por DevOps.

A importância da equipe de infraestrutura nos projetos de software atuais tornou-se equiparável à das equipes de arquitetura e de desenvolvimento, o que não era tão evidente há anos atrás, uma vez que a evolução dos sistemas de infraestrutura têm tornado as organizações cada vez mais dependentes da tecnologia para iniciar, manter ou ampliar seus negócios.

Como exemplo, imaginemos hipotéticamente duas instituições bancárias que possuem o mesmo número de clientes, ativos, inativos e novos mensalmente. Ambas possuem os mesmos serviços de abertura de contas, análise de crédito, financiamento e seguros de automóveis e de imóveis, seguros de saúde, pagamento de contas e atendimento dos gerentes pessoalmente aos clientes.

A instituição Banco A tem agências espalhadas por todas as cidades do país e todos os processos e serviços ocorrem em mais de 50% dos casos com os clientes nas agências bancárias, havendo assim necessidade de impressão, assinatura, cópia e arquivamento de contratos e documentos em papel.

Por outro lado, a instituição Banco B não tem nenhuma agência física no país e todos os seus processos de serviços ocorrem 100% de forma digital, através de website ou de aplicativo de telemóvel.

Ao comparar ambas as instituições bancárias, é expectável que a infraestrutura tecnológica da segunda instituição possa ser maior do que da primeira, pois esta precisará suportar e manter todos os serviços funcionando praticamente 24h, além da necessidade de garantir e responder a maiores exigências de segurança e acessibilidade de contratos e documentos. Isso significa que é expectável que a infraestrutura tecnológica da segunda instituição possua, pois exigirá, mais recursos e competências da equipe de infraestrutura.

Após descrever a área da arquitetura de software de uma forma mais ampla, serão descritos os dois tipos específicos de arquitetura que serão aqui comparadas, e são a base de motivação deste trabalho de dissertação. Entretanto, como já foi percebido até este ponto, a tecnologia da informação é um meio de soluções para um problema do mundo real, ou para um modelo de negócio, que neste caso se trata de um sistema de gestão de alunos.

Ter conhecimento do modelo de negócio é parte importante no processo de se definir e escolher o tipo de arquitetura ideal para ser adotada e também poderá ajudar os leitores a perceberem melhor as arquiteturas implementadas nesta dissertação.

2.4- Sistema de gestão de alunos em contexto universitário

A gestão de alunos de uma universidade é o modelo de negócio utilizado neste trabalho de dissertação para representar uma necessidade do mundo real a ser suportada, automatizada e melhorada através de um sistema informático.

O sistema de gestão de alunos é um componente que faz parte de um sistema informático de uma universidade como um todo, sendo este responsável pela gestão e controle de todo o ciclo de vida acadêmico e financeiro de um aluno na universidade. Esta gestão e controle vai desde o início, quando na recepção de candidatura de um candidato a aluno, até o fim, quando na impressão do diploma de conclusão do curso pelo aluno.

É importante clarificar que o modelo de negócio implementado foi pensado e desenvolvido seguindo a experiência do aluno responsável por esse trabalho de dissertação como utilizador do sistema de gestão de alunos da própria Universidade Lusófona.
Não houve o contato com nenhuma equipe profissional da Universidade Lusófona, de qualquer setor, seja secretaria, serviços acadêmicos, financeiro etc, a fim de obter informações sobre as modelagens de negócio (regras, processos, entidades, etc) ou modelagens e desenvolvimento tecnológico (arquitetura de software, linguagens ou metodologias de programação, casos de uso, modelagem de dados, etc).

2.4.1 – Os componentes da gestão de alunos

O sistema de gestão de alunos possui componentes que podem ser representados e descritos como entidades e objetos de negócio ou também como regras ou processos de negócio, e todos estes componentes são descritos a seguir.

Algumas regras e processos de negócio podem impactar mais, e outras menos, o desenho arquitetural de um sistema, principalmente quanto à relação das entidades de negócio no modelo monolítico, e quanto à comunicação síncrona ou assíncrona entre os micro-serviços.

As entidades e objetos de negócio da gestão de alunos são:

Curso - Representa um curso da universidade. Um curso pode ter inúmeras disciplinas.

Turma/Disciplina - Representa uma turma, ou uma disciplina, de um curso. alunos são inscritos para estudar e esta pode ter um (ou mais) professor(es) para lecionar. As turmas possuem obrigatoriamente uma data para começar e encerrar. As turmas/disciplinas deverão registrar 4 valores de frequência e o valor de frequência final para cada aluno.

Professor - Representa o professor que irá lecionar em uma determinada turma.

Candidato - Representa uma pessoa interessada em tornar-se um aluno da universidade. Para o candidato se tornar um aluno, este deve passar pelo processo de candidatura.

Aluno - Representa um aluno da universidade e que passou, obrigatoriamente, pelo processo de candidatura. Os alunos são inscritos em uma ou mais turmas do mesmo curso do processo de candidatura. Todo aluno obrigatoriamente deve possuir uma ficha financeira e um respectivo registro acadêmico.

Ficha financeira - É responsável por registar para o aluno seus dados financeiros e quaisquer processos que envolvam questões financeiras, como por exemplo as propinas ou pedidos de documentos. O pagamento do ano letivo (propinas) pode ser realizado em pronto pagamento, em semestralidade ou em mensalidade.

Registro acadêmico - É responsável por armazenar as informações acadêmicas dos alunos como as turmas do curso em que está inscrito. Registra também, através do professor da turma, as notas dos alunos para os trabalhos e exames realizados.

As regras e processos de negócio da gestão de alunos são:

Adicionar professor a uma turma - Após registrar um novo professor o mesmo torna-se apto a ser adicionado para lecionar em uma turma.

Candidatura - É o processo de análise e avaliação de um candidato para algum curso na Universidade. Pode ser realizada pessoalmente na universidade ou à distância através de

um portal web de candidaturas. Ao ser criada uma candidatura esta deve receber o candidato e o curso. Não há a obrigatoriedade de verificar se um curso é ativo ou não.

Registrar novo aluno - É o processo que registra um novo aluno na universidade a partir de um candidato pré-existente. Ao ser criado um novo aluno, imediatamente deve ser criado também uma ficha financeira e um registro acadêmico para este aluno. É necessário informar um candidato com uma candidatura para criar um novo aluno.

Inscrição de aluno em turma - Após registrar um novo aluno o mesmo torna-se apto a realizar a inscrição nas disciplinas do curso que se candidatou e foi aprovado. Estas inscrições em disciplinas podem ocorrer pessoalmente na secretaria da universidade ou online. Ao confirmar a inscrição nas disciplinas devem ser executados automaticamente, e internamente pelo sistema de gestão de alunos, o processo de gravação destas disciplinas no registro acadêmico do aluno.

Registrar nota de aluno - É o processo que registra para o aluno a nota recebida (de trabalho, exame, frequência, Teste etc) e lançada pelo Professor da disciplina. As notas ficam salvas e registradas no registro acadêmico do aluno.

Faturação de propina - É o processo de cálculo e geração das faturas para os alunos realizarem o pagamento de sua propina. As faturas geradas ficam salvas e registradas na ficha financeira do aluno. Para calcular a propina é necessário informar a quantidade de parcelas da propina e o aluno. Internamente serão obtidas as disciplinas do aluno, em seu registro acadêmico, pois cada disciplina guarda em si o seu valor específico.

2.5 – Arquitetura monolítica

Esta é uma arquitetura de software clássica que pode ser descrita como um software único, mono, desenhado para ser independente de qualquer outro software de forma que por si só seja capaz de atender todas as necessidades de negócio, relação/interação com utilizador e tratamento/manipulação dos dados nele processados. Esta arquitetura segue, geralmente, um modelo chamado de *arquitetura em camadas*, que será descrito a seguir.

2.5.1 - Modelo de arquitetura em camadas

Este modelo de arquitetura pode ser definida em arquitetura em 3 camadas, contendo uma camada de apresentação, uma camada de negócios e uma camada de acesso a dados, ou *arquitetura em N camadas,* contendo as 3 camadas anteriormente citadas e também camadas de interface para interação com sistemas terceiros, outro módulos de software que compõem a arquitetura, etc.

Estas camadas são conceitos lógicos de implementação não sendo necessariamente construídas, processadas e disponibilizadas todas no mesmo servidor.

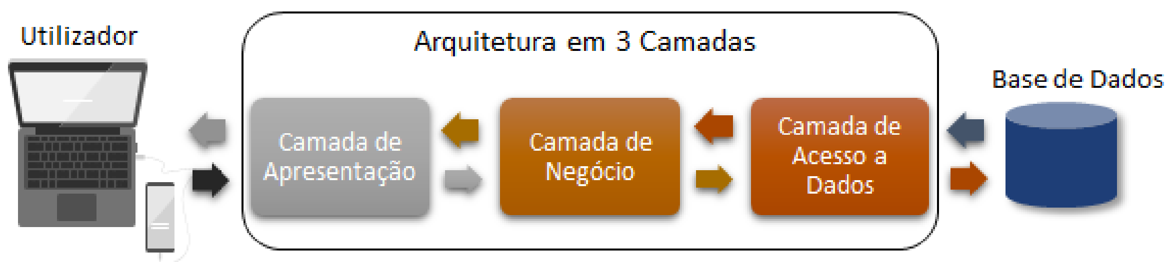


Figura 1 - Arquitetura em 3 camadas

A **Camada de Apresentação** é a responsável por oferecer uma interface de comunicação visual entre um utilizador e o software, capturando as entradas de dados, ações ou eventos - *inputs* - do utilizador e enviando para a camada de negócio. Ela obtém um retorno da camada de negócio e retorna para o utilizador uma saída de dados, ou reações às ações e aos eventos como resposta - *outputs*. Esta camada pode possuir alguma inteligência para realizar validações mais básicas dos Inputs do utilizador.

A **Camada de Negócio** é a responsável por ser a inteligência do software, ou seja, é a camada que possui a implementação das regras, processos e entidades de negócio. Ela recebe os Inputs da camada de apresentação e assim realiza validações e análises mais profundas de negócio, processa os inputs, os transforma em entidades de negócio dados e envia estas entidades para a camada de acesso a dados. Ocorrendo nesta camada um processamento com sucesso ou com erro, um retorno específico, de sucesso ou de erro, será feito para a camada de apresentação.

A **Camada de Acesso a Dados** é a responsável por transformar as entidades de negócio, recebidas da camada de negócio, em dados e assim armazená-los. Também reage a pedidos vindos da camada de negócio para acessar, ou consultar, os dados armazenados. Por essa razão é a camada que possui a relação com a base de dados do software.

Com o passar dos anos a evolução das comunicações entre sistemas se tornou cada vez maior – mesmo que em linguagens de programação diferentes – possibilitando atender a novas demandas de necessidades de negócio. Com essa evolução houve também uma atualização teórica e prática, para além do modelo com apenas 3 camadas, se tornando então uma arquitetura em N camadas, provendo dessa forma camadas de interface.

A **Camada de Interface** passou então a ser responsável por um propósito específico como por exemplo para comunicação com sistemas legados, sistemas internos de diferentes áreas e setores da mesma empresa,, sistemas terceiros, outros componentes externos e até mesmo como uma camada a mais de acesso a dados, por exemplo para acesso a uma base de dados dimensional sobre Data Warehouse. O processamento nesta camada é realizado através de uma relação junto a camada de negócio não sendo recomendado que esta tenha interação direta com nenhuma outra camada. Por ser, ou deveria ser, para um propósito específico isso poderá resultar em inúmeras camadas de interface, de acordo com as inúmeras necessidades de implementação do negócio.

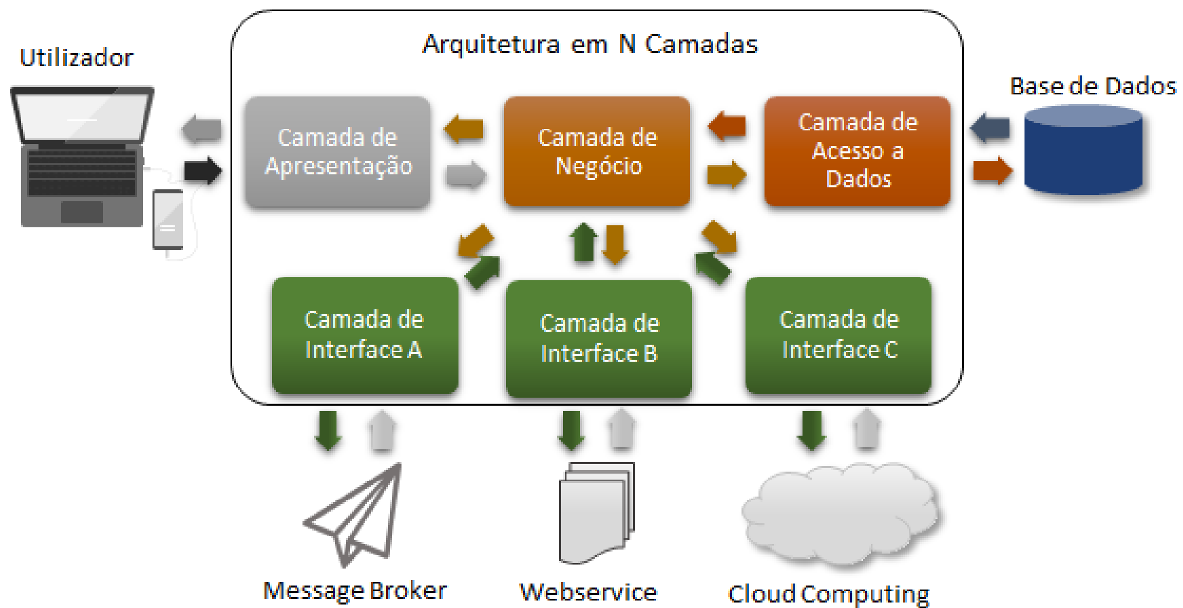


Figura 2 – Arquitetura em N camadas

Na Figura 2 acima podemos também ver, além das camadas, os componentes extras integrados que são uma Base de Dados, um message broker, para troca de mensagens entre sistemas, um Webservice, para consumo de algum serviço externo fornecido por

outros sistemas, ou uma Cloud Computing, que pode ser utilizada para outros tipos de aplicações como por exemplo para apoio a tomadas de decisão como um Business Intelligence.

2.5.2 - Arquitetura implementada

Com essa pequena introdução sobre a arquitetura em camadas fica então mais fácil perceber como uma arquitetura monolítica é pensada, organizada e desenvolvida.

Para este trabalho foi adotado o modelo em 3 camadas para a implementação resultando uma arquitetura monolítica tradicional, e ainda muito utilizada, entretanto com uma alteração sendo retirado dela a *camada de apresentação.* Desta forma a camada de apresentação também se torna um componente externo a arquitetura e deixando-a responsável somente pela *camada de negócio* e *camada de acesso a dados*, obtendo uma arquitetura monolítica um pouco mais moderna
Com esta separação entre frontend e backend é obtida uma redução na complexidade da arquitetura para seu desenvolvimento e sua manutenção, corretiva ou evolutiva, uma vez que frontend e backend poderão ser tratados como sistemas independentes com linguagens de programação e plataformas de desenvolvimento totalmente independentes, ambientes de servidores diferentes para testes, homologação e produção e até mesmo equipes de desenvolvimento diferentes e mais especializadas.

Este tipo de modelo de arquitetura no qual o frontend fica totalmente desacoplado e independente do backend ainda cria em alguns profissionais desenvolvedores, analistas e arquitetos o sentimento e/ou sensação de que o frontend continua a ser parte da arquitetura em 3 camadas como a camada de apresentação, no entanto é preciso ser feito o questionamento se a arquitetura monolítica moderna pode ser construída como uma solução de software sem o frontend. E a resposta para esse questionamento é Sim. Atualmente, muitas aplicações já foram, e estão sendo, construídas seguindo o modelo de arquitetura em camadas apenas com backend, oferecendo como interfaces de comunicação outras soluções, como por exemplo:

- a troca de mensagens via um Intermediador de Mensagens (*Message Broker*) onde consumir e publicar as mensagens representaria respectivamente os *inputs* e *outputs* da camada de apresentação;

- ou através de uma API Rest na qual há uma comunicação direta através de *Requests* (Requisições) e *Responses* (Respostas) representando respectivamente os inputs e outputs da camada de apresentação, e esta foi a interface de comunicação adotada.

A API utilizada para ambas as arquiteturas deste trabalho de dissertação será descrita no capítulo 7, na seção 7.1.

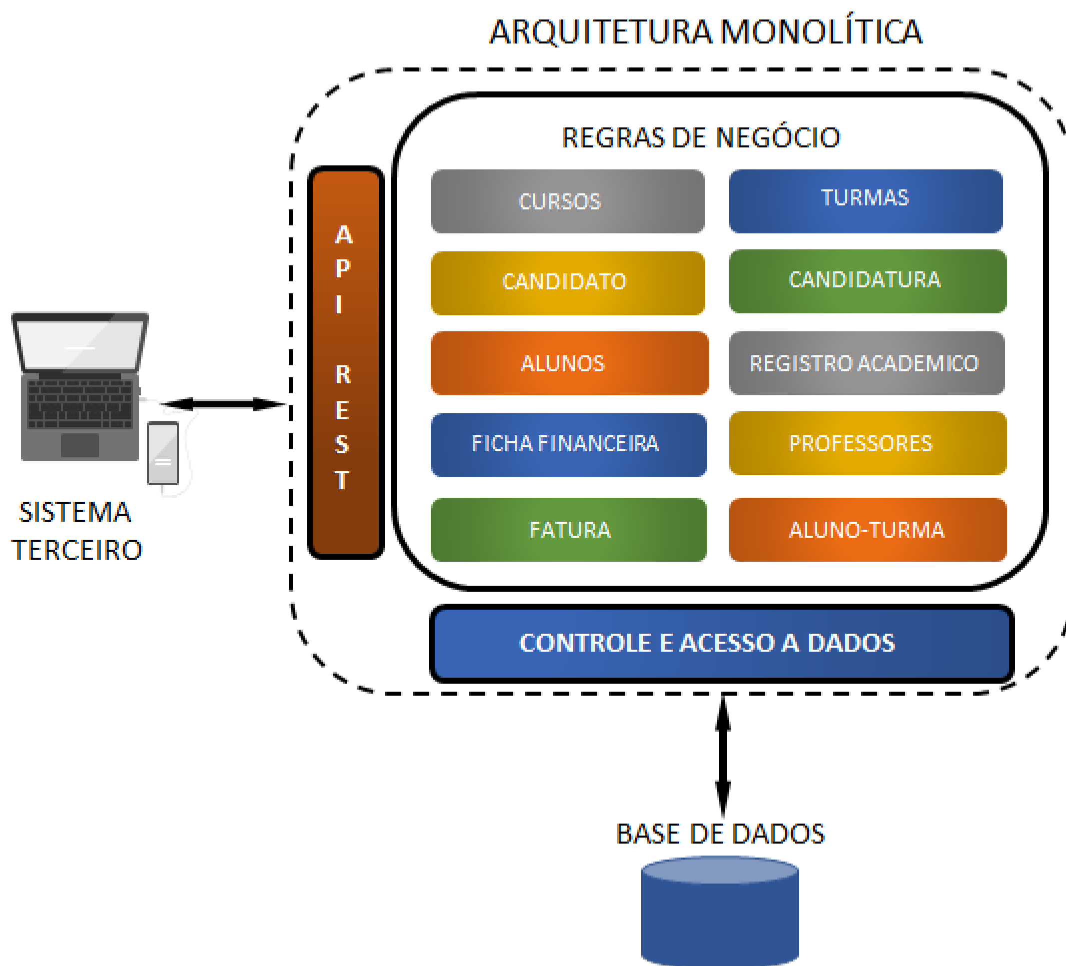


Figura 3 – Arquitetura monolítica implementada para gestão de alunos

Na Figura 3 temos então a representação da arquitetura monolítica implementada neste trabalho sobre um sistema de gestão de alunos num contexto universitário.
Podemos perceber as camadas de interface API e de acesso a dados, e seus respectivos componentes extras como um sistema terceiro e uma base de dados, como camadas mais "Simples" ao compararmos com a camada de regras de negócio, onde estão aglutinadas

todas as regras e processos de negócio implementados e que são, possivelmente, totalmente dependentes umas das outras para o correto funcionamento de todo o sistema, que como explicado anteriormente deve ser independente de qualquer outro software de forma que por si só seja capaz de atender todas as necessidades de negócio.
Cada item que compõe a camada de negócio é chamado de Entidade, ou Objeto, de negócio e por estarem de forma tão fortemente relacionadas umas com as outras, estas entidades de negócio obrigam que quaisquer novos desenvolvimentos ou manutenções corretivas e evolutivas devam seguir e respeitar as regras de relacionamento das entidades a fim de garantir a integridade e qualidade das informações processadas e armazenadas. Isso demonstra que existe um alto acoplamento dentro dessa arquitetura, porém também existe uma garantia de que o fluxo de dados que são processados entre as regras, e suas entidades de negócio, irão fluir sem maiores problemas por estarem tão fortemente relacionados e conectados.
Aqui também há o risco de que quaisquer uma das regras de negócio, ou algum de seus processos de negócios desenvolvidos, são um possível ponto crítico para a ocorrência de falhas que possam impactar todo o sistema, causando assim um problema de resiliência, como veremos mais detalhadamente à frente.

Uma das estratégias mais populares para modelar as entidades de negócio de uma arquitetura monolítica é a utilização do Diagrama de Classes da UML - *Unified Model Language*. Seguindo as normas desse tipo de diagrama é possível identificar e estruturar as relações das entidades de negócio garantindo a integridade e consistência dos dados ao serem processados e armazenados na base de dados.

Na figura 4 abaixo é apresentado o modelo de classes, em UML, que representa a gestão de alunos em contexto universitário, da arquitetura monolítica desenvolvida para este trabalho de dissertação. Ela nos permite perceber como é importante este tipo de análise no âmbito da arquitetura monolítica sobre os requisitos funcionais de negócio, pois é identificado que há classes criadas que não fazem parte do grupo de entidades de negócio demonstrados na figura 3, entretanto estas classes não identificadas anteriormente são fundamentais para a garantia de um relacionamento íntegro entre as entidades principais.

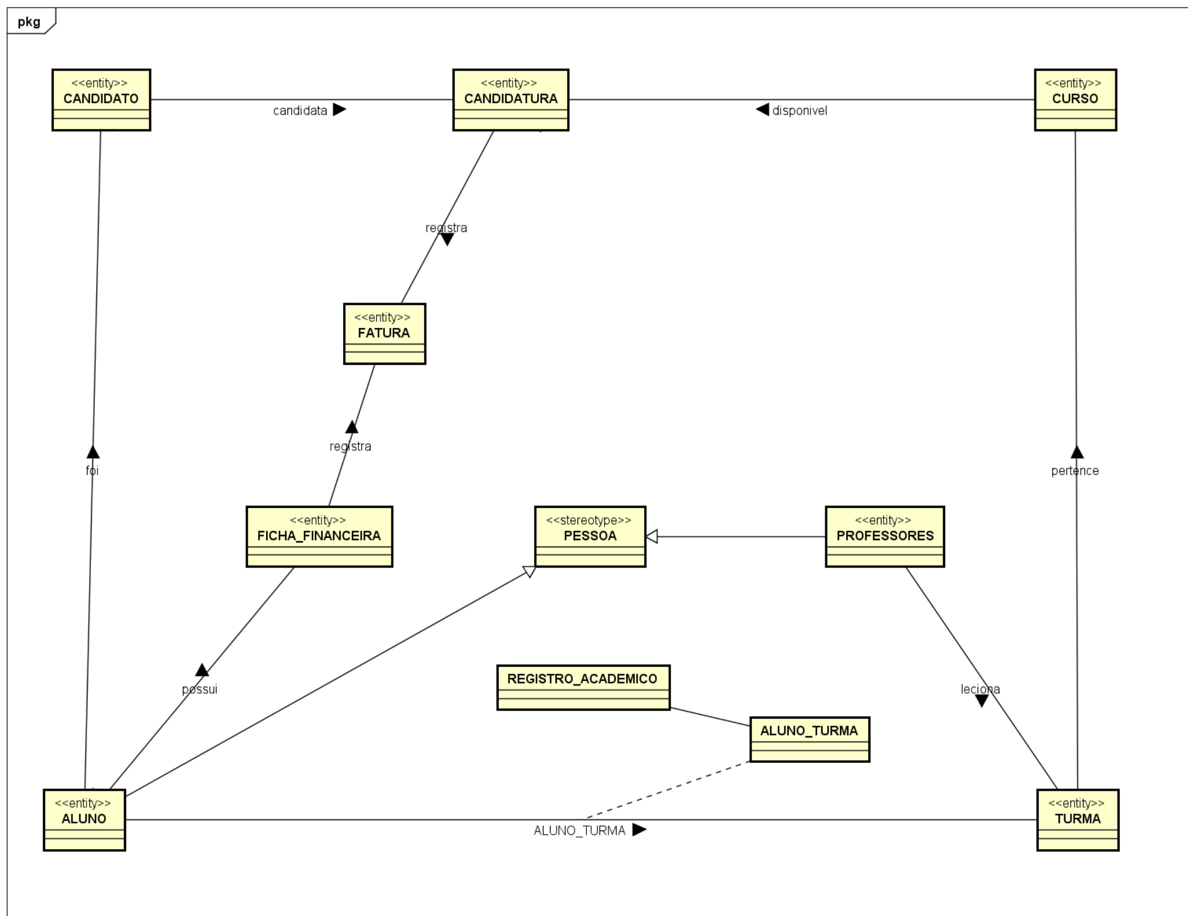


Figura 4 - Diagrama de Classes da arquitetura monolítica

Uma modelagem como a apresentada acima pode ser tão simples ou tão complexa quanto as regras e processos de negócio definidas pela empresa ou pelo *stakeholder* responsável pelo projeto.

2.5.3 - Base de Dados Relacional - BDR

Percebemos facilmente que a arquitetura monolítica utiliza, geralmente, somente uma base de dados para armazenar todas os dados operacionais sob regras de negócio desenvolvidas e processadas e seu projeto e Modelagem de base de dados segue o conceito de Modelo Entidade Relacionamento – MER – implementado assim sobre uma Base de Dados relacional.

Este é o tipo de armazenamento de dados mais difundido, sugerido e utilizado em uma arquitetura monolítica, independentemente do modelo em Camadas adotado, mas não é

incomum que também sejam utilizadas outras diferentes estruturas de armazenamento de dados como servidores de ficheiros em formatos CSV ou planilhas eletrônicas.

Como descreve a Oracle, empresa dona de uma base de dados relacional dentre as mais populares e utilizadas no Mundo:

> “... is a type of database that stores and provides access to data points that are related to one another. Relational databases are based on the relational model, an intuitive, straightforward way of representing data in tables. In a relational database, each row in the table is a record with a unique ID called the key. The columns of the table hold attributes of the data, and each record usually has a value for each attribute, making it easy to establish the relationships among data points.” (Oracle, 2021)

O armazenamento dos dados é realizado em estruturas lógicas de Tabelas (*Table*), Exibições (*View*) e Índices (*Index*) e estas estruturas lógicas ficam separadas das estruturas físicas dos dados. Isso permite que um administrador de base de dados (DBA - Database Administrator) possa gerenciar o armazenamento físico sem comprometer a estrutura lógica dos dados. Por exemplo, caso o nome da base de dados seja renomeado por motivos de backup, as tabelas nele existentes não serão renomeadas também.

A BDR fornece uma padronização na forma de armazenar, organizar, representar e consultar os dados para que possam ser utilizados por qualquer software que implemente a comunicação com este.

Os profissionais de tecnologia, sejam desenvolvedores, arquitetos ou engenheiros de software, reconhecem que um grande motivo do sucesso e adesão do modelo de uma BRD está no uso de Tabelas, por serem uma maneira intuitiva, eficiente e flexível de atender e suportar a padronização citada.

Tabela é uma estrutura de armazenamento de dados em matriz Linha X Coluna, ou seja, enquanto uma linha (também conhecida como *Tupla*) pode armazenar todos os dados de uma “instância” da entidade representada pela tabela, a coluna pode armazenar o dado de uma específica característica da entidade representada. Por exemplo, uma tabela que representa a entidade CAO teria uma, ou mais linhas, e também colunas contendo os dados das características de um cão como: ID, Raça, Sexo, Data de Nascimento, Vacinado, Nome, ID do Dono.

ID	RAÇA	SEXO	DATA_NASCIMENTO	NOME	VACINADO	ID_DONO
3	Podengo Português	M	10/01/2020	Pastel	SIM	1
4	Chiuaua	F	21/07/2019	Raclette	NÃO	2

Tabela 1 - Representação da entidade CAO em uma tabela de uma BDR

Uma das principais características que torna as tabelas de uma BDR intuitiva e eficiente é o fato de possuírem campos que são classificados como CHAVES (*Keys*) de cada linha, ou também chamado de registro e estas chaves são informações obrigatórias de serem armazenadas e, também obrigatoriamente, devem ser classificadas como Chave Primária e Única (*Primary Key*). No exemplo acima, na tabela CAO podem ser identificadas as chaves de cada registro através do campo ID. Também é percebido que há na tabela CAO a coluna ID_DONO e esta também é uma informação do tipo chave, entretanto esta não é uma chave que identifica unicamente os registros de cada CAO mas sim os registros de outra tabela, que será chamada de PESSOA. Nesse caso, esta chave por identificar o registro de outra tabela ela é chamada então de Chave Estrangeira (*Foreign Key*). Neste exemplo, a tabela que representa a entidade PESSOA possuirá as colunas: ID, Nome, Sexo, Data de Nascimento, BI, Concelho e Distrito.

ID	NOME	SEXO	DATA_NASCIMENTO	BI	CONCELHO	DISTRITO
1	Rodney	M	20/02/1991	1234567890	Sintra	Lisboa
2	Hannah	F	10/07/1995	9876543210	Matosinhos	Porto

Tabela 2 - Representação da entidade PESSOA em uma tabela de uma BDR

Essa caraterística de registros que armazenam chaves estrangeiras de outras entidades é chamada de Relacionamento. E o fato de existir estes relacionamentos entre entidades diferentes é o que classifica uma base de dados Relacional como tal se descreve.

Para que o modelo entidade relacionamento chegasse ao nível apresentado acima, houve durante muitos anos a evolução no processo de Normalização dos Dados que foram chamados de Primeira Forma Normal - 1FN, Segunda Forma Normal - 2FN e Terceira Forma Normal - 3FN, sendo esta última a que representa o nível de relacionamento

apresentado para as tabelas CAO e PESSOA, e a mais implementada em bases de dados operacionais na grande maioria das empresas, pelo motivo descrito pela própria Microsoft, empresa que também é dona de uma base de dados relacional dentre as mais populares e utilizadas no Mundo:

> “Normalização é o processo de organização de dados em uma base de dados. Isso inclui a criação de tabelas e o estabelecimento de relações entre essas tabelas de acordo com as regras projetadas para proteger os dados e tornar a base de dados mais flexível, eliminando a redundância e a dependência inconsistente.” (Microsoft, 2021)

Ter dados redundantes em uma base de dados relacional significa desperdício de espaço e aumento da complexidade das execuções transacionais dos dados, isso significa que um dado ao ser criado, alterado ou deletado deverá este mesmo dado ser redundantemente criado, alterado e deletado em todas as tabelas onde existir.

O que é uma "dependência inconsistente"? Embora seja intuitivo para um usuário procurar na tabela clientes o endereço de um determinado cliente, talvez não faça sentido procurar o salário do funcionário que chama esse cliente. O salário do funcionário está relacionado ou depende do funcionário e, portanto, deve ser movido para a tabela funcionários. As dependências inconsistentes podem dificultar o acesso aos dados porque o caminho para encontrar os dados pode estar ausente ou quebrado.

Outra força do modelo relacional de dados é o uso da linguagem de consulta estruturada chamada *SQL - Structure Query Language,* para criar, editar, deletar e consultar dados em uma base de dados, e esta tem sido, muitos anos, amplamente utilizada como a linguagem para consultas de bases de dados. A SQL é uma linguagem de escrita léxica da língua inglesa, o que fornece uma compreensão e estruturação de seus comandos através de palavras e expressões linguísticas, entretanto possui uma linguagem matemática internamente consistente, baseada na álgebra relacional, que busca otimizar o desempenho de todas as consultas à base de dados.
Um exemplo simples de SQL utilizando as tabelas acima, CAO e PESSOA, poderia ser o comando abaixo com o interesse de obter algumas informações dos cães e junto deles algumas informações de seu respectivo dono:

```
SELECT    C.RACA, C.NOME AS NOME_CAO, C.SEXO, C.DATA_NASCIMENTO, C.VACINADO,
          P.NOME AS NOME_DONO, P.BI AS BI_DONO

FROM      CAO AS C
          JOIN PESSOA AS P ON (C.ID_PESSOA = P.ID)

WHERE     P.BI != NULL  AND  TRIM(P.BI) != “”

ORDER BY  NOME_CAO
```

Com esse comando SQL estamos definindo o que queremos selecionar (*SELECT*) para serem exibidos todas as colunas de CAO e todas as colunas de PESSOA. Das (*FROM*) tabelas CAO e PESSOA, que seja respeitado o relacionamento no qual juntamos (*JOIN*) e sejam iguais os dados das colunas chave estrangeira “ID_PESSOA” em CAO e chave primária “ID”, em PESSOA. E com condição que filtre os dados a serem exibidos onde (*WHERE*) a PESSOA possua BI e, para finalizar, que ainda seja ordenado (*ORDER BY*) pelo nome da PESSOA.

Um comando SQL executado terá quase sempre a projeção de uma nova tabela como resultado. Assim, o SQL escrito acima como exemplo resultaria na projeção abaixo:

RACA	NOME	SEXO	DATA_NASCIMENTO	VACINADO	DONO	BI_DONO
Podengo Português	Pastel	M	20/02/1991	Sim	Rodney	1234567890
Chiuaua	Raclette	F	10/07/1995	Não	Hannah	9876543210

Tabela 3 - Resultado do SQL de exemplo para buscar informações sobre os cães

Um grande benefício em utilizar uma BDR são as propriedades conhecidas como ***ACID - Atomicidade, Consistência, Isolamento e Durabilidade***.

> “A atomicidade define todos os elementos que compõem uma transação completa da base de dados.
> A consistência define as regras para manter os pontos de dados em um estado correto após uma transação.
> O isolamento mantém o efeito de uma transação invisível para outras pessoas até ser confirmada, para evitar confusão.
> A durabilidade garante que as alterações de dados se tornem permanentes quando a transação for confirmada.” (Oracle, 2021)

Além da estrutura lógica e da padronização já mencionados, as propriedades ACID são benefícios obtidos quanto à segurança e garantia na integridade dos dados armazenados, no qual mesmo um modelo simples é considerado eficiente neste quesito. Isso ocorre porque estas propriedades são implementadas para atuarem no controle e gestão das transações (criação, alteração e remoção de um dado) executadas no BDR..

2.6 – Arquitetura de micro-serviços

Apesar de esta ser uma arquitetura mais moderna do que a arquitetura monolítica, não é uma ideia, ou filosofia, ainda assim tão nova quanto possa parecer o surgimento desse modelo arquitetural. Nós podemos encontrar muito dessa filosofia de micros-serviços, como micro componentes de software executando em prol de um todo em (McIlroy et al., 1978):

> “Make each program do one thing well. To do a new job, build afresh rather than compliance old program by adding new features.”, “Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information.(…)”, “Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.” (McIlroy et al.,, 1978)

Dessa forma a arquitetura oferece a proposta de trabalhar sobre vários componentes de software, entretanto pequenos e objetivos, pouco complexos, de baixíssimo acoplamento, mais resilientes, mais flexíveis para desenvolver e manter e mais facilmente escaláveis.

Como consequência desses benefícios, para que seja possível o desenvolvimento e manutenção de um sistema com vários micro componentes de software, há a necessidade de um cenário mais complexo, com mais componentes extras, a fim de garantir o bom funcionamento de todos os micro componentes de software como um todo, diferentemente do sistema monolítico que é somente um software e dessa forma mais simples. Os componentes que ajudam a construir a arquitetura de micro-serviços são: um *Gateway Server*, bases de dados não relacionais - *NoSQL*, um *Service Discovery* com *Load Balancer* e um *Message Broker*.

Vejamos a seguir a explicação de cada um destes componentes acima.

2.6.1 – API Gateway

A Figura 6 também demonstra que o frontend e o sistema terceiro se comunicam com os micro-serviços através de uma API REST pública, executando as Requests e obtendo as Responses de cada micro-serviço. Como os micro-serviços são executados em portas diferentes no mesmo servidor, ou executados em servidores diferentes, ou ainda podem haver mais de uma instância para cada um dos micro-serviços ao mesmo tempo, este modelo de arquitetura necessita de um recurso para otimizar a comunicação entre os micro-serviços e quaisquer plataformas externas ou otimizar a comunicação dos micro-serviços entre si. O recurso utilizado para resolver este problema é um “Serviço” a mais responsável por orquestrar as chamadas externas, ou internas, para as APIs públicas, ou privadas, de todos os micro-serviços de forma que todas as APIs chamadas possam ser acessadas pelo mesmo caminho (*PATH* ou *URL*) adicionando somente a parte específica que identifica cada micro-serviço e sua específica API. O serviço que controla e orquestra essa comunicação é chamado de *API Gateway*, e pode ser identificado na Figura 6, mais adiante, como o GATEWAY SERVER.

Além da API pública que é disponibilizada por cada micro-serviço também pode existir uma API privada, e é recomendável que esta exista, na qual apenas está disponível para a comunicação síncrona (descrita na seção 2.6.4) internamente entre os micro-serviços. APIs privadas são úteis para que entre os micro-serviços possam ser disponibilizadas APIs otimizadas quanto a Request e a Response não sendo obrigatório atender a todas as necessidades dos consumidores externos dos serviços.

Um fato também importante sobre o *API Gateway* é o de prover a utilização de APIs públicas dos micro-serviços idênticas, ou quase, as de uma API pública de um software monolítico já em ambiente de produção, tornando assim transparente uma possível atualização da arquitetura monolítica para a de micro-serviços. Essa funcionalidade do API Gateway remove a necessidade de atualizar todo um sistema da arquitetura monolítica para a de micro-serviços de uma única vez e sem impactar, em quase nada, na comunicação com sistemas externos e/ou clientes. Isso ocorre porque um sistema monolítico possui, em regra geral, somente um caminho (*PATH* ou *URL*) para aceder a todos os seus recursos fornecidos separados somente de forma distinta por tipo de regra e/ou Entidade de negócio.

Como prova dessa possibilidade, as APIs desenvolvidas para este trabalho de Dissertação seguem todas a mesma estrutura tanto para a arquitetura monolítica quanto para a arquitetura de micro-serviços. Esta estrutura é:
/HTTP_METHOD {PATH_URL}/{recurso_ou_Microserviço}[complemento_específico]

A API utilizada para ambas as arquiteturas deste trabalho de dissertação será descrita no capítulo 7, na seção 7.1.

2.6.2 – Base de dados não relacional – NoSQL

Pode ser percebido na Figura 6 que a camada de Acesso a Dados mudou e deixou de existir conceitualmente para Todo o sistema, porque nesse momento deixa de ser orientada um modelo de negócio para ser orientada a um processo de negócio específico ou a uma entidade de negócio específica, que será responsável por lidar com suas própria regras de negócio. Temos assim uma base de dados mais coesa, mais simples e pertencente a um software pequeno, reduzindo a complexidade sobre a integridade dos dados a serem armazenados. Porém neste caso essa redução da complexidade da base de dados cria a aceitação de duplicidade dos dados, não atendendo assim a *3ª Forma Normal* das regras de *Normalização*. Essa duplicidade dos dados será importante para que um micro-serviço não possua a necessidade de se comunicar com outros micro-serviços quando for acionado para responder simples APIs de consultas dos dados obtendo respostas com melhores performances.
Isso significa que um micro-serviço A que interage com um micro-serviço B irá armazenar para si, Micro-serviço A, os mesmos dados armazenados em micro-serviço B, e o micro-serviço B também armazenará para si dados do micro-serviço A, porém um micro-serviço armazena somente as informações que lhe interessam e importam e que são publicamente disponíveis por outro Micro-serviço. Será apresentado um pouco mais a frente um exemplo prático.
Outro fato é que nesta arquitetura a base de dados em si também muda, não somente o conceito, e seus dados deixam de estar em um modelo Relacional que segue o conceito ACID e passam a ser armazenados em um modelo *Não Relacional NoSQL – Not Only SQL*, ou seja, uma base de dados que não segue as regras e o Modelo Entidade Relacionamento - MER, regras de Normalização e o conceito ACID.

As Bases de Dados do modelo NoSQL surgiram para atender a uma necessidade de melhores performances e desempenhos para enormes volumes de dados, algo que os Bases de Dados Relacionais têm em detrimento com o passar do tempo em prol do Benefício das garantias de integridade e qualidade dos dados pela Normalização destes, principalmente sob a modelagem na 3ª Forma Normal.
No modelo NoSQL não há o conceito de “Tabelas” que se relacionam através de *Chaves Primárias* e *Chaves Estrangeiras,* o que há é o conceito de *Estrutura de Dados* que podem ser, por exemplo: - base de dados do tipo Chave – Valor, do tipo Grafos ou do tipo Documentos (armazenados como objetos JSON), além de outras estruturas de dados. Este último exemplo é exatamente o modelo adotado para este trabalho.

Segundo por exemplo a Amazon, empresa dona de um dos mais populares e utilizados serviços de Cloud Computing com base de dados NoSQL, estas são as razões pelas quais deveríamos adotar este modelo: “NoSQL databases are a great fit for many modern applications such as mobile, web, and gaming that require flexible, scalable, high-performance, and highly functional databases to provide great user experiences.” (Amazon, 2020).

Flexibilidade (flexible): é obtida geralmente por serem fornecidos esquemas flexíveis que irão permitir a equipe de desenvolvimento uma produtividade mais rápida e interativa e também por serem ideais para dados semi estruturados e não estruturados.

Escalabilidade (scalable): é obtida geralmente por serem projetados e desenvolvidos a fim de suportarem que sejam escalados horizontalmente usando clusters distribuídos de hardware, não sendo escalados verticalmente exigindo que sejam adicionando mais servidores, caros e robustos. Alguns provedores de Cloud Computing tratam dessas operações como um serviço totalmente gerenciado à parte do software que os utilizam.

Alta performance (high-performance): esta é obtida por serem desenvolvidos com otimização para atender modelos de dados específicos e com padrões de acesso que permitem gerar uma maior performance quando se tenta realizar semelhantes funcionalidades sobre uma base de dados relacional.

Altamente funcional (highly functional): esta característica é obtida por que este modelo de base de dados fornece APIs e tipos de dados altamente funcionais que foram criados, especificamente, para atender e suportar cada um de seus respectivos modelos de dados.

Como exemplo, podem ser utilizados os anexos Apêndice I, a representar o registro de um aluno na base de dados *STUDENTS*, Apêndice II, a representar o registro de uma ficha financeira de um aluno na base de dados *FINANCIAL_STATEMENTS* e Apêndice III, a representar o registro de um registro acadêmico de um aluno na base de dados *REGISTRY_ACADEMICS*, para análise de como os dados são salvos e replicados nos micro-serviços de forma que o micro-serviço de alunos possui os mesmos dados que sejam disponibilizados pelos micro-serviços de ficha financeira e registro acadêmico. Assim, caso os micro-serviços ficha financeira e registro acadêmico parem de responder por qualquer motivo, o micro-serviços de alunos possuirá as últimas informações atualizadas para si que poderão ser utilizadas.

2.6.3 – Service Discovery e Message Broker

Neste modo arquitetural se um micro-serviço A tornar-se problemático e/ou indisponível este não afetará o bom funcionamento de todos os outros micro-serviços. Somente um micro-serviço N qualquer que depender de uma comunicação síncrona com micro-serviço A poderá sofrer algum impacto sobre alguma funcionalidade.
Para tentar evitar, ou reduzir, este problema da comunicação síncrona entre os micro-serviços é utilizado, sempre que possível, o recurso de comunicação assíncrona.
Para perceber melhor esta questão podem ser também observados na Figura 5 a existência de um Service Discovery, para comunicação síncrona, e de um Message Broker, para comunicação assíncrona, ambos recursos explicados a seguir.

Comunicação síncrona é aquela realizada diretamente entre dois micro-serviços em que há uma Request de um micro-serviço A para um micro-serviço B e é, geralmente, esperado a Response do micro-serviço B para o micro-serviço A. Esta comunicação é orquestrada pelo Service Discovery. Nesse ponto os micro-serviços são “cegos” entre si, ou seja, não conhecem o caminho (*Path* ou *URL*) completo do micro-serviço que possui a API que estão consumindo, apenas executam um pedido ao Service Discovery, informando qual é o micro-serviço e sua respectiva API que deseja executar e o obter uma resposta, quando esta for expectável.

O Service Discovery será o responsável por internamente realizar esta chamada e retornar a devida resposta entre os micro-serviços, e com o fato de que os micro-serviços podem possuir N instâncias a correr ao mesmo tempo em diferentes Servidores o Service Discovery também possui um recurso chamado de "*Load Balance*", sendo assim também responsável por equilibrar os pedidos feitos a um micro-serviço, com o intuito de evitar a sobrecarga de requisições sobre uma mesma instância, o que poderia criar um possível problema de performance.

Quando um micro-serviço A estiver indisponível, ou houver uma falha na rede de comunicação (Internet, Intranet ou extranet) o Service Discovery será o responsável por tratar esta falha retornando para um micro-serviço B, que tenha iniciado a requisição para micro-serviço A, uma resposta devida permitindo-o continuar seu pleno funcionamento com o conhecimento do status momentâneo de micro-serviço A. Geralmente nestes casos a resposta mais comum é a de "Serviço Temporariamente Indisponível".

A comunicação assíncrona é a responsável por atender a um dos requisitos necessários para que uma arquitetura seja considerada de micro-serviços, pois esta comunicação é realizada com base no conceito de arquitetura Orientada a Eventos (*Event-Driven Architecture*). No caso deste trabalho os eventos irão resultar na troca de Mensagens de Eventos (*Event Messages*) entre os micro-serviços. Essa estratégia é extremamente importante para o sucesso da arquitetura de micro-serviços ao garantir que uma regra de negócio seja iniciada e totalmente finalizada com sucesso no micro-serviço responsável por esta regra, prevenindo algum erro no processamento desta regra principal por causa de um outro micro-serviço.

Utilizando um exemplo prático implementado neste trabalho de dissertação temos:

- o micro-serviço DISCIPLINES é o responsável por processar e armazenar as regras de negócio e os dados das disciplinas. Uma das regras diz que a disciplina deve guardar para si quem são os professores que estão registrados para lecioná-la.
- o micro-serviço PROFESSORS é o responsável por processar e armazenar as regras de negócio e os dados dos professores. Uma das regras é atualizar um professor como Ativo ou Inativo na Universidade.

Dessa forma ao executar um processo para atualizar o Status de um professor de Ativo para Inativo, este processo inicia, é executado e é finalizado dentro do próprio micro-serviço PROFESSORS não havendo a necessidade de ir até o micro-serviço DISCIPLINES e aguardar que este execute e finalize seu processo internamente. O micro-serviço

PROFESSORS ao finalizar seu processamento irá publicar (*Publish*) uma mensagem no Message Broker realizando uma comunicação assíncrona para informar aos micro-serviços interessados que um professor foi atualizado. Nesse exemplo um interessado é o micro-serviço DISCIPLINES que irá consumir (*Subscribe*) esta mensagem do Message Broker e realizará seu processamento independentemente do micro-serviço PROFESSORS.

Através deste simples exemplo acima é possível perceber a importância do Message Broker com o desacoplamento de regras e processos de negócio dos micro-serviços, ou seja caso algum erro venha a ocorrer no micro-serviço DISCIPLINES nada irá impactar o processo bem sucedido no micro-serviço PROFESSORS. Isso nos garante também os benefícios de melhores manutenção e resiliência, que serão descritos mais adiante nesta dissertação.

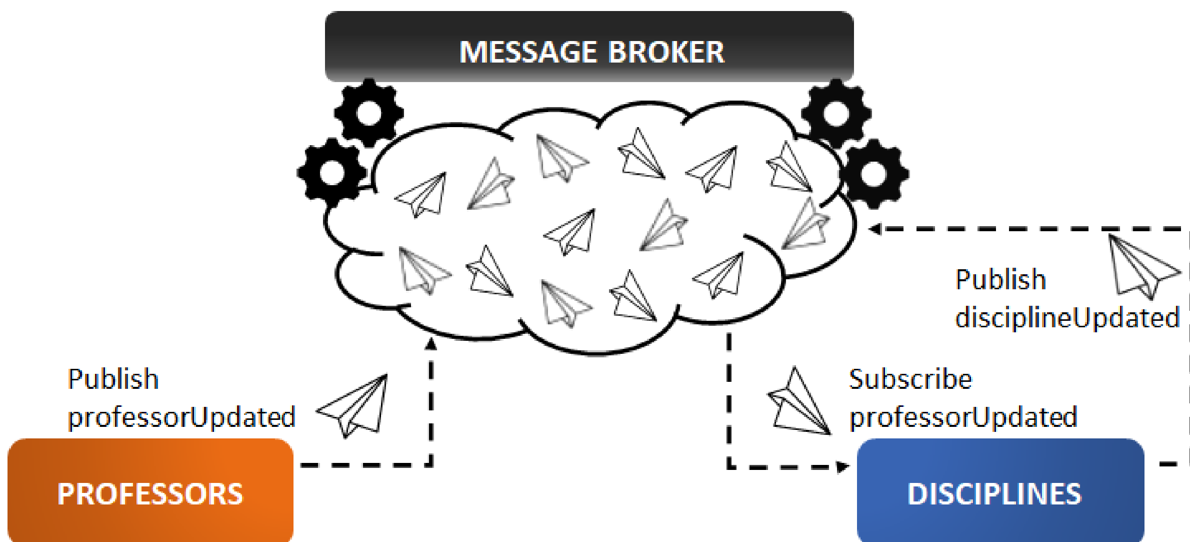


Figura 5 – Exemplo de atuação do Message Broker

A figura 5 acima demonstra o funcionamento do Message Broker utilizando o mesmo exemplo entre os micro-serviços PROFESSORS e DISCIPLINES.

Perceba que DISCIPLINES também realiza um Publish informando que uma disciplina foi atualizada, para que qualquer micro-serviço que seja interessado sobre essa informação possa consumi-la e utilizá-la.

2.6.4- A arquitetura Implementada

Após a descrição de como é estruturada e organizada uma arquitetura de micro-serviços, segue então o modelo desta para atender o modelo de negócio de gestão de alunos.

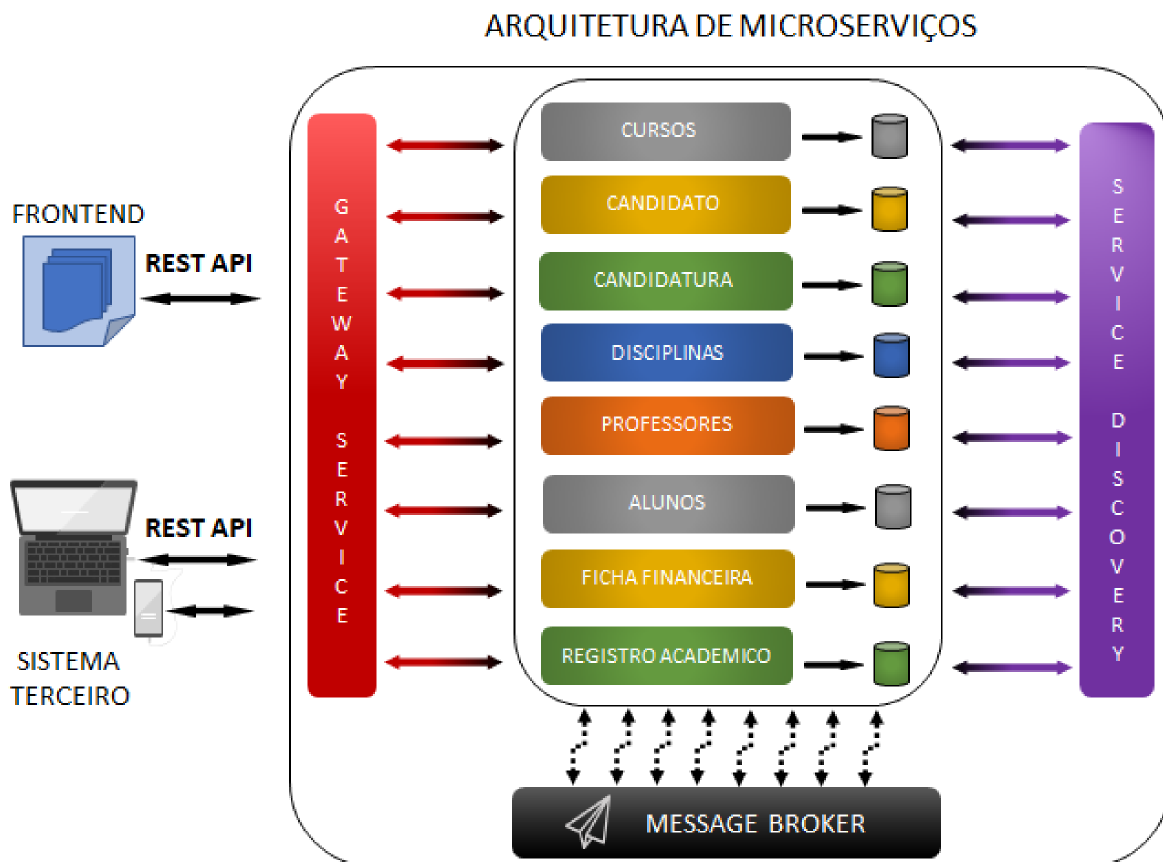


Figura 6 – Arquitetura de micro-serviços implementada para gestão de alunos

Na Figura 6 acima, temos então a arquitetura de micro-serviços planejada e estruturada para este trabalho de dissertação.

Pode ser percebido que a camada de Front End e um sistema Terceiro qualquer são independentes e caracterizados como componentes extras na arquitetura e passam por um serviço extra de *API Gateway (*ou *Gateway Server)* para utilizar as APIs dos micro-serviços, e também cada micro-serviço, ou regra de negócio, torna-se uma “Micro” aplicação de software única e independente de todas as outras aplicações e que representa uma “Micro” parte de um sistema Todo. Isso possibilita, por exemplo, que cada micro-serviço seja desenvolvido por equipes diferentes e até mesmo em Linguagens de programação diferentes. A base de dados nesta arquitetura não é somente uma e concentradora, mas são várias base de dados “Especialistas”, uma para cada micro-serviço, e também são *Bases de Dados Não Relacionais – NoSQL*. É possível verificar ainda as interfaces

primordiais de comunicação entre os micro-serviços, o *Service Discovery* e o *Message Broker*. O *Load Balancer* não é referenciado na figura por se tratar de um componente com o funcionamento interno dentro do Service Discovery.

A figura 7 abaixo apresenta como ficou a arquitetura sob a ótica das comunicações entre os micro-serviços desenvolvidos nesta dissertação.

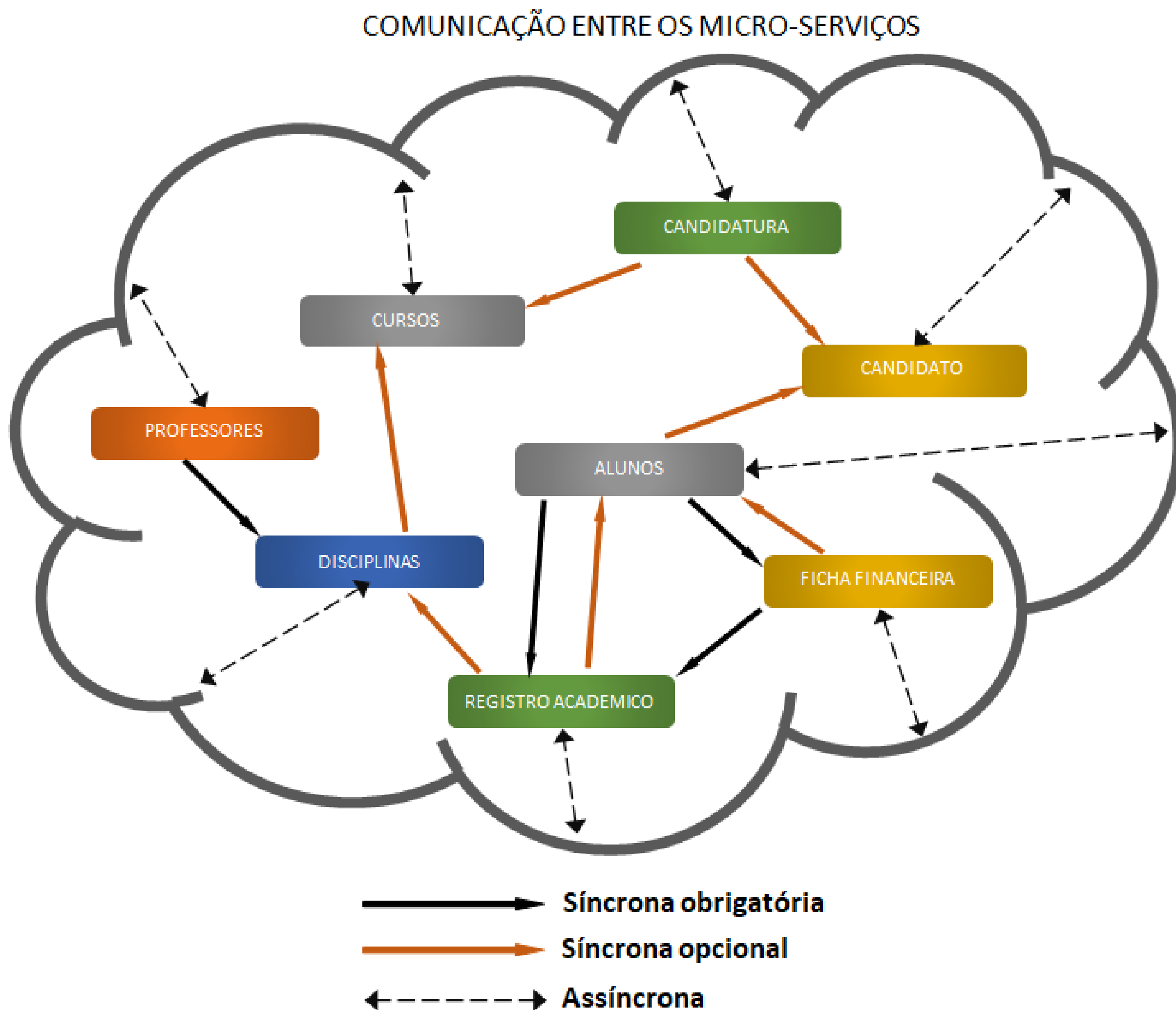


Figura 7 – Comunicação síncrona e assíncrona entre os micro-serviços

Dois relevantes pontos aqui são a existência de comunicação síncrona obrigatória e síncrona opcional. O micro-serviço STUDENTS será utilizado como exemplo destes casos.
1º- Ao criar um novo aluno, o micro-serviço STUDENTS vai ao micro-serviço APPLICANTS para verificar se o registro recebido de candidato existe ou não. Essa etapa é útil para

garantir que o candidato realmente exista, mas ela não é obrigatória para que o processo todo de criação de novo aluno prossiga com sucesso;
2º- Ao registrar uma nova disciplina para um aluno, o micro-serviço STUDENTS, onde o processo inicia, precisa obrigatoriamente acionar o micro-serviço REGISTRY_ACADEMICS pois este é o responsável por armazenar as disciplinas dos alunos, e REGISTRY_ACADEMICS vai ao micro-serviço DISCIPLINES verificar se a disciplina enviada é válida (existente) ou não.

É recomendável que seja, sempre que possível, utilizada a comunicação assíncrona ao invés da comunicação síncrona, contudo definir qual tipo deverá ser utilizado entre dois micro-serviços pode algumas vezes não estar nas mãos das equipes técnicas por motivo de regras de negócio já definida que não permita evitar a comunicação síncrona.

A arquitetura monolítica e a arquitetura de micro-serviços, são modelos arquiteturais maduros e consistentes, desenhados, testados e validados pela comunidade tecnológica e por milhares de empresas no mundo e que, portanto, são assim classificados, qualificados e definidos como tipos de arquitetura de software.

Capítulo 3 – A Computação em Nuvem

Pode-se definir a Computação em Nuvem (*Cloud Computing*) como “(...) fornecimento de serviços informáticos, incluindo servidores, armazenamento, bases de dados, rede, software, análises e inteligência, através da internet (‘a cloud’) para disponibilizar mais rapidamente inovação, recursos flexíveis e poupanças no dimensionamento.” (Microsoft, 2020). Ou seja, a computação em nuvem em resumo é um recurso de infraestrutura que tem o objetivo de ser uma infraestrutura de data centers mais dinâmica para as empresas através de serviços fornecidos por meio da Internet e também pode ser utilizado por diferentes dispositivos e plataformas como por exemplo PC, Tablet, Smartphone e até mesmo dispositivos de Internet das Coisas (IoT - *Internet of Things*).

3.1 – O início da Computação em Nuvem

A ideia de como funciona a Computação em Nuvem não é algo tão novo, é algo relativamente (à altura de escrita deste trabalho) antigo, que teve essa concepção iniciada por volta da década de 1950, pois os antigos computadores, os Mainframes, eram muito caros para quaisquer organizações terem em grandes quantidades. Desta forma seus funcionários (utilizadores do Mainframe) utilizavam estações conectadas a um Mainframe Central para realizar suas tarefas. Durante os anos seguintes na década de 1960 a então versão de uma “Computação em Nuvem” foi ganhando algumas adaptações e neste momento começou então a discussão sobre uma implementação do uso compartilhado do computador de forma simultânea, promovida pelo cientista da computação John McCarthy (Garfinkel, 2011) criando nesta altura o conceito de "Utility Computing”. Outro cientista que também é considerado um dos criadores do conceito de computação em nuvem foi Joseph Carl Robnett Licklider (ECPI University, 2020) que estudou novas ideias para fornecer recursos de computação através de uma Rede de Computadores, e que talvez não coincidentemente foi um dos fundadores da ARPANET.

Um marco para a Computação em Nuvem como temos hoje pode ser considerado o serviço *Hotmail* na década de 90, sendo pioneiro no uso de website para fornecer um serviço através de um aplicativo. Então na década de 2000 a Computação em Nuvem começa a ganhar força pelo investimento e início da oferta comercial para Armazenamento de dados,

processamento computacional, Virtualização otimizada, melhor controle de Aplicações web/remotas como serviços, Machine Learning e etc.

3.2 – Tipos e serviços de Computação em Nuvem

Há tipos de Computação em Nuvem e tipos de serviços diferentes a fim de serem fornecidos com mais assertividade como solução ideal de acordo com cada necessidade:

3.2.1 - Tipos de Computação em Nuvem

Pública (*Public Cloud*) - o serviço de nuvem é terceirizado e a administração dos serviços é toda feita por este, assim como a infraestrutura de hardware e software também pertence ao provedor terceirizado. Os serviços contratados são todos geridos através da Internet.

Privada (*Private Cloud*) - o serviço de nuvem é fornecido por uma única organização, em uma rede privada, e esta organização é a responsável pela infraestrutura utilizada. A organização possui seus próprios Data Centers locais e podem contratar serviços externos de outros fornecedores para serem alocados na Nuvem privada.

Híbrida (*Hybrid Cloud*) - combina os tipos de Nuvem Público e Privado compartilhando entre ambas a gestão dos recursos de infraestrutura e dados disponibilizados. Possui a flexibilidade de contratar serviços de Nuvem terceirizados de acordo com uma demanda específica e também ampliar, ou melhorar, um serviço de Nuvem próprio ao identificar uma necessidade mais duradoura de recursos.

3.2.2 - Tipos de serviços de Computação em Nuvem

Infraestrutura como um serviço (IaaS - Infrastructure as a Service) - categoria que possui as componentes mais básicas dos serviços de infraestrutura de TI na Nuvem. É contratada de forma a oferecer um flexível controle dos recursos necessários de acordo com a demanda.

Plataforma como um serviço (PaaS - Platform as a Service) - categoria na qual a gestão do ambiente e infraestrutura são feitas pelo provedor do serviço de nuvem de forma a garantir por exemplo a disponibilidade dos recursos contratados, de serviços, backup etc. Comumente utilizada para o ciclo de desenvolvimento de aplicações e de forma que o foco do cliente seja, quase, somente na funcionalidade de suas aplicações.

software como um serviço (Saas - software as a Service) - categoria considerada a solução mais completa na qual o fornecedor do serviço de Nuvem é o responsável pela alocação e gestão das aplicações de software e da infraestrutura, bem como de toda a manutenção necessária sobre ambas. Estas aplicações, ou módulos de software, são de facto serviços fornecidos, muitas vezes, a utilizadores finais (como por exemplo um serviço de Webmail), que possuem até mesmo algum recurso de administrador para uma autogestão da aplicação e seus utilizadores.

A Computação em Nuvem também pode ser considerada como um dos maiores estímulos e motivos para o avanço e disseminação da arquitetura de micro-serviços que ocorrem tanto sobre estudos, pesquisa e investigação, pela comunidade científica tecnológica, quanto também pela adesão e investimento das organizações em todas as áreas de atuação. Algumas características específicas que diferenciam a arquitetura de micro-serviços da arquitetura monolítica são ainda mais realçadas e otimizadas pela Computação em Nuvem quanto às propriedades de escalabilidade, manutenção e resiliência tornando a arquitetura de micro-serviços mais recomendada do que a arquitetura monolítica, de acordo com o cenário do modelo e regras de negócio apresentado. Deve ser ponderado ainda que “mais recomendada” não deveria ser interpretada como a “única recomendada”.

3.3 – A importância da Computação em Nuvem

A Computação em Nuvem trouxe para a gestão técnica e desenho técnico significativas melhorias, otimizações e novas possibilidades tecnológicas de infraestrutura para todos os tipos de arquiteturas de software, principalmente ao reduzir a complexidade sobre a gestão da infraestrutura de hardware e as propriedades de escalabilidade, manutenção e resiliência de um software. Essas significativas melhorias no lado tecnológico se refletiram também no lado da gestão de negócios das organizações permitindo a criação de novos, ou a melhoria

de, produtos e/ou serviços uma vez que o lado tecnológico passou a estar ainda mais flexível para atender as demandas de negócio organizacionais.

Alguns benefícios da Computação em Nuvem são:

- agilidade de acesso a grande variedade de tecnologias/software, conforme a necessidade, sem precisar instalá-los na máquina do utilizador;
- aumento de produtividade na gestão e manutenção dos recursos de infraestrutura com a redução física de hardwares e de Data Centers para serem mantidos;
- construção e manutenção de uma estrutura tecnológica em Escala Global, com o dimensionamento específico de cada região, em minutos;
- avançadas políticas e estratégias de segurança para uso do ambiente contratado;
- utilização somente de recursos por demanda de necessidade (armazenamento, processamento, memória), sem provisionamento excessivo;
- optimização na capacidade de processamento de forma a oferecer escalabilidade horizontal ao disponibilizar assim centenas (ou milhares) de processadores, em minutos, sem dificuldades.

Todos estes benefícios listados anteriormente resultarão no principal benefício buscado pelas organizações, que é o aumento da eficiência na relação custo/benefício para implementarem suas gestão e estratégias de negócios. Por este motivo a computação em nuvem tornou-se importante no meio corporativo, até mesmo como um dos fatores de competitividade ao permitir que micro e pequenas empresas pudessem competir contra grandes empresas, sendo então muito pesquisada, avaliada e implementada para otimização da infraestrutura tecnológica de todos os tipos de arquiteturas de software implementadas pelas organizações.

Entretanto, algumas perguntas ainda passaram a ser, naturalmente, feitas tal como, É possível obter e aproveitar ainda melhor os benefícios da infraestrutura disponibilizados e suportados pela Computação em Nuvem? É possível desenvolver software otimizado capazes de aproveitar melhor a escalabilidade, manutenção e resiliência suportados pela Computação em Nuvem? A resposta para estas perguntas é: - sim, é possível, utilizando a programação reativa.

O capítulo a seguir irá descrever as características de escalabilidade, manutenção e resiliência das arquiteturas de software monolítico e de micro-serviços sendo seguido por um outro capítulo que irá descrever os tipos de programação bloqueante e reativa que podem ser utilizados para desenvolver software.

Capítulo 4 – Escalabilidade, manutenção e resiliência

4.1 – Escalabilidade de software

A escalabilidade pode ser entendida como a capacidade de algo em ter seus recursos ampliados ou reduzidos de acordo com uma necessidade, seja esta uma necessidade momentânea ou não. Por exemplo, na época do Natal em que o consumo aumenta expressivamente uma loja pode decidir “escalar” seu poder de venda aumentando seu número de vendedores e/ou aumentando a quantidade de produtos em estoque para pronta entrega sem que o consumidor tenha de esperar pelo produto. Após o Natal o estoque retorna a capacidade normal e os vendedores temporários passam a ser dispensados.

A escalabilidade de software segue o mesmo conceito o qual seria então a capacidade de um software em ter seus recursos de processamento e de memória ampliados ou reduzidos de acordo com uma necessidade, momentânea ou não. Usando a mesma lógica anteriormente da época do Natal e uma loja online, temos neste caso o software de pesquisa e escolha de produtos online que poderíamos “escalar” a capacidade de suportar um número maior de visitantes ao mesmo tempo, saindo de centenas em épocas comuns para milhares em época do Natal, e também “escalar” a capacidade processamento de autorização, validação, cálculo de pagamentos. Após o Natal os recursos de processamento e memória utilizados momentaneamente a mais podem ser reduzidos para a capacidade normal.

A seguir é demonstrado como se aplica o conceito de escalabilidade nas arquiteturas de software monolítico e de micro-serviços.

4.1.1 – Escalabilidade de arquitetura monolítica

Em uma necessidade de escalabilidade do software na arquitetura monolítica temos a seguinte estrutura a seguir:

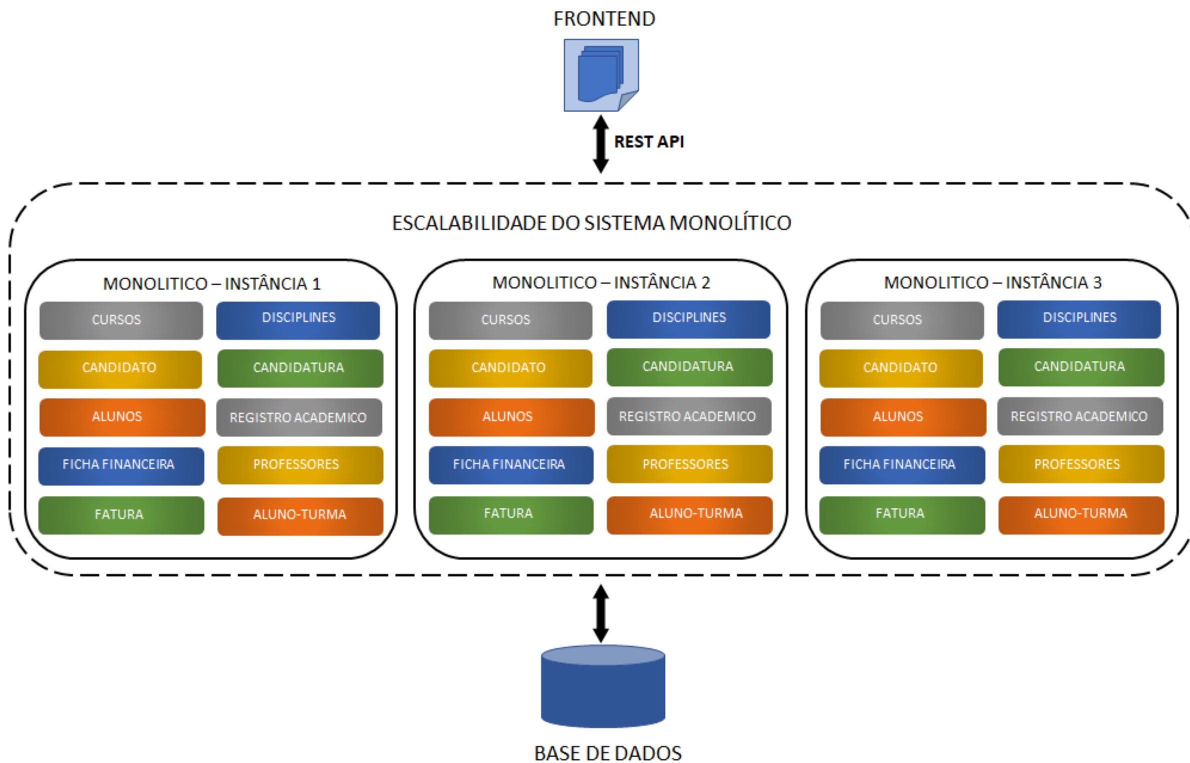


Figura 8 – Escalabilidade da arquitetura monolítica

A Figura 8 acima nos apresenta um modelo de como funciona a escalabilidade de um software de arquitetura monolítica onde para cada instância nova criada no servidor para o software todas as regras de negócio são criadas juntas, ou seja, todo o software é criado/instanciado. Isso significa que mesmo com uma necessidade de aumentar os recursos de processamento e de memória momentaneamente por causa de somente uma regra/processo de negócio todo o sistema deverá será escalado, ou seja, o custo da escalabilidade poderá dessa forma até mesmo ser totalmente multiplicada de acordo com a quantidade de instâncias criadas a mais.

Nota-se também que mesmo que haja mais do que uma instância, a base de dados não segue este número de quantidade de instâncias/componentes. A base de dados continua sendo somente uma. De acordo com a necessidade também pode haver mais do que uma base de dados no projeto, mas não é necessário haver uma instância de base de dados para cada Instância das regras de negócio do software.

4.1.2 – Escalabilidade de arquitetura de micro-serviços

Numa modelagem de escalabilidade, a arquitetura de micro-serviços possui a estrutura seguinte:

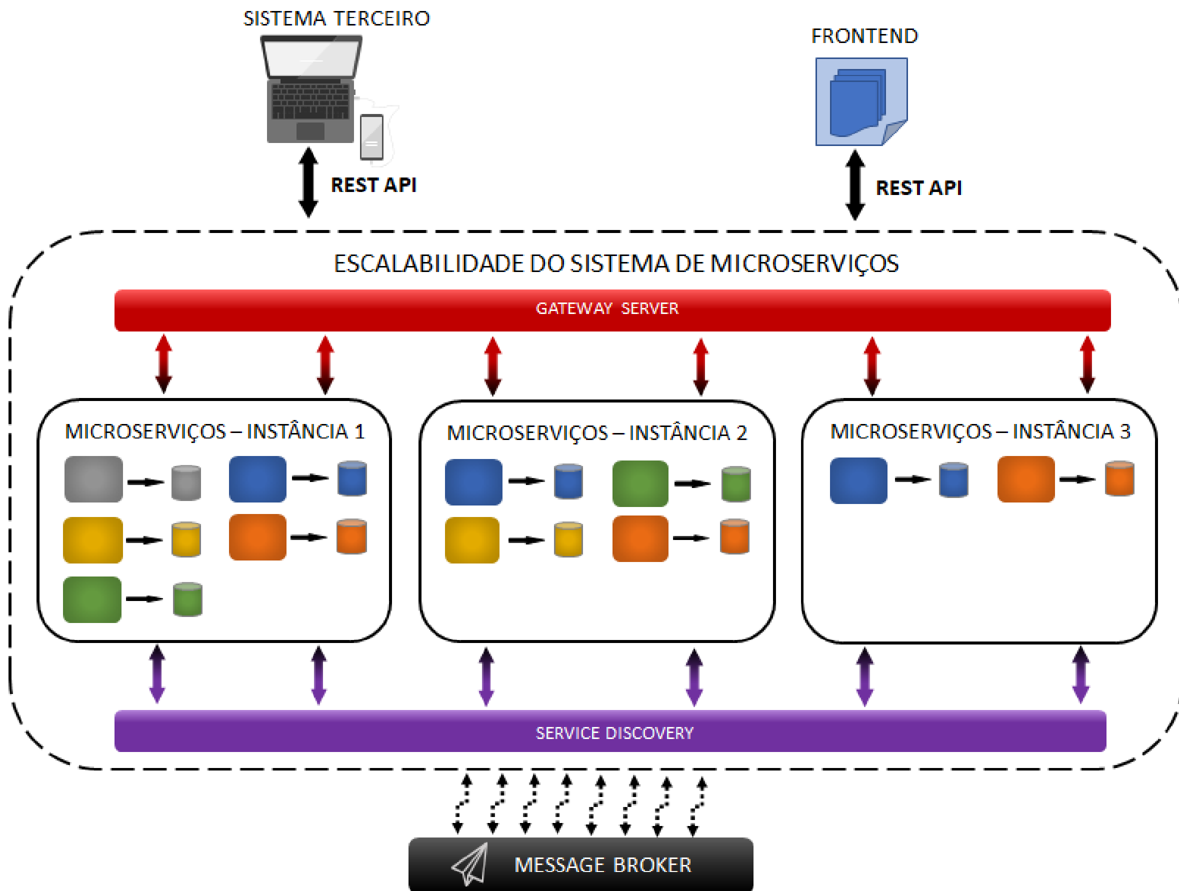


Figura 9 – Escalabilidade da arquitetura de micro-serviços

A Figura 9 acima nos apresenta um modelo de como funciona a escalabilidade na arquitetura de micro-serviços onde para cada instância nova criada no servidor podemos ter todos ou apenas um micro-serviço escalado. Dessa forma temos somente o micro-serviço que possua a regra/processo de negócio, que realmente é importante e crítico, em um determinado espaço de tempo, escalado e assim ampliando os recursos de processamento e ou memória para somente este obtendo a redução do curso com a escalabilidade.

Nota-se também que mesmo que haja mais do que uma instância, a base de dados não segue este número de quantidade de instâncias/componentes. A base de dados continua sendo somente uma. De acordo com a necessidade também pode haver mais do que uma base de dados no projeto, mas não é necessário haver uma instância de base de dados para cada Instância das regras de negócio.

4.2 – Manutenção de software

A manutenção é uma etapa do Ciclo de Vida de um software na qual podem ser dois tipos: manutenção corretiva, na qual são feitas correções no sistema sobre erros (*bugs*) encontrados no mesmo, ou manutenção evolutiva, na qual são desenvolvidas novas funcionalidades, regras de negócio e etc.

A seguir é descrito como se aplica a etapa de manutenção sobre ambas as arquiteturas.

4.2.1 – Manutenção da arquitetura monolítica

A manutenção corretiva ou evolutiva de um software monolítico muitas vezes pode tornar-se em um trabalho complexo, pelo fato do alto acoplamento e relacionamento das regras de negócio e componentes internas da aplicação, como já descrito anteriormente, pois essa manutenção pode se traduzir em alterar um ponto A da aplicação que possui uma alta probabilidade de impactar diretamente um ponto B da aplicação de forma não intencional e esta tarefa pode tornar-se difícil de ser dividida e/ou compartilhada entre os membros da equipa de desenvolvimento.

Por exemplo: no sistema de gestão de alunos proposto a entidade ALUNO possui uma relação direta com as entidades CANDIDATO, FICHA FINANCEIRA e REGISTRO ACADÊMICO. Essa relação envolve tanto o âmbito da modelagem relacional da base de dados quanto o desenvolvimento e programação das regras de negócio.

Um exemplo prático desenvolvido neste trabalho pode ser analisado através do Diagrama de Sequência a seguir sobre a regra de negócio abaixo para registar um novo aluno:

1. Identificar o registo do candidato para este aluno;
2. A informação de candidato é obrigatória para criar o novo aluno;
3. Ao criar um novo aluno, sua ficha financeira deve ser criada automaticamente;
4. Ao criar um novo aluno e sua ficha financeira, seu registro acadêmico deve ser criado automaticamente;

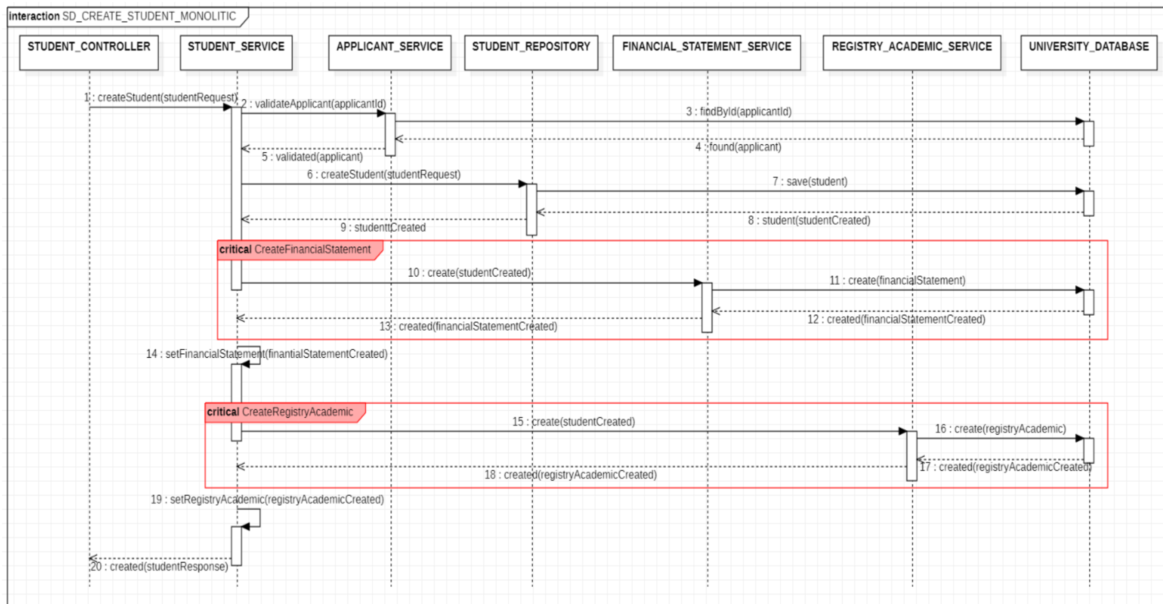


Figura 10 – Diagrama de Sequência para criar um novo aluno em arquitetura monolítica

Como a regra de negócio principal é definida pela criação de um novo aluno então o diagrama é apresentado no âmbito da entidade de negócio aluno (*Student*), como pode ser percebido pelo nome do primeiro objeto de iteração do diagrama *StudentController*, e o/a método/função *createStudent*(*StudentRequest*) representa o início deste processo.

Como pode ser visto pelo diagrama na Figura 10 uma, aparentemente, simples regra de negócio acaba se tornando um processo um pouco mais complexo e com pontos críticos sobre a sua execução com sucesso em uma arquitetura monolítica.

Após as etapas para verificar a validade do candidato (*Applicant*) previamente criado e a criação do aluno (*Student*) de facto na base de dados serem realizadas com sucesso a transação aberta para a criação do registo do novo aluno poderá sofrer um cancelamento ou desfazimento ("*Rollback*") caso ocorra algum erro posteriormente nos processos de criação da ficha financeira ou do registro acadêmico. O "*Rollback*" também poderá ocorrer sobre as transações para a criação de uma ficha financeira na base de dados, após criação do aluno com sucesso, caso ocorra algum erro no processo de criação do registro acadêmico. Desta forma a criação de uma ficha financeira e de um registro acadêmico tornam-se processos críticos para o sucesso completo de todas as etapas de criação de aluno pois estão diretamente, e fortemente, ligados ao processo para a criação de aluno em si, ou seja, tornam-se então, também, pontos Críticos e altamente Sensíveis durante sua manutenção evolutiva ou corretiva, uma vez que durante a execução do código de qualquer uma destas

etapas caso ocorra algum erro na criação de um novo aluno será diretamente impactada e afetada mesmo sem que esta em nada tenha sido alterada no âmbito específico do código de aluno.

Esse Diagrama demonstra então como processos e regras de negócio em uma arquitetura de software monolítico podem ser altamente Acoplados e Dependentes.

4.2.2 – Manutenção da arquitetura de micro-serviços

A manutenção corretiva ou evolutiva de um software de micro-serviços muitas vezes tornar-se em um trabalho mais simples, pelo fato do baixíssimo, ou quase nenhum, acoplamento e relacionamento das regras de negócio e das aplicações, como já descrito anteriormente, pois mesmo que a alteração de um micro-serviço A impacte diretamente em um micro-serviço B de forma não intencional poderá haver ainda assim a divisão dessas tarefas de manutenções entre os membros da equipa de desenvolvimento, ou até mesmo serem equipas diferentes trabalhando cada um sobre um micro-serviço especificamente. Cada um membro da equipa, ou cada equipa, trabalhará sobre seu próprio micro software sem interferir no outro, sendo necessário apenas definirem como será a interface de comunicação entre os micro-serviços, ou seja os objetos de Request e Response. Se esta for uma comunicação síncrona poderá será definida e/ou atualizada por exemplo a API REST a ser utilizada para tal com as devidas assinaturas de Request e Response, e se for uma comunicação assíncrona será definido e/ou atualizados os processos de Publicação e Subscrição (*Publishing* e *Subscribing)* sobre as mensagens e canais de comunicação via Message Broker.

Por exemplo: no sistema de gestão de alunos proposto o Microservice ALUNOS possui uma Ação direta para validação no micro-serviço CANDIDATO e Reações indiretas para atualizar um aluno com dados publicados pelos micro-serviços FICHAS FINANCEIRAS e REGISTROS ACADEMICOS.

Desse modo podemos utilizar o mesmo exemplo prático com as mesmas regras de negócio para criação de alunos como utilizado no tópico anterior sobre a arquitetura monolítica, analisando através do Diagrama de Sequência:

1. Identificar o registo do Candidato para este aluno;
2. A informação de Candidato é obrigatória para criar o novo aluno;

3. Ao criar um novo aluno, sua ficha financeira deve ser criada automaticamente;
4. Ao criar um novo aluno e sua ficha Financeira, seu registro acadêmico deve ser criado automaticamente;

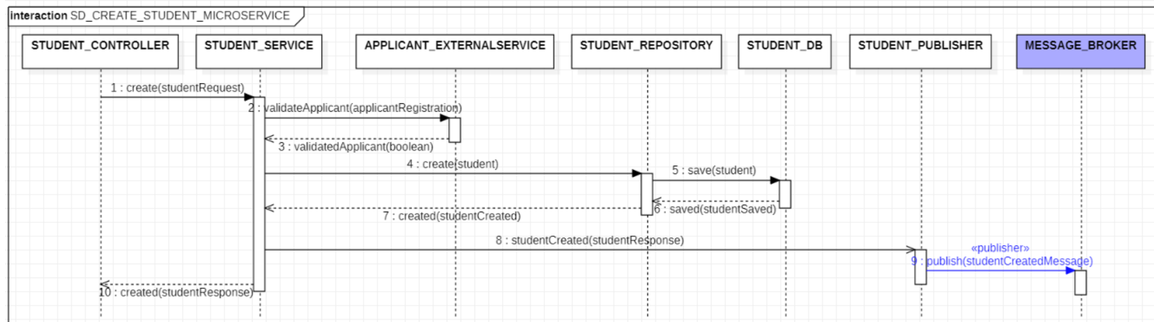


Figura 11 – Diagrama de Sequência para criar um novo aluno em arquitetura de micro-serviços

Como a regra de negócio define a Criação de um novo aluno, então o diagrama é apresentado no âmbito do micro-serviço de alunos (*Students*) e o processo é iniciado através do/da método/função *create*(*StudentRequest*).

O objeto da iteração chamado *Applicant_External_Service* não possui nenhum código ou regra de negócio sobre a validação de um Candidato, este possui apenas uma comunicação externa, e síncrona, com o micro-serviço de Candidatos (*Applicants*) pedindo a este que valide a informação do Candidato então no diagrama não há nenhum detalhamento da etapa referente ao processo de validação do Candidato, nem mesmo a base de dados de Candidatos é demonstrada neste diagrama, significando que esta responsabilidade é exclusivamente do micro-serviço de Candidatos.

Importante realçar também que a única e especificamente base de dados demonstrada é sobre alunos, e não mais sobre inúmeras entidades de negócio e seus inúmeros relacionamentos umas com as outras, ou seja, uma Base de Dados mais coesa e menos acoplada.

Nota-se então o ponto mais importante sobre a manutenção em micro-serviços, que as regras 3 e 4 não serão atendidas no âmbito do micro-serviço alunos, pois ambas as regras para criar Fichas Financeiras e Registros Acadêmicos seguirão o processo de comunicação assíncrona (descrito no Capítulo 1, tópico 1.2.3), ou seja, o Micro-serviço alunos, após criar e salvar o aluno em sua base de dados, publica um Evento de Mensagem no *Message*

Broker com o aluno criado disponibilizando-o para quem interessado for sobre o resultado deste processo. Os micro-serviços de Fichas Financeiras (*FinancialStatements*) e de Registros Acadêmicos (*RegistryAcademics*) Subscrevem e Consomem esta Mensagem de alunos, dando prosseguimento ao seu próprio processamento internamente. Caso algum erro e/ou problema ocorra em uma futura manutenção ou alteração, corretiva ou evolutiva, nos micro-serviços de Registros Acadêmicos ou Fichas Financeiras não haverá impacto ou efeito colateral sobre as regras e processamento no âmbito de alunos. O micro-serviço de alunos também deverá possuir para si próprio algumas informações da ficha financeira e do registro acadêmico criados, mas este será um processo também Assíncrono para atualizar o aluno, pois ao serem criados a ficha financeira e o registro acadêmico também irão publicar uma mensagem no *Message Broker* que será Subscrita e Consumida pelo Micro-serviço alunos. Caso ocorra algum problema sobre os micro-serviços de Registros Acadêmicos ou Fichas Financeiras durante seus processo de criação o novo aluno criado não irá possuir essas informações uma vez que não haverá nenhuma mensagem a ser consumida sobre ambos, entretanto o micro-serviço de alunos não irá parar de funcionar. No momento em que os micro-serviços de Registros Acadêmicos e Fichas Financeiras voltarem a correr sem problemas e Publicarem seus Eventos de Mensagem, o Micro-serviço de alunos irá então Subscrever estas mensagens e atualizará os alunos que ainda não possuem atualizados os dados da ficha financeira e do registro acadêmico.

Vejamos na Figura 12 abaixo um exemplo da estrutura de como seriam essas iterações entre os micro-serviços onde para comunicação assíncrona os quadrados Azuis representam um processo de *Publisher*, os quadrados Verdes representam um processo de *Subscriber* e para a comunicação síncrona os quadrados Cinzas escuros representam um processo via API REST.

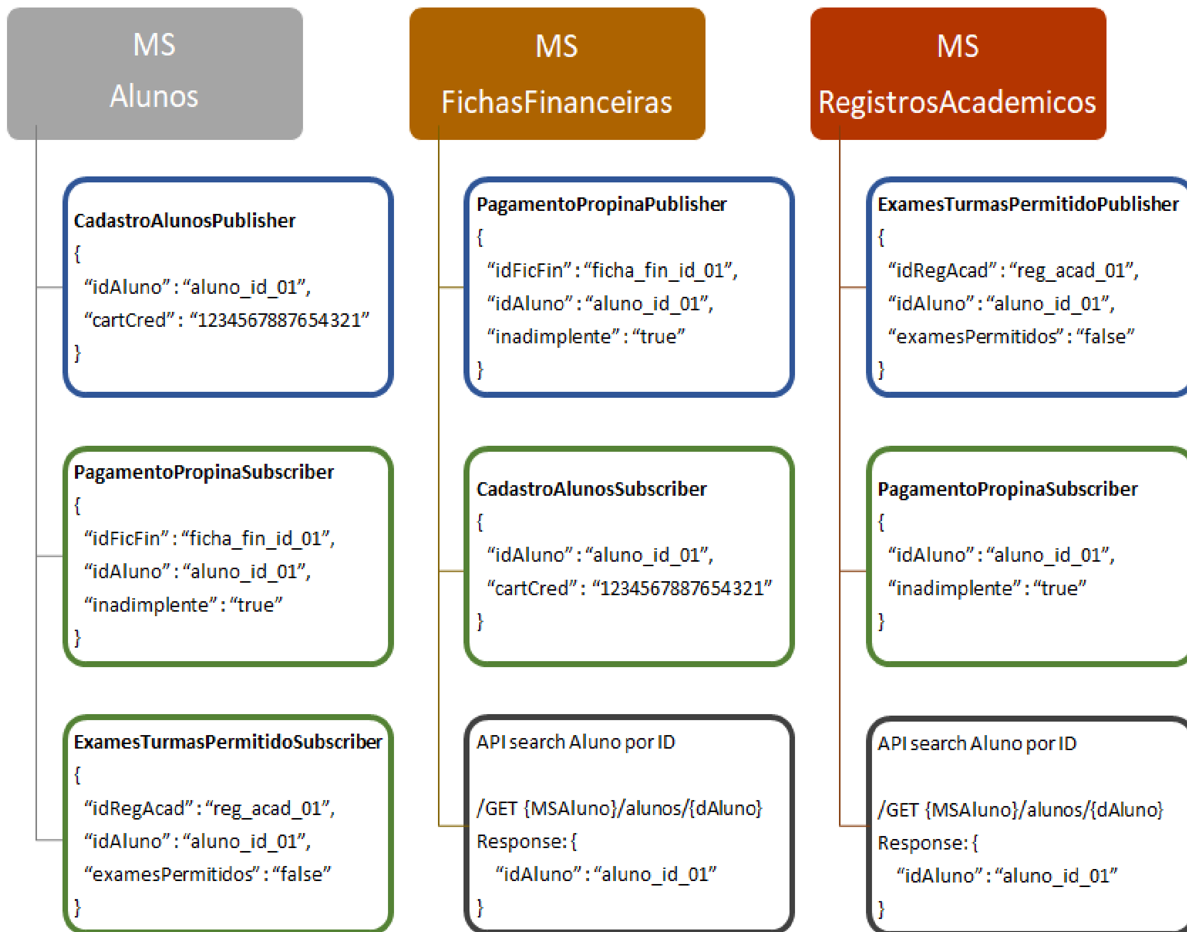


Figura 12 – Comunicação assíncrona (via Publishers e Subscribers) e síncrona (via API Rest)

Observe que uma mensagem publicada (Publisher) pode ser reconhecida como uma mensagem consumida (Subscriber) utilizando os nomes de cada uma delas, antes mesmo de comparar o corpo do objeto transmitido. Este é um exemplo de estratégia que auxilia na manutenção e testes das trocas de mensagens entre os micro-serviços.

A API REST para buscar um aluno pelo ID é importante no processo de Validação e Checagem da identificação do aluno recebido na mensagem. Assim há a certeza de que tudo está corretamente válido como se espera. Não é uma etapa obrigatória dentro da comunicação entre os micro-serviços durante um processo de Subscrição/Consumo de Menagem, mas fortemente recomendado.

4.3 – Resiliência de software

Usando a definição de resiliência de um dicionário, "[Física] Propriedade de um corpo de recuperar a sua forma original após sofrer choque ou deformação.", [Figurado] Capacidade

de superar, de recuperar de adversidades." (Priberam, 03/04/2021), podemos então afirmar que a resiliência de software é a capacidade deste de se Recuperar (voltar a responder) de um problema que o tenha feito parar de funcionar (responder), recuperando também seu Estado (seus Dados) antes de sua parada.

A resiliência de software é importante porque impacta diretamente no risco de indisponibilidade do software, que podem ser riscos muito críticos ou pouco críticos, de acordo com o âmbito do negócio desenvolvido, como até mesmo problemas Financeiros e/ou Jurídicos às empresas.

Seguindo o Fluxo, ou processo, de desenvolvimento de software, no qual possui as etapas de desenvolvimento (*Development*), Construção (*Build*), Testes (*Tests*) e Implantação (*Deploy*) a característica de resiliência de um software está correlacionada com, principalmente mas não somente, a etapa de Implantação do software pois uma vez que esteja implantado e a correr em ambiente de Produção um software que sofra ou apresente quaisquer avarias ou problemas que o impossibilite de funcionar e/ou responder deverá ter seu processo de Implantação executado novamente.

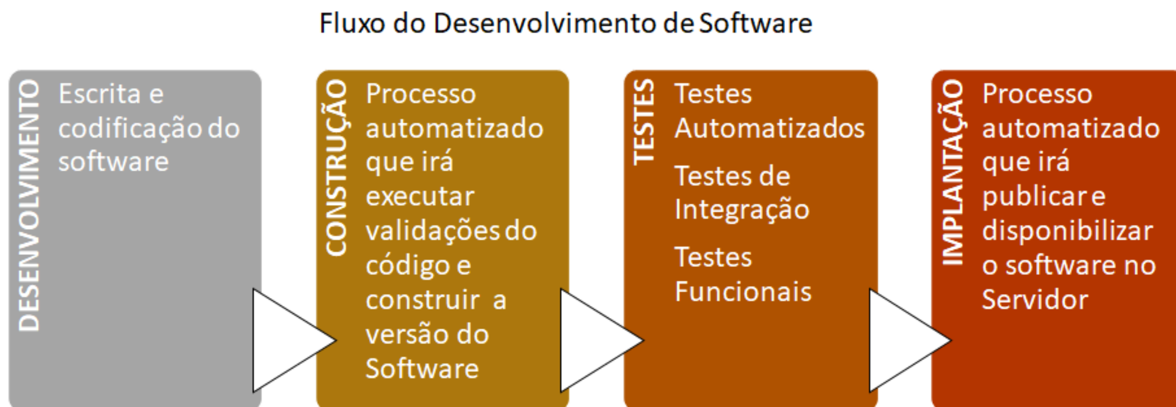


Figura 13 – Fluxo de desenvolvimento de um software

Perceba que este fluxo não se trata do *Ciclo de Vida do desenvolvimento de software*, pois este é mais amplo e possui no mínimo mais uma etapa chamada *Manutenção: corretiva e evolutiva* após a implantação e como o próprio nome sugere é um processo *cíclico.* O fluxo de desenvolvimento de um software visa demonstrar uma tarefa de desenvolvimento como um processo definido com Início (o desenvolvimento), Meio (a construção e os testes) e Fim (a Implantação).

A seguir são descritos os dois pontos, a resiliência e o risco de indisponibilidade, sobre as arquitetura monolítica e de micro-serviços e serão utilizados dois exemplos de cenários hipotéticos, os mesmos, para ambas as arquiteturas.

- Exemplo 1: imagine um cenário no qual durante a Inscrição de um aluno novo do estrangeiro, para o programa de Intercâmbio Universitário, o sistema da Universidade A precise se comunicar com o sistema da Universidade B deste aluno novo no estrangeiro e que durante esse processo ocorre então uma falha de comunicação não prevista, e por tanto não tratada, entre ambos os sistemas destas Universidades.
- Exemplo 2: a base de dados, ou o servidor da base de dados, sofre um problema e esta para de funcionar.

4.3.1 – Resiliência da arquitetura monolítica

Nesta arquitetura a resiliência e o risco de indisponibilidade do software possuem um peso muito grande sobre a manutenção do mesmo, isso porque como vimos no Capítulo 1, tópico 1.1, o software monolítico possui características que o torna um software considerado “grande” e “concentrador” por possuir todas as regras de negócio em um só software.

Um software “grande” provavelmente irá exigir um *maior tempo* de processamento para a sua etapa de Implantação e re-publicação/disponibilização, neste sentido uma Reimplantação (*Redeploy*). Ou em um cenário mais grave também pode haver a necessidade de uma Reconstrução (*Rebuild*) do software antes de sua re-implantação, o que poderia aumentar em muito mais tempo do que o desejado, ou aceitável, o risco de não o ter o software disponível novamente em ambiente de Produção.
Outra questão é sobre a característica de ser um software “concentrador”, que não possui tolerância a falhas e imprevistos por nenhum ponto de funcionamento, ou seja, qualquer regra e/ou processo de negócio, e/ou código desenvolvidos, são críticos para o funcionamento do software e podem levá-lo a uma falha e indisponibilidade como Todo.

Uma solução adotada por muitas empresas para reduzir e/ou dirimir esses riscos é a utilização de uma Implantação a mais do software em outro Servidor (“*Instância Espelho*”), espelhando exatamente sua versão atual e sua base de dados, dessa forma ao ocorrer uma indisponibilidade do software será feita uma troca de contexto para o servidor e software espelho que irá o tempo, e o risco, de indisponibilidade do software para os utilizadores.

Entretanto, esta solução cria um Custo em Dobro para a manutenção do software reduzindo a eficiência da relação custo/benefício para as empresas.

Seguindo os exemplos dados:
- Exemplo 1: o sistema da Universidade A obterá uma falha em um ponto específico sobre o serviço de Secretaria, mas que por comprometer Todo o sistema, irá afetar por exemplo o pagamento de Inscrições em Disciplinas ou Propinas que são serviços do Financeiro e o lançamento das notas dos alunos pelos professores nas Disciplinas que é um específico serviço do registro acadêmico.
- Exemplo 2: Neste cenário, todo o sistema para de funcionar juntamente com sua base de dados, até que esteja seja recuperada e/ou reinicializada novamente.

4.3.2 – Resiliência da arquitetura de micro-serviços

Nesta arquitetura a resiliência e o risco de indisponibilidade do software, como um todo, possuem um peso muito menor sobre a manutenção deste, isso porque como vimos no Capítulo 1, tópico 1.2, o software baseado em micro-serviços possue características que o tornam “micro” e “desacoplado” a possuir somente uma regra, ou entidade, de negócio específica com sua funcionalidade independente de outros, o que por consequência torna o software como um todo um *Sistema Distribuído*.

Um software “micro” irá exigir um *menor tempo* de processamento para a etapa de Re-implantação ou, também, de reconstrução antes de sua reimplantação, caso sejam necessárias ambas, reduzindo o risco de não o ter o software disponível novamente em ambiente de produção em tempo desejado e aceitável.
Outra questão, benéfica e importante, sobre os micro-serviços é que por serem “desacoplados” e independentes eles tornam a arquitetura muito mais tolerante a falhas e imprevistos de qualquer tipo (códigos ou ambientes interno e externo) evitando uma parada e indisponibilidade que comprometa a operacionalidade de um software como um Todo.

Por ser um “micro” software que exige um menor tempo de processamento para sua Implantação não há, geralmente, a utilização de uma “Instância Espelho”, pois a inicialização e disponibilização de uma nova instância do Micro-serviço ocorre, geralmente, em tempo hábil suficiente para que a indisponibilidade não ocorra em um nível crítico. Entretanto, quando não há margem de tolerância, de minutos ou segundos, sobre a

indisponibilidade do serviço por motivo da Alta criticidade da regra e processo de negócios implementado ter uma “Instância Espelho” para somente o Micro-serviço específico não torna o Custo em Dobro para o sistema como um Todo, melhorando assim a eficiência da relação custo/benefício para as empresas.

Contudo, apesar de a resiliência e o risco de indisponibilidade na arquitetura de micro-serviços serem menores e otimizados do que na arquitetura monolítica existe por outra lado o aumento na complexidade do controlo e da gestão sobre a etapa de Implantação pois muda-se de um cenário com apenas 1 software e 1 base de dados, com provavelmente poucas iterações para Implantar e Desplantar novas instâncias, para um cenário de N “Micro” componentes de software e N Bases de Dados, com possivelmente muitas iterações para Implantar e Desplantar novas instâncias. Dessa forma o controle e gestão da Implantação dos micro-serviços torna-se improvável de ser feito manualmente, sendo assim, o software, e/ou ferramentas, para auxiliar/automatizar a *Virtualização de Servidores* também vem evoluindo e sua utilização tornou-se obrigatória. Por exemplo, o software *Docker Engine*, utilizado neste trabalho de Dissertação e que será apresentado no tópico 4.2.1.

Seguindo os exemplos dados:
- Exemplo 1: o sistema da Universidade A obterá uma falha em um ponto específico sobre o serviço de Secretaria, entretanto, por ser desacoplado, somente o Micro-serviço possuidor deste processo irá ser afetado. Os micro-serviços da ficha financeira e do registro acadêmico continuarão a funcionar normalmente, deste modo o pagamento de Inscrições em Disciplinas ou Propinas e o lançamento das notas dos alunos pelos professores nas disciplinas não serão impactados.
- Exemplo 2: neste cenário somente o Micro-serviço possuidor desta base de dados irá ser afetado até que a base de dados seja recuperada e/ou reinicializada novamente. O sistema como um todo continuará a funcionar normalmente para todos os outros micro-serviços.

Capítulo 5 – Programação bloqueante e programação reativa

5.1 – A tecnologia Java e o Framework Spring

Como o próprio dono e mantenedor da tecnologia Java informa em seu web site, "Java is the #1 programming language and development platform." (Oracle, 2021), isso significa que a tecnologia Java por ser uma plataforma é disponibilizado para a comunidade profissional, além do que somente a linguagem de programação, uma série de outros Recursos e soluções para todo o ciclo de vida de um software cobrindo assim o desenvolvimento, teste, disponibilização e manutenção do software.

A Linguagem de programação Java pode ser executada independente da plataforma (laptops, PC, telemóveis, servidores, dispositivos IoT, etc) e independentemente do sistema operativo (Windows, Linux, Unix, Android, Ios, etc) porque ela será interpretada através de seu interpretador multiplataforma chamado *Java Virtual Machine – JVM* (Máquina Virtual Java). Após ser instalada, a JVM irá atuar como uma *Inteface/Middleware* responsável por interpretar e executar os códigos Java compilados na plataforma e sistema operacional, dessa forma o planeamento e desenvolvimento do software desejado em Java não precisa se preocupar em criar versões, códigos e estratégias de manutenção diferentes para cada plataforma e sistema operativo de forma específica. Um único código desenvolvido, uma única compilação e executável em qualquer lugar.

Algumas outras características, além de ser multiplataforma, que tornaram Java uma tecnologia mundialmente tão difundida e adotada são, por exemplo:

- Garbage Collector, recurso da JVM responsável pelo autogestão de memória, reduzindo a complexidade do desenvolvimento;
- Programação Orientada a Objetos - POO, o que permite desenhar, planear e escrever códigos com uma melhor compreensão próximo à questão "Mundo Real" que pretende-se desenvolver uma solução;
- Grande número de *Libraries* (bibliotecas) e *Frameworks* desenvolvidos e disponibilizados em iniciativas independentes da Oracle, a fim de contribuir para melhorar e aumentar a produtividade do ciclo de vida de um software.

Dentre estes Frameworks, foi escolhido para este trabalho o Spring, da Pivotal.
O Framework Spring quando começou teve a intenção de ser uma nova solução alternativa ao Java JEE (*Java Enterprise Edition*) para facilitar a agilizar o desenvolvimento de Aplicações Web com a tecnologia Java através do Spring MVC.
Como o próprio dono e mantenedor do Framework Spring informa em seu web site, "*Spring makes programming Java quicker, easier, and safer for everybody. Spring's focus on speed, simplicity, and productivity has made it the world's most popular Java framework.*" (Spring, 2020). As palavras da própria Spring explicam a razão pela utilização do Spring neste trabalho de dissertação. Rapidez, facilidade, segurança, um dos mais utilizados pelas empresas e mercado profissional Java.

Atualmente o Framework Spring tornou-se mais do que apenas mais um Framework Java, o Spring hoje é um ecossistema provedor de Frameworks e bibliotecas para o desenvolvimento de todo o ciclo de vida de um software de qualquer tipo ou natureza (Standalone, Web, Middleware, Cloud) e muitas soluções para software que são componentes externas ao software principal, como por exemplo conexão e acesso a bases de dados ou comunicação com um Message Broker.

Outra razão pela escolha do ecossistema Spring, é que será possível utilizar Frameworks e Libraries semelhantes, disponibilizados por este ecossistema, para atender o desenvolvimento de ambas versões do software para a gestão de alunos proposto neste trabalho, ou seja, tanto no modelo monolítico com programação bloqueante e quanto no modelo de micro-serviços com a programação reativa.

O código de todos os projectos deste trabalho de dissertação estão disponíveis no repositório do Bitbucket, e poderá ser solicitado acesso a este repositório ao enviar um email a rodneynporto@gmail.com. O endereço para aceder a este repositório é https://bitbucket.org/rodney_dissertation_2020/workspace/projects/UN .

5.2 – Evolução da capacidade de processamento computacional

Como descrito na seção Resumo deste trabalho, a evolução tecnológica também tem se tornado parte integrante, ou mesmo a base, dos argumentos para organizações empresariais decidirem por otimizar a relação custo/benefício de algum modelo ou processo de negócio já operacionalmente ativo ou novo a ser implementado. Uma parte importante

desta evolução tecnológica é a capacidade de processamento computacional através de processadores que vem aumentando sua velocidade de execução de processamento, aumento de memória *Cache* interna (sem a necessidade de utilizar a memória RAM), e o aumento da quantidade de núcleos de processamento (*Cores*) para execuções de tarefas de processamento em paralelo.

Um exemplo da capacidade de processamento computacional potencializada nos dias atuais é a Computação na Nuvem, descrita no capítulo 4, que oferece uma infraestrutura para suportar a escalabilidade de aplicações/componentes de software sobre a uma visão horizontal com centenas (e até mesmo milhares) de processadores, e cada um com vários núcleos (*cores)*. Uma vez que há disponível essa infraestrutura de processamento potencializada, passa então o software desenvolvido a ser o responsável por aproveitar os benefícios dos recursos da Computação em Nuvem.

Será então apresentado a seguir a programação bloqueante e a programação reativa e como estas funcionam sobre os recursos de processamento computacional disponíveis e suportados para ambas.

Antes será feita uma definição básica, objetiva e comum para ambos os tipos de programação que um ***processo de execução de software*** será resumido em "Executar códigos programados com as tarefas de criar, inicializar, manipular e avaliar variáveis, simples ou complexas, assim como executar e avaliar funções para retornar, ou não, Valores para outras funções ou variáveis, sendo este processo executado dentro de um recurso do processador chamado de ***Thread*** ."

5.3 – Programação bloqueante

Pode ser definida como programação bloqueante a implementação de um código no qual o *processo de execução de software* acontece de forma sequencial (ou síncrona) em que um pedaço (linha ou bloco) de código implementado fica em "*Espera* ou *Aguardo*" e somente será executado após o pedaço de código anterior, em execução, ser totalmente finalizado e não esteja mais a *Bloquear* a *Thread* em que o processo de execução se encontre, ou seja, cada pedaço de código programado ao entrar em processo de execução faz a Thread assumir estado de Bloqueada até que este esteja finalizado.

O controle do processo de execução do software sobre o código fica sob a responsabilidade do programador/desenvolvedor que o tenha escrito, pois este irá escrever o código definindo a sequência (sincronismo) de execução de cada pedaço do código. Por isso este tipo de

programação também pode ser chamada de *programação Imperativa*, porque o programador/desenvolvedor define (manda ou arbitra) em seu código sobre as premissas da tarefa em processamento que são “O que executar” e “Quando, em que ordem, executar”.

A programação bloqueante, ou Imperativa, é um modo Tradicional de escrita e programação de códigos possuindo o benefício de ser, relativamente, facilmente implementada, compreendida, testada, mantida e avaliada em tempo de execução (técnica de *DEBUG*) pela equipa de programadores/desenvolvedores e assim a produtividade para a criação e manutenção do software também pode ser considerada como um processo executado de forma, relativamente, satisfatoriamente rápida. Entretanto esta forma de programação pode tornar-se um fator limitador da Capacidade de processamento, de Performance e de Desempenho de um software a depender do volume de Utilizadores e Requests de processamento (Threads) que sejam realizadas dentro de um curtíssimo espaço de tempo simultaneamente.

Por exemplo, durante o tempo disponível para o processo de inscrição, recebimento e avaliação de Candidaturas para novos alunos uma Universidade A pode possuir um número simultâneo de acessos por minuto ao seu Portal Web de Candidaturas muito maior do que uma Universidade B, hipoteticamente para a Universidade A 10.000 acessos/minuto e a Universidade B 1.000 acessos por minuto, e estes acessos podem resultar em um alto número de Pesquisas e detalhamento sobre os Cursos, as Disciplinas e os Professores, e também um alto número de requisições para a criação de Candidatos bem como de requisições para a realização de Candidaturas com o envio e pré-avaliações de documentação.
É nítido que a Universidade A irá demandar uma maior capacidade computacional do que a Universidade B para atender as requisições em seu sistema a fim de não perder para a Universidade B possíveis alunos (em outras palavras Clientes e Consumidores) e este tipo de análise de “estresse” de Consumo do sistema é importante para avaliar se uma organização está preparada para atender “satisfatoriamente” ao seu Mercado Consumidor, trazendo para o assunto deste tópico, se o software desenvolvido está a receber e processar as requisições e a responder de forma “satisfatória” aos Utilizadores.

5.3.1 – Programação Não-bloqueante

Pode ser considerado como programação Não-bloqueante a técnica de programação utilizando-se Multi-Threads, ou seja, divide-se a carga e demanda computacional pela quantidade de Threads simultâneas que correspondam a mesma quantidade de processadores disponíveis, ou seja, cada uma Thread será utilizada por um processador suportando assim um processamento Paralelo.

Esta técnica obviamente oferece um ganho de performance melhor do que a programação bloqueante mas esse ganho de performance também possui algumas limitações como:
- quantidade de Threads (processadores) disponíveis;
- cada processador a mais representa um aumento no custo monetário do sistema;
- cada Thread individualmente ainda é um processamento bloqueante;
- aumento da complexidade da programação Imperativa para controlo da troca de contexto entre as Multi-Threads;
- aumento da complexidade da programação Imperativa para avaliação da carga e demanda computacional, uma vez que para baixa necessidade de processamento computacional a performance de Multi-Thread perde seu valor de ganho de performance.

Em razão destes pontos acima a técnica de Multi-Threads também pode ser chamada de “programação Não-bloqueante limitada”.

5.4 – Programação reativa

Pode ser definida como programação reativa a implementação de um código no qual o *processo de execução de software* acontece de forma não sequencial em que os pedaços (blocos) de código implementados serão executados independentes de algum pedaço de código anterior que ainda não esteja totalmente executado e finalizado, ou seja, o código executado não fica em “*Espera* ou *Aguardo*” de uma resposta até que o pedaço de código previamente inicializado não esteja mais a ***Bloquear*** a *Thread* em que o processo de execução se encontra. Por isso este tipo de programação também pode ser chamada de *programação assíncrona*, porque o programador/desenvolvedor define em seu código as premissas das tarefas em processamento sobre “O que executar” entretanto não arbitra mais sobre “Quando, em que ordem, executar”, ficando esta segunda premissa em responsabilidade do Framework ou Biblioteca da tecnologia utilizada.

É um modo, geralmente, *Funcional – Não Tradicional* – para escrita e programação de códigos não sendo os mais fáceis de serem entendidos, compreendidos e avaliados em tempo de execução (técnica de *Debug* de software) no qual exige uma curva de aprendizado e adaptação maior dos programadores e desenvolvedores.

5.4.1 – Manifesto reativo

Em Setembro de 2014 foi criado um *Manifesto reativo* que definiu as características de um sistema de software para que esse pudesse ser chamado de reativo:

> "We believe that a coherent approach to systems architecture is needed, and we believe that all necessary aspects are already recognised individually: we want systems that are Responsive, Resilient, Elastic and Message Driven. We call these Reactive Systems" (Bonér et al., 2014).

As características de um sistema reativo devem ser:

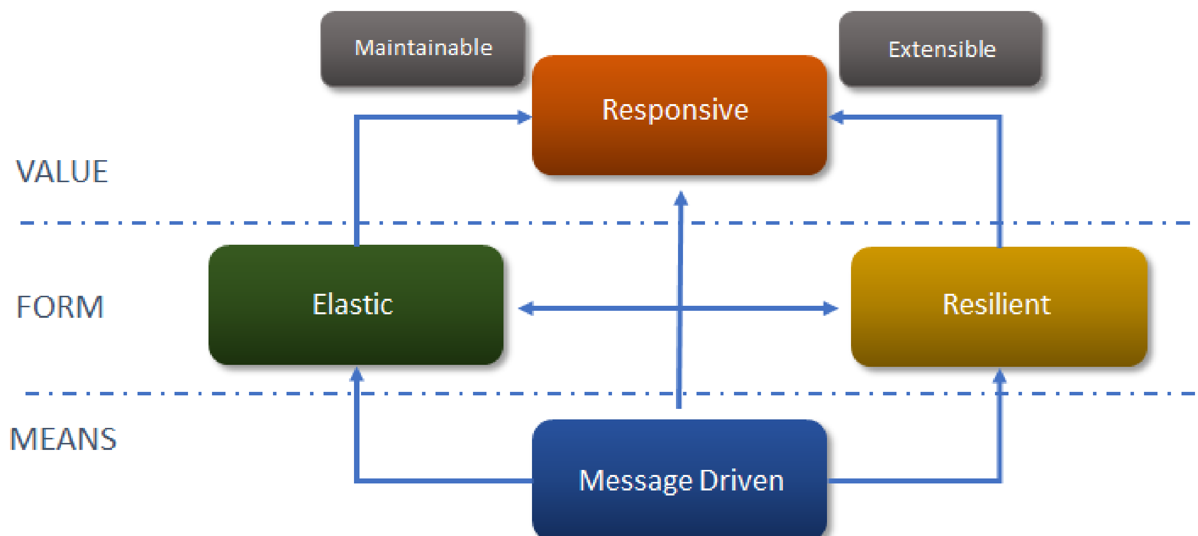


Figura 14 – Características de um sistema segundo o Manifesto Reativo (Reactive Manifesto Org, 2021)

Responsivo - a resposta deve ser o mais rápido possível, consistente, com limites superiores confiáveis para garantir a qualidade do serviço em execução. A capacidade de resposta é crucial para, além de uma melhor usabilidade, detectar e tratar rapidamente problemas com eficácia (Bonér et al., 2014).

Resiliente - a responsividade deve permanecer perante falhas em serviço de qualquer missão crítica de disponibilidade (alta ou baixa) a garantir que este seja resiliente. A replicação, contenção, isolamento e delegação são propriedades da resiliência. As falhas devem ser contidas no próprio componente onde ocorram, devem ser tratadas isoladamente de outros componentes garantindo que, ao ocorrerem, não comprometam todo o sistema, de forma a não sobrecarregar o componente com as falhas durante sua utilização. A recuperação deverá ser delegada do componente com falhas para outro componente (externo) enquanto que a disponibilidade continue a ser assegurada pela replicação, quando esta for necessária (Bonér et al., 2014).

Elástico - a responsividade permanece independentemente da carga variável de execução/processamento exigida. “Podem reagir a mudanças na taxa de entrada, aumentando ou diminuindo os recursos alocados para atender a essas entradas. Isto implica projetos que não têm pontos de contenção ou gargalos centrais, resultando na capacidade de fragmentar ou replicar componentes e distribuir insumos entre eles. Os Sistemas reativos suportam algoritmos preditivos, bem como reativos, de escalonamento, fornecendo medidas relevantes de desempenho ao vivo. Eles alcançam elasticidade de forma econômica em plataformas de hardware e software de commodities” (Bonér et al., 2014).

Orientado a Mensagem - deve implementar a passagem assíncrona de mensagens entre os componentes a estabelecer uma fronteira que garanta-os o acoplamento solto, o isolamento e a transparência de localização entre si. Também fornece meios para delegar mensagens quanto a falhas ocorridas. A comunicação orientada a mensagens explícitas, junto a sua localização transparente, permite a gestão destas mensagens, através de filas, sobre sua carga, elasticidade e fluxo e também sobre as falhas com construções e semânticas através de um cluster ou um host único. Esta forma de comunicação permite que o consumo de recursos ocorra pelos destinatários/interessados das mensagens enquanto, ou quando, estes estiverem ativos, reduzindo uma possível sobrecarga do sistema como um todo (Bonér et al., 2014).

A programação reativa ao seguir as características definidas pelo Manifesto Reativo torna-se ideal para o desenvolvimento de software que será executado dentro de uma

arquitetura de micro-serviços pois este tipo de programação irá permitir que seja aproveitado ao máximo os benefícios característicos desta arquitetura.

5.4.2 – Como funciona a programação reativa

Uma vez que a programação reativa não fica sob o estado de “Aguardo ou Espera” até que um bloco de código seja finalizado para que um outro bloco de código seguinte seja executado, seu processo de execução de código segue, então, o estado de “Pronto para *Reagir*” assim que puder, assincronamente.

“In computer science, the term Asynchrony refers to the occurrence of events independent of the main program flow and ways to deal with such events. Hence the main program flow keeps running and those events or operations are done parallel then it notifies the main program flow” (Ankur, 2021).

Para atender as características do Manifesto Reativo e possuir o comportamento de execução de código reativo e assíncrono, a programação reativa é desenvolvida e realizada com base no Padrão de Projeto (*Design Pattern*) “***Observer***” combinado com o Padrão de Projeto (*Design Pattern*) ***Iterator*** e com algumas melhorias como a estratégia de “***Backpressure***”. Os dados/objetos em execução, que serão os eventos como descrito na citação acima do *Ankur*, deverão ser parte de um “***Data Stream”*** (Fluxo de Dados).
Desta forma, a programação reativa segue o *Paradigma Reativo* que propõe, diferentemente da programação imperativa, ao programador/desenvolvedor uma escrita de código desacoplado ao permitir que os dados/eventos sejam isolados das rotinas que os irão manipular, principalmente pelo fato de utilizar a combinação de Padrões de desenvolvimento e Estrutura de Dados já conhecidos ao longo de muitos anos pela comunidade de desenvolvimento de software.

Como os padrões de projeto Observer e Iterator, o Data Stream e a Back Pressure, podem ser considerados os pontos chave para o alcance do benefício de performance expectável e/ou obtido na programação reativa, seguirá então uma breve explicação sobre estas estratégias adotadas.

P**adrão Observer -** faz parte do catálogo "*GoF (Gang of Four)*" de Padrões de Projeto e pertence ao grupo classificado como Padrões de Comportamento (existem outras duas classificações chamadas Padrões de Criação e Padrões de Estrutura).
A intenção do Observer é: "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically." (Gamma et al., 1994, p.293). Ou seja, os objetos programados irão "Reagir" imediatamente ao "escutarem" a mudança de estado de seus objetos relacionados, sem a necessidade de ficarem à espera de forma Imperativa (e bloqueante) para saber se deverão fazer algo ou quando deverão fazer algo.
Para que esse comportamento de Reação a algo ocorra automaticamente, o Padrão Observer define seus objetos a ser do tipo *Publisher* (publicador) e *Subscriber* (subscritor) de forma que os Publishers, ao sofrerem uma mudança de estado, irão notificar seus objetos Subscribers relacionados e estes irão então reagir a essa notificação automaticamente.

Padrão Iterator - faz parte do catálogo "*GoF (Gang of Four)*" de Padrões de Projeto e pertence ao grupo classificado como Padrões de Comportamento.
A intenção do Iterator é: "Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation." (Gamma et al., 1994, p. 257). Ou seja, cada objeto será integrante de uma Iteração de objetos, de seu mesmo tipo, de forma otimizada para acessar todos estes objetos independente do tipo de coleção/estrutura de dados (Lista, Matriz, Mapa etc) que estes possam fazer parte.

Stream ou Data Stream - "When we think of a stream we might picture water flowing through a fixed channel or path. In programming, it is data that's flowing. So, simply put, a stream in programming means the flow of data." (Study.com, 2021). Ou seja, este Fluxo de Dados é um tipo de conjunto, ou coleção, de dados (objetos) que fluem de uma fonte dos dados até os devidos destinos que os irão consumir. Estes dados irão representar os Eventos a serem percorridos e transformados durante a execução do código do software.
Já na programação reativa há o **Reactive Streams**, ou **Reactive Data Streams** (ou fluxos de dados reativos), e que é essencialmente o mesmo tipo Stream mas no qual os dados são recebidos com o fator *Tempo* adicionado. Isso significa que, enquanto um Stream recebe os dados imediatamente, um Reactive Stream recebe os dados de acordo com o Tempo. Esse fator Tempo, a mais, é muito importante uma vez que a Fonte dos Dados os produz ao longo do Tempo e isso resultará também nestes dados serem recebidos pelo seu Consumidor ao

longo do tempo. Com isso o fato deste Tempo ser previamente desconhecido e o Consumidor dos dados não saber quando estes irão chegar, pois não são mais imediatamente recebidos, cria um impedimento para que o programa/software seja escrito com um paradigma de código de execução bloqueante.

Backpressure - para descrever e exemplificar o que é o *Backpressure*:

> “Imagine um data stream muito grande e um subscriber com uma capacidade de consumo limitada, de forma que a emissão é muito mais rápida do que o consumo dos eventos. Nessa situação corremos um grande risco de *OutOfMemoryError*. O backpressure é um mecanismo para evitar esse problema: ele permite que o emissor controle a cadência no consumo do data stream para que o subscriber não fique sobrecarregado. Uma analogia com mundo real seria uma válvula de torneira que delimita o volume de água evitando o transbordo da pia.” (NSTech, 2019).

O Backpressure então é um tipo de estratégia para que objetos, ou implementações, *Observables* (observáveis) produzam seus dados em quantidade e velocidade suficiente para que os objetos, ou implementações, Observers (observadores) sejam capazes de consumir estes dados sem a ocorrência de erros, ou problemas, por falta de uma performance do Observer equivalente a performance do Observable.

Após a descrição das técnicas de desenvolvimento de software utilizadas para compor o funcionamento da programação reativa, pode ser analisada em termos práticos, ou resumidos, a combinação de todas estas.
O Observer irá prover através dos objetos Publishers e Subscribers os pontos da comunicação assíncrona responsáveis por “Publicar” e “Consumir” os dados em execução. O Iterator será o responsável por prover para os objetos do Observer os mecanismos para percorrer, e navegar, entre os dados em execução. O Reactive Stream representará o fluxo (volume) dos dados produzidos por um objeto Observable (observável) que serão recebidos ao longo do tempo, e não imediatamente E o Backpressure irá controlar, e racionalizar, o Reactive Stream para que um objeto Observer não tenha erros de sobrecarga de dados durante a iteração, e manipulação, destes dados do stream que foram fornecidos, ou produzidos, pelo objeto Observable.

Capítulo 6 – Tecnologias complementares para as arquiteturas

Cada tipo de arquitetura de software irá possuir suas próprias componentes complementares além do software desenvolvido para atender as necessidades de negócio em questão. E estes componentes podem ser outros componentes de software desenvolvidos ou tecnologias e software já existentes.
Um ponto importante que deve ser percebido é que para este trabalho de dissertação a camada de Apresentação, ou de Frontend, não é descrita e apresentada uma vez que a idéia central é a de comparar arquiteturas de software mantendo transparente a utilização de ambas para os meios que as utilizará. Por este motivo também não será descrito sobre tecnologias complementares da camada de Front End.

6.1 – Tecnologias complementares para arquitetura monolítica

Para atender a esta arquitetura não são necessárias muitas tecnologias. Isso irá variar de acordo com a complexidade da necessidade de negócio. Neste caso em especificamente foi implementada uma necessidade de negócio sob um modelo arquitetural clássico, com apenas uma tecnologia de base de dados relacional.

6.1.1 – MySQL Server

O MySQL Server é um dos mais populares SGBD - Sistema Gerenciador de Base de Dados – na área de desenvolvimento de software monolíticos tendo ganhado muita notoriedade pela grande aceitação e adesão da comunidade acadêmica e logo em seguida também passou a ser adotado por muitas empresas. É desenvolvido, e com suporte mantido, pela Oracle Corporation que o distribui como um software *Open Source* (quando o código fonte é disponibilizado para qualquer um estudar, ou até mesmo alterar de acordo com sua necessidade, sem pagar por isso) pela licença *GPL*.

O MySQL Server é uma base de dados do tipo Relacional, ou seja, todos os dados são armazenados utilizando a estrutura de Tabelas, provendo assim a possibilidade de que dados sejam armazenados seguindo uma Modelagem Relacional de Dados sobre as regras de Normalização de Dados.

Ele também suporta a linguagem "*SQL – Structured Query Language*" que se tornou um dos mais utilizados padrões de linguagem para acesso e manipulação de dados.

6.2 – Tecnologias complementares para arquitetura de micro-serviços

Como já descrito até aqui, além dos micro-serviços desenvolvidos para atender às regras de negócio, esta arquitetura possui alguns outros componentes que complementam a sua estrutura e neste ponto irá ser descrito um pouco sobre o software utilizado para atender a cada componente dessa estrutura.

Primeiro ponto é que os serviços de Gateway Server (o *API Gateway*) e o Service Discovery não são software terceito utilizado para atender a estes dois componentes da arquitetura. Ambos foram, também, desenvolvidos para atender cada um a sua específica necessidade. Também foram utilizadas a tecnologia de desenvolvimento Java e as bibliotecas do Ecossistema Spring: - Spring Cloud Gateway, que foi utilizado para desenvolver o Gateway Server; - Spring Cloud Netflix Eureka, utilizado para desenvolver o Service Discovery.

6.2.1 – MongoDB

O MongoDB é talvez a mais famosa e conhecida base de dados não relacional – NoSQL utilizado pela comunidade de tecnologia para micro-serviços, além de também já ser adotada em muitos projetos por muitas empresas. Algumas delas, como a Google, já possui hoje uma versão de implementação própria do MongoDB para ser utilizada em sua própria plataforma de computação em nuvem a fim de otimizar seus serviços oferecidos.

Seu modelo NoSQL para armazenamento de dados segue o conceito de "Documentos", e estes armazenam os dados seguindo a estrutura de objetos do tipo JSON. Seus campos podem variar de documento para documento e sua estrutura de dados em formato JSON pode ser alterada ao longo tempo de utilização.
Os documentos são classificados e armazenados dentro da base de dados em "Coleções" (*Collections*) de documentos.

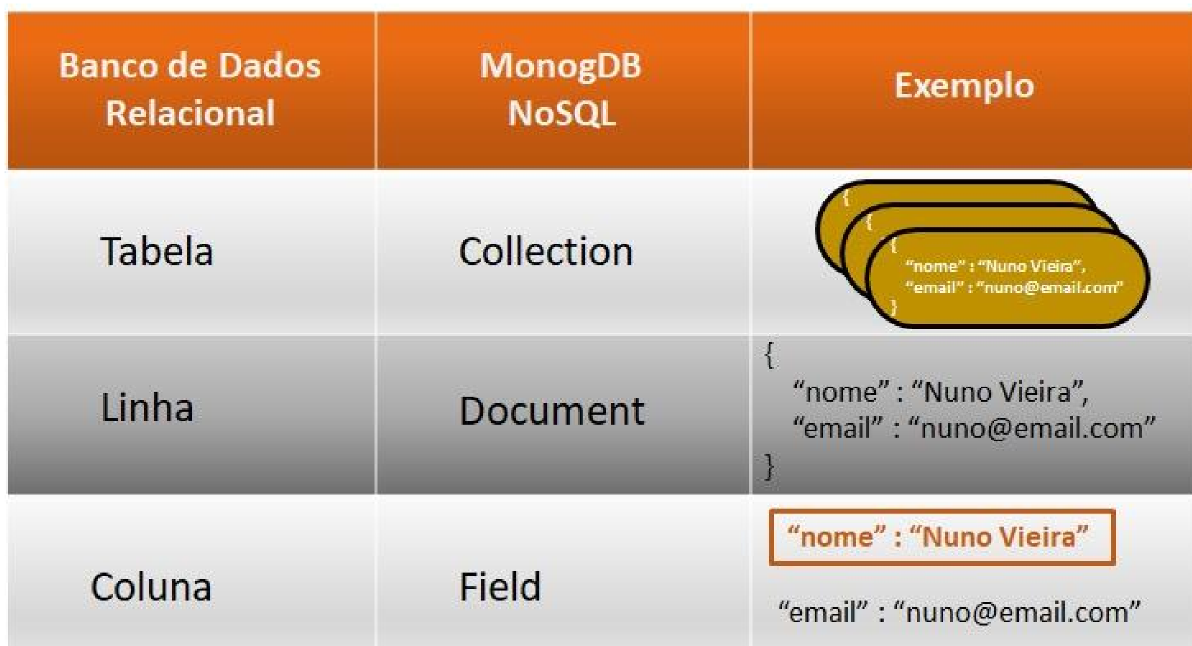
"MongoDB is a **distributed database at its core**, so high availability, horizontal scaling, and geographic distribution are built in and easy to use. " (MongoDB, 2021)

É uma excelente solução para compor a arquitetura de micro-serviços uma vez que é uma base de dados totalmente elástica e desenvolvida seguindo as melhores práticas para atender a este conceito de software elástico.

Outros dois destaques sobre vantagens do MonogDB sobre outras tecnologias são os recursos de **alta disponibilidade** e de **cluster fragmentado**.

A garantia de uma alta disponibilidade da base de dados ocorre por meio do processo de Replicação (*Replication*). Isso significa que um servidor que está a executá-lo pode ter cópias de suas bases de dados em outros 2 servidores por exemplo. As cópias são as chamadas Réplicas e um conjunto destas cópias é chamado de “*Réplica Sets*”.

O cluster fragmentado fornece uma capacidade de lidar com um grande volume de dados. Essa capacidade é obtida pela estratégia de dividir uma coleção de considerável volume em vários servidores, criando um o chamado cluster fragmentado. Nesse processo de divisão da coleção, é definida/escolhida uma chave de fragmento dentre os campos presentes nos documentos da coleção. Esse fragmento (*Shard*) de chave será então usado pelo balanceador pelo “Aglomerador de Fragmentos” (*Cluster Shard*) para determinar a distribuição apropriada de documentos.



Banco de Dados Relacional	MonogDB NoSQL	Exemplo
Tabela	Collection	{ { { “nome” : “Nuno Vieira”, “email” : “nuno@email.com” }
Linha	Document	{ “nome” : “Nuno Vieira”, “email” : “nuno@email.com” }
Coluna	Field	“nome” : “Nuno Vieira” “email” : “nuno@email.com”

Com base em original de https://www.alura.com.br/artigos/assets/mongodb-o-banco-baseado-em-documentos/image0.png

Figura 15 - Comparação entre NoSQL MongoDB e uma BDR.

6.2.2 - RabbitMQ

O RabbitMQ é um software open source utilizado como *Message Broker* (intermediador de mensagens) para o controle e gestão de um serviço de mensageria, e atua como uma interface para um processo de comunicação assíncrona entre aplicações e servidores. É um dos message brokers mais populares e utilizados desde empresas pequenas, como startups, a grandes empresas, como bancos ou instituições governamentais. Ele também trabalha sobre o protocolo *AMQP - Advanced Message Queueing Protocol*.

O RabbitMQ segue a organização lógica de mensagens publicadas por um *Publisher* e consumidas por um *Subscriber* através de uma estrutura de *Exchanges* (Intercâmbios), *Queues* (Filas), e *Bindings* (Ligações). Uma exchange pode possuir inúmeras bindings, cada uma binding deve ter somente uma uma queue configurada e cada queue pode possuir apenas um subscriber.
O RabbitMQ persiste uma mensagem no âmbito de uma queue somente enquanto nenhum subscriber não a consumiu, ou seja, uma vez que uma mensagem é consumida por um subscriber esta deixa de existir.

Há três possibilidades de troca de mensagens entre um Publisher e um Subscriber.

1ª- Um publisher publica uma mensagem para uma queue que possui somente um subscriber e este consome a mensagem. É o modelo mais simples dos três. A figura 16 abaixo demonstra esta funcionalidade.

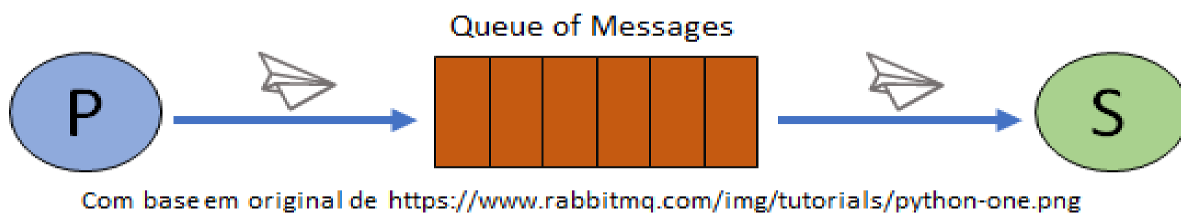


Figura 16 - Simples publish e subscribe uma mensagem no RabbitMQ

2ª- Um publisher publica uma mensagem para uma queue na qual possui 2 subscribers. Nesse caso, somente um subscriber consumirá a mensagem, que será o primeiro a assim fazê-la. Uma vez consumida, a mensagem deixa de existir para o outro subscriber. É um modelo de concorrência entre subscribers. A figura 17 abaixo demonstra essa funcionalidade.

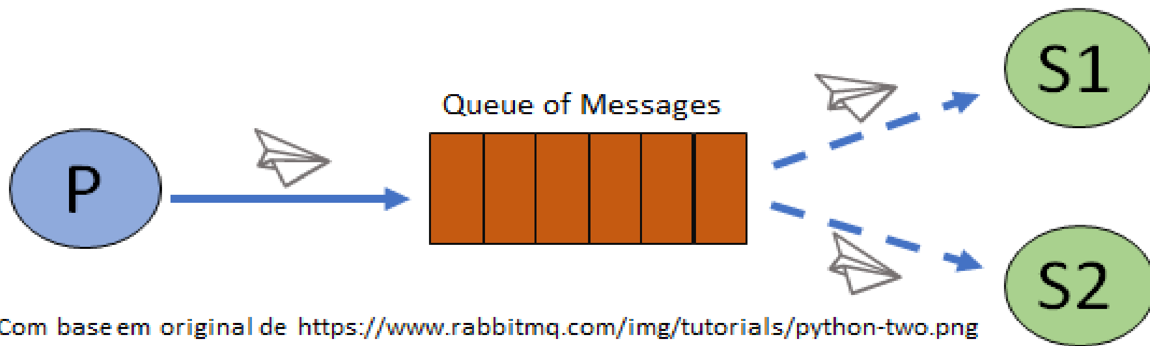


Figura 17 - Publish de uma mensagem em concorrência de subscribers no RabbitMQ

3ª- Um publisher publica uma mensagem em uma exchange que irá replicar e re-encaminhar a mesma mensagem para todas as queues que estiverem nela binded (ligadas). Cada queue então envia e disponibiliza a mensagem para um subscriber. É um modelo de consumo em paralelismo sobre os subscribers. A figura 18 abaixo demonstra esta funcionalidade.

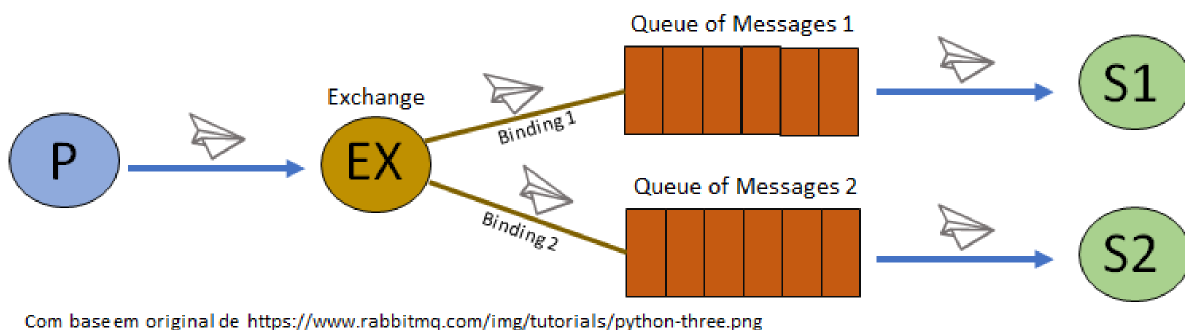


Figura 18 - Publish de uma mensagem para vários subscribers no RabbitMQ

6.2.3 - Docker e o Kubernetes

O *Docker* junto com o *Kubernetes* podem ser considerados como as tecnologias e os componentes mais importantes na popularização e maior sucesso à adesão da arquitetura de micro-serviços nos últimos anos, pois eles dois são os responsáveis pela dinâmica que irá facilitar e agilizar a construção e a gestão dos micro-serviços quase totalmente automatizada nos servidores de Computação em Nuvem.

O Docker é uma tecnologia que oferece o recurso de "conteinerização" de software, ou seja, ele encapsula dentro de **Containers** virtualizados o software que for necessário e desta forma, o mesmo software poderá ser disponibilizado para N desenvolvedores.

> “A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.”, (Docker Inc., 01/07/2021)

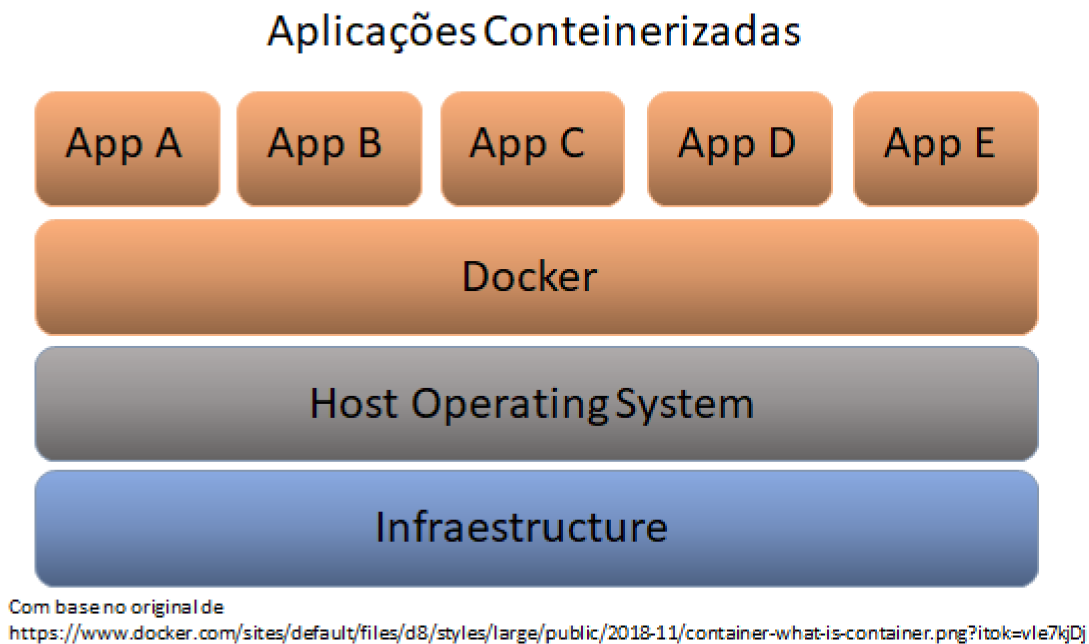


Figura 19 - Aplicações conteinerizadas com o Docker, (Docker., 01/07/2021)

O Kubernetes é uma tecnologia que oferece o recurso de “orquestrador” de containers. Ou seja, ele é o responsável por automatizar todo o ciclo de vida operacional de um container construído através do Docker.

> “Containers are a good way to bundle and run your applications. In a production environment, you need to manage the containers that run the applications and ensure that there is no downtime. For example, if a container goes down, another container needs to start. Wouldn't it be easier if this behavior was handled by a system?
> That's how Kubernetes comes to the rescue! Kubernetes provides you with a framework to run distributed systems resiliently. It takes care of scaling and failover for your

> application, provides deployment patterns, and more. For example, Kubernetes can easily manage a canary deployment for your system." (Kubernetes.io, 2021)

Kubernetes fornece e suporta importantes funcionalidades sob a sua responsabilidade de orquestrador de containers que são: *Service discovery* e *load balancing*, *Storage orchestration*, *Automated rollouts* e *rollbacks*, *Automatic bin packing*, *Self-healing* e *Secret* e *Configuration management*.

Service discovery and load balancing (Descobridor de serviços e balanceamento de carga) - são funcionalidades que expõem os contêineres usando um nome como endereço através de um DNS ou o endereço IP. Enquanto o Service Discovery é o mesmo já descrito na seção 2.5.2, o load balancing é a funcionalidade capaz de equilibrar e distribuir a carga do tráfego de rede quando este for muito alto para um contêiner, realizando assim um desdobramento desta carga mantendo ao máximo possível os contêiner estáveis.

Storage orchestration (Orquestração de armazenamento) - é uma funcionalidade que fornece uma gestão que automaticamente cria, disponibiliza, indisponibiliza e apaga dinamicamente um sistema de armazenamento de acordo com a demanda, como armazenamentos locais, provedores de nuvens públicas, e outros mais.

Automated rollouts and rollbacks (Implantação e recuo automatizados) - essa funcionalidade permite que o utilizador defina uma política ou estratégia para a implantação de seus contêineres deixando o kubernetes responsável por seguir esta definição automaticamente. Exemplo: automatizar a criação de novos contêineres para a implantação enquanto realiza a remoção de contêineres existentes, de forma que os novos contêineres adotem os recursos do contêiner em remoção.

Automatic bin packing (Empacotamento automático de contentores) - fornece um cluster (conjunto) de nodes (nós) que pode usar para executar tarefas conteinerizadas. Por exemplo, são definidas características e especificações de quanto de recurso quanto a CPU e/ou a memória (RAM) cada contêiner irá precisar para ser executado, deixando o Kubernetes responsável por encaixar os containers com essas definições em seus nós de forma a obter uma melhor utilização e consumo de seus recursos.

Self-healing (Auto-recuperação) - é uma função que automaticamente mata, reinicia e substitui containers que falham e não respondem mais ao *Health check* (controle de saúde) que foi definido pelo utilizador e este contêineres não são disponibilizados aos clientes até que estejam prontos para servir novamente a suas funcções.

Secret and configuration management (gestão de configuração e senhas) - permite que informações sensíveis sejam armazenadas e geridas. O utilizador consegue realizar implementações e atualizações de configurações e senhas de suas aplicações sem a necessidade de reconstruir as imagens de seus contêineres e sem expor estas informações sensíveis. Exemplo: senhas, fichas OAuth e chaves SSH.

Capítulo 7 – Testes das arquiteturas: monolítico x micro-serviços

Neste capítulo será apresentado a comparação entre as arquiteturas monolítica e de micro-serviços de forma mais prática e não apenas teórica.
A API Rest implementada para a utilização dos sistemas, a ferramenta do Apache JMeter para a realização dos testes e por fim os resultados consolidados dos testes.

7.1 – A API REST

Uma API - *Application Program Interface*, como o próprio nome já sugere e ajuda a perceber, é uma interface programática de uma aplicação e que é utilizada para possibilitar a comunicação entre dois módulos de software, ou seja, entre o que fornece um serviço e o que irá utilizá-lo.

Um exemplo muito comum de API existente e que poucas pessoas interpretam como tal é a possibilidade de utilizar algumas funcionalidades ou periféricos de um computador através do Sistema operativo.
Por exemplo: se um software qualquer precisa utilizar a câmera de um computador, seja do tipo desktop ou do tipo portátil, independentemente da marca, do fabricante e do modelo do computador e também da câmera, esse software ou incorpora o desenvolvimento de todas as marcas, modelos e tipos de computador e câmeras possíveis ou então utiliza uma API que muito provavelmente já é disponibilizada pelo sistema operativo.

No caso do sistema operativo MS Windows, por exemplo, as APIs são disponibilizadas e acessadas através de compilações de arquivos DLL, que é um tipo de biblioteca dinâmica que pode possuir e oferecer diferentes recursos, como dados, sub rotinas ou imagens (ícones, fontes, cursores etc), para utilização por outros programas instalados.

7.1.1 – API REST

Uma API REST é um tipo desenvolvida e disponibilizada sobre o(s) protocolo(s) web HTTP e/ou HTTPS, e sendo também conhecida como Web API.

Isso significa que o software fornecedor e o software consumidor irão realizar a sua comunicação seguindo as regras e normas do(s) protocolo(s) web HTTP e/ou HTTPS. Isto torna a API REST altamente utilizável com um alcance a nível global, uma vez que a única necessidade desta comunicação é a Internet, ou então uma Intranet para casos em que o software fornecedor e consumidor estejam dentro de uma mesma empresa e de seu próprio domínio de rede.

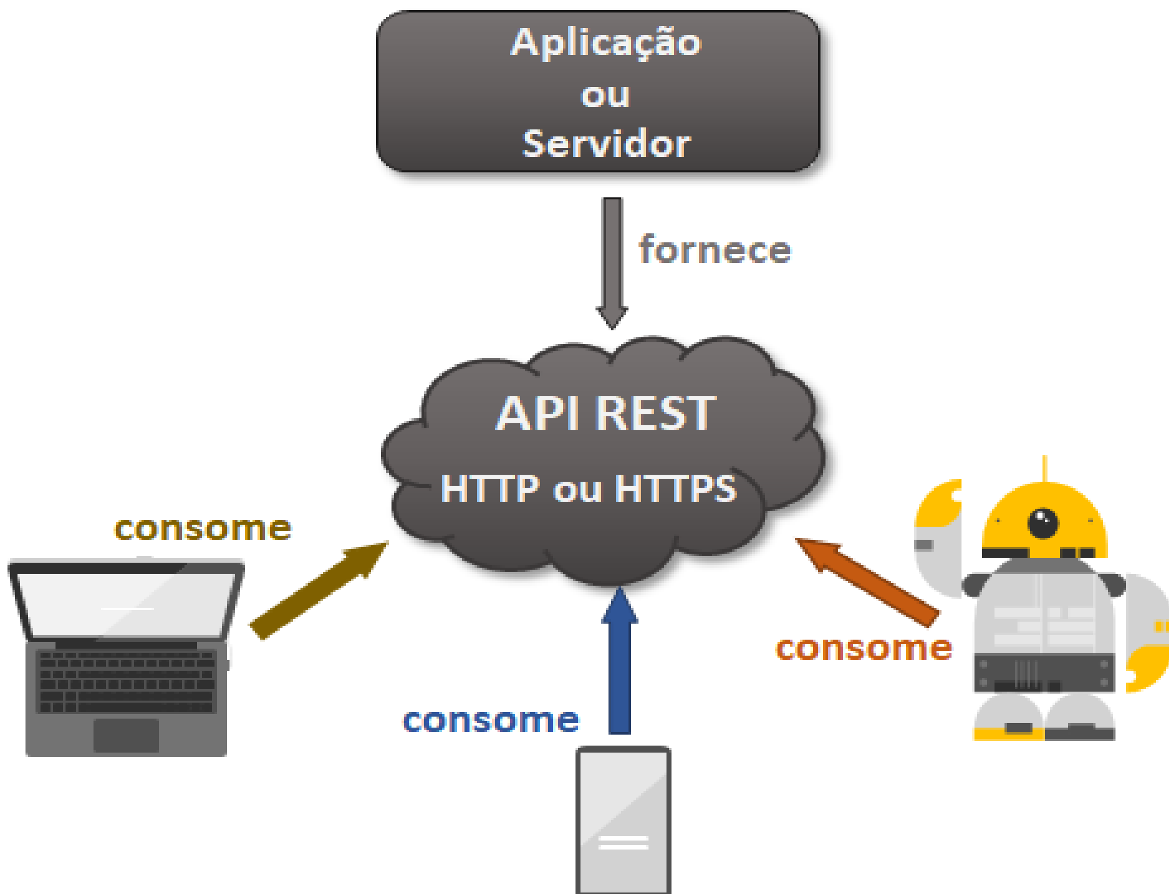


Figura 20 - Modelo de uma Web API REST sobre os protocolos HTTP ou HTTPS

7.1.2 – API REST desenvolvida

Embora possa ser plausível que a API REST siga diferentes abordagens e estratégias ao decidir por uma ou outra arquitetura de software, a API REST que foi desenvolvida neste trabalho é a mesma, ao máximo possível, para ambas as arquiteturas com a intenção de evitar qualquer tipo de influência da API sobre o resultado da comparação e/ou tomada de

decisão de uma organização, demonstrando que uma API não deveria ser um fator tão relevante.
A API sendo a mesma, é possível evitar que haja algumas preocupações seja no âmbito técnico ou no âmbito de negócios, como por exemplo sobre tomar ou não uma decisão para trocar da arquitetura monolítica para a arquitetura de micro-serviços, levantando a questão:: "Qual seria o impacto da mudança da API?"

- Qual seria o esforço para refatorar todos os Testes Unitários, Testes Funcionais, Testes de Integração?
- Qual seria o impacto sobre os clientes que consomem a API atual, de serem forçados a também realizar esta alteração em seu lado?

A API desenvolvida para ambas arquiteturas possui uma URL que segue a estrutura lógica /HTTP_METHOD http://dominio/entidade_microservico/funcionalidade, na qual cada parte significa:

- HTTP_METHOD - são os métodos de requisição do protocolo http, também conhecidos como *HTTP Verbs*. (POST, PATCH, DELETE, GET e etc)
- HTTP:// - protocolo web de requisição
- DOMINIO - o nome de domínio ou do servidor mais a porta
- ENTIDADE_MICROSERVICO - uma entidade de negócio identificada no modelo de classes, como descrito na seção 2.4.2, ou um micro-serviço desenvolvido;
- FUNCIONALIDADE - complementa a API com uma funcionalidade específica, como por exemplo um único parâmetro do tipo ID. Não é obrigatório existir.

Abaixo são apresentados alguns exemplos de API idêntica em ambas arquiteturas, em que o valor da variável PATH terá como exemplo o protocolo web adicionado de um domínio de exemplo, resultando em ***http://api.university.com:8080***. Todas as expressões em que se encontrarem entre os símbolos de **chaves { valor }** irão representar uma variável que pode ter um valor qualquer, de um tipo qualquer como texto alfanumérico, número, data, booleano. Este valor da variável será de acordo com o cliente consumidor que estiver executando a API.

HTTP METHOD	Entidade ou Micro-serviço	Funcionalidade
/POST	PATH/courses	/
URL	http://api.university.com:8080/courses	
Descrição	Cria um curso	
/GET	PATH/courses	/
URL	http://api.university.com:8080/courses	
Descrição	Busca e lista todos os cursos	
/GET	PATH/applicants	/{applicant_id}
URL	http://api.university.com:8080/applicants/{applicant_id}	
Descrição	Busca um candidato específico pelo ID	
/POST	PATH/students	/
URL	http://api.university.com:8080/students	
Descrição	Cria um novo aluno	
/PATCH	PATH/professors	/{professor_id}/addDiscipline
URL	http://api.university.com:8080/professors//{professor_id}/addDiscipline	
Descrição	Atualiza um professor adicionando a Ele uma nova disciplina para lecionar	
/PATCH	PATH/students	/{student_id}/enrollDiscipline/{discipline_id}
URL	http://api.university.com:8080/students/{student_id}/enrollDiscipline/{disciplineId_create}	
Descrição	Atualiza um aluno realizando a sua inscrição em uma disciplina	
/PATCH	PATH/students	/{registration}/tuitionFee/{divideBy}
URL	http://api.university.com:8080/students/{registration}/tuitionFee/{divideBy}	
Descrição	Atualiza um aluno realizando a sua faturação quanto a propinas	

Tabela 4 - Exemplos de URL idênticas entre a APIs monolítica e de micro-serviços

7.2 – Apache JMeter

É um software open source pertencente a Apache Foundation e desenvolvido totalmente em tecnologia Java. Originalmente seu intuito foi o de ser uma ferramenta para a execução de testes sobre Aplicações Web, entretanto, com o passar do tempo, também foi evoluindo e expandiu-se para inúmeros outros tipos de testes. Atualmente é uma das mais utilizadas

ferramentas para a execução de testes de desempenho e de carga para avaliar e medir o comportamento de aplicações.

> “Apache JMeter may be used to test performance both on static and dynamic resources, Web dynamic applications.
> It can be used to simulate a heavy load on a server, group of servers, network or object to test its strength or to analyze overall performance under different load types.” (Apache, 2021)

A suas funcionalidades permitem a execução de testes sobre diferentes tipos de aplicações, servidores e protocolos como, por exemplo:

- Aplicações web em protocols HTTP e HTTPS;
- Webservices dos tipos SOAP e REST;
- Bases de Dados diretamente via o JDBC do Java;
- Serviços de mensageria diretamente via o JMS do Java;
- Serviços de email e seus protocolos SMTP, POP3 e IMAP,
- entre outros

A ferramenta possui como estrutura e organização a criação de um Plano de Teste (*Test Plan*), no qual podem ser feitas configurações gerais e necessárias para definir um comportamento padrão. Por exemplo, a criação de variáveis de ambiente que contém as informações das Requests e dos Responses das URLs de uma API REST.

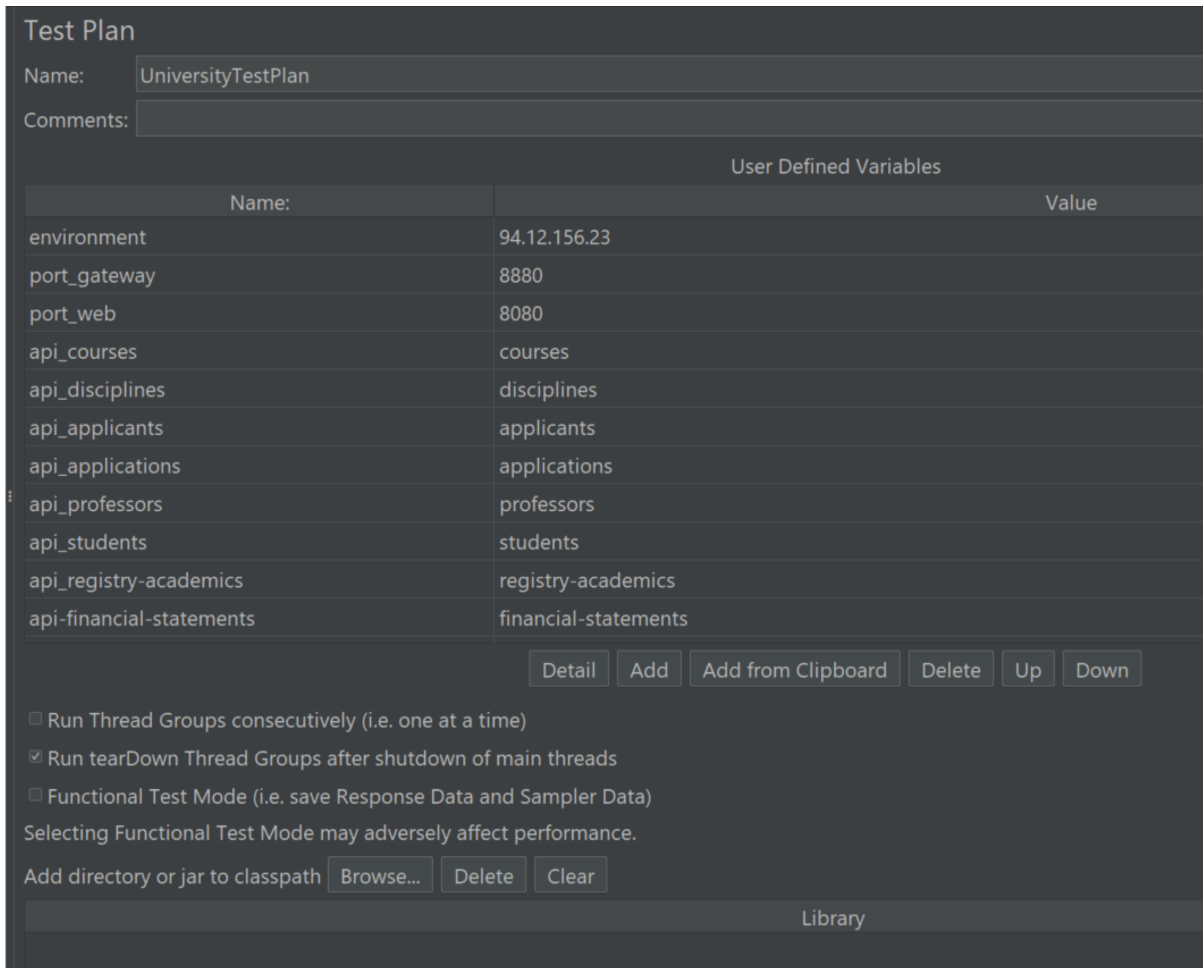


Figura 21 - Configuração de um Plano de Testes do Jmeter

Sob o plano de teste é definido um, ou mais, grupo de Threads (*Threads Group*) que representa o teste de desempenho de fato que será executado. São definidos neste componente as configurações mais específicas sobre “o que executar”, “como executar” e o “nível de estresse a executar”. Por ser uma ferramenta multi-tarefa, o JMeter suporta a execução de threads de teste simultaneamente dentro de cada grupo de threads e também a execução de testes para mais de um grupos de threads simultaneamente, disponibilizando consequentemente a geração dos resultados e amostragens por threads ou por cada grupo de threads.

As threads definidas no JMeter simulam os utilizadores da aplicação a ser testada.

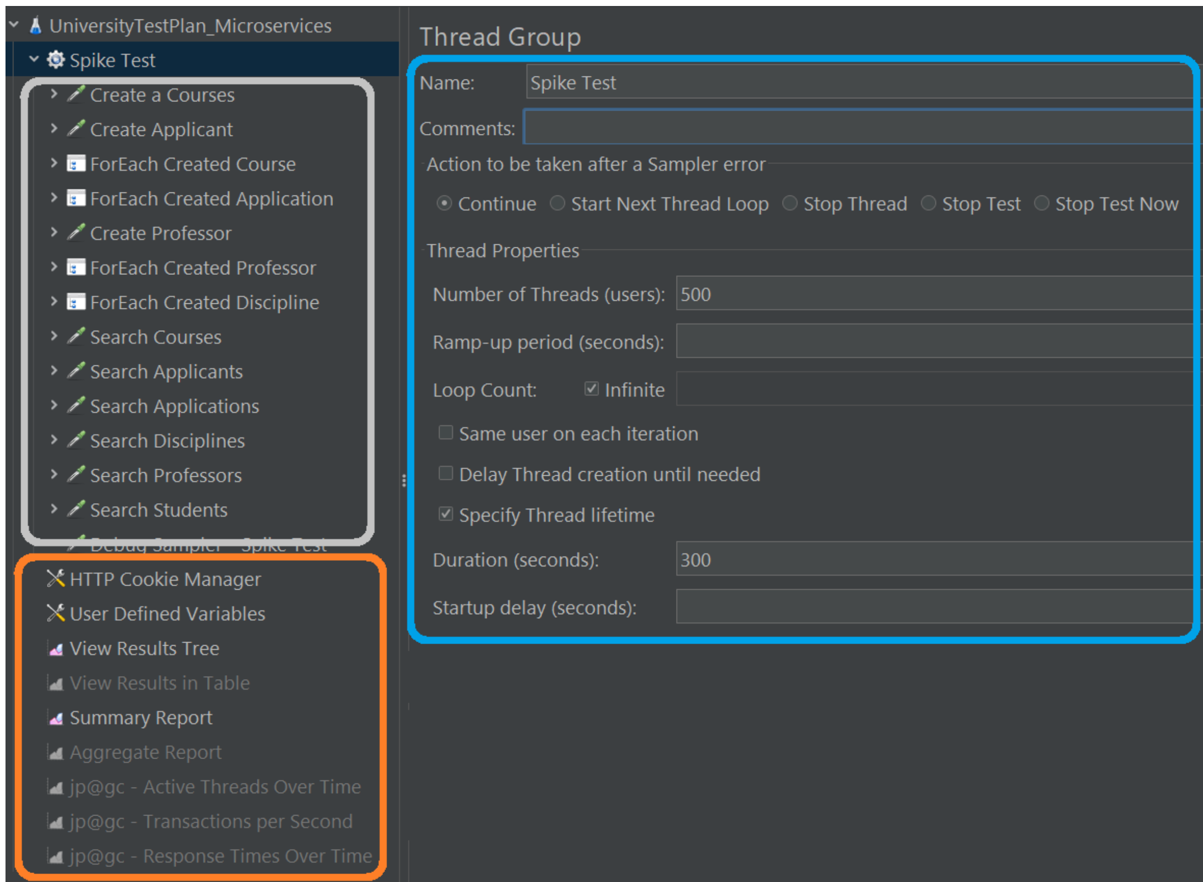


Figura 22 - Exemplo de um Grupo de Threads do Jmeter

Na figura 22 acima pode ser identificado: - na área de borda cinza temos as requisições que serão executadas, que neste caso são da Web API REST; - na área de borda laranja temos os “listeners” que podem ser algum tipo de configuração ou algum tipo de resultado dos testes demonstrados em tabelas ou gráficos; - na área de borda azul temos as parametrizações do teste de estresse que se deseja executar e avaliar, e que irão variar de acordo com o tipo de teste definido dentre os tipos disponibilizados pelo JMeter.

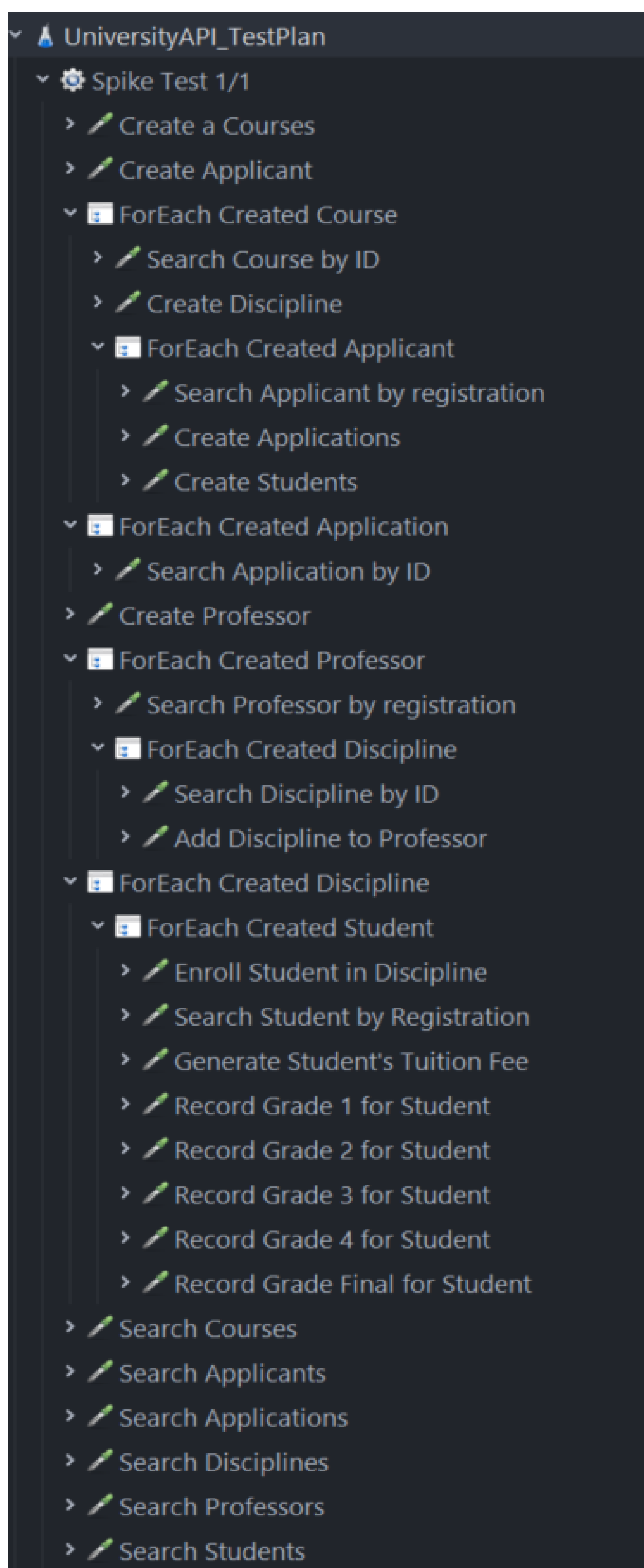


Figura 23 - API do sistema de gestão de alunos criado no JMeter

A figura 23 acima apresenta o plano de testes criado no Apache JMeter, representado pela área de borda cinza na figura 22, para testar a Web API Rest desenvolvida para um sistema de gestão de alunos. São um total de 26 requisições a serem testadas.

O Apache JMeter disponibiliza 4 tipos de testes de desempenho: SPIKE, LINEAR RAMP-UP, STEPPING LOAD RAMP-UP e o RAMPING UP SPIKE.

SPIKE - Teste de uso total de todo o número de utilizadores definidos e todos são inicializados imediatamente. Podem ser definidas parametrizações de quantidade específica de vezes que o teste deva ser executado (um loop que caso não seja definido será então assumido como infinito para reexecução) e/ou definido um número de tempo em segundos de duração total do teste.
Exemplo: define-se 100 utilizadores, reexecução em loop de 5 vezes e sem a duração de um tempo limite.
Todos os 100 utilizadores são inicializados ao mesmo tempo, o mais rápido e imediatamente possível para execução da API. Os testes da API são executados 5 vezes e sem um tempo definido para ser finalizado.
Este é o tipo de teste que foi utilizado para a comparação das arquiteturas descritas neste trabalho.

LINEAR RAMP-UP - Ao ser executado, o número de utilizadores definido vai sendo inicializado e incrementado linearmente dentro de um tempo em segundos até atingir o total de utilizadores desejados. Quando o total de utilizadores ativos é alcançado, podem ser seguidas as parametrizações de quantidade específica de vezes que o teste deva ser executado (um loop que caso não seja definido será então assumido como infinito para reexecução) e/ou definido um número de tempo em segundos de duração do teste.
Exemplo: Define-se 100 utilizadores, 20 segundos de “ramp up”, nenhuma quantidade de reexecução (ficando assim como infinita), e define-se um tempo total de duração do teste em 60 segundos.
Os utilizadores são inicializados linearmente durante o tempo de 20 segundos(ou seja 100/20 são 5 utilizadores/segundo) e em seguida ao alcançar o tempo de 20 segundos já com todos os 100 utilizadores segue a execução dos testes por mais 40 segundos, chegando ao total definido de 60 segundos.

Os resultados dos testes executados no JMeter podem ser apresentados em diferentes tabelas e gráficos assim como desejados e configurados para a execução.

O Apache Jmeter oferece uma extensa variedade e possibilidades de Testes de desempenho sob cada um destes 4 tipos descritos acima, além de mais de um tipo de Thread Group e dezenas de funcionalidades extras (como pré-processamento, pós-processamento, temporizadores, "*assertions tests*" e ainda inúmeros plugins) que podem ser adicionadas ao plano de testes, ao threads group ou à uma thread. É uma ferramenta na qual requer muito tempo de estudo, investigação e pesquisa prática para obter o melhor proveito possível de sua usabilidade e capacidade e para também contornar suas limitações e ou imprevisibilidades durante as execuções dos testes de desempenho.
Quanto mais detalhes um analista de Testes possuir como parâmetros e critérios para análise a avaliação do desempenho expectável de uma aplicação, servidor, base dados etc, melhor será o aproveitamento do Apache JMeter.

7.3 – Testes de Desempenho

O plano de testes deste trabalho de dissertação não possuiu ou seguiu requisitos com critérios, parâmetros ou estimativas como de âmbito profissional com demanda conhecida ou prevista de consumidores. A ideia planejada foi unicamente a comparação do desempenho de ambas arquiteturas descritas como uma prova de conceito, dentro de um mesmo cenário ao máximo possível, para que fosse analisado se uma arquitetura de micro-serviços pode realmente ser considerada mais eficiente do que uma arquitetura monolítica, simulando a usabilidade dos utilizadores através de suas requisições.

Os testes realizados sobre ambas arquiteturas foram:

1- Testes de sistema "Frio":

- 1.1- Tempo gasto em segundos para executar toda a API para 1 utilizador;
- 1.2- Tempo gasto em segundos para executar toda a API para 10 utilizadores;

2- Teste de desempenho sequencial:

- 2.1- para 50 utilizadores simulando um cenário de baixo estresse;
- 2.2- para 250 utilizadores simulando um cenário de alto estresse;

3- Teste de desempenho com 3 micro-serviços desativados

As etapas e processo para a execução dos testes foram:

1º- Testes para arquitetura monolítica:

1.1- Criar a arquitetura monolítica no ambiente do Google Cloud;

1.2- Executar todos os testes para o sistema monolítico;

1.3- Limpar a arquitetura monolítica do ambiente do Google Cloud.

2º- Testes para arquitetura de micro-serviços:

2.1- Criar a arquitetura de micro-serviços no ambiente do Google Cloud;

2.2- Executar todos os testes para o sistema de micro-serviços;

3º- Testes para arquitetura de micro-serviços com 3 serviços desativados:

3.1- Desativar 3 micro-serviços no ambiente do Google Cloud;

3.2- Executar todos os testes para o sistema com 3 serviços a menos;

7.3.1 - Ambiente de Computação em Nuvem

Foi utilizado um serviço de infraestrutura em Computação em Nuvem (*Cloud Computing*) para a execução dos sistemas de ambas arquiteturas e assim realizar os testes de desempenho sob um cenário o mais real e factível possível para esta prova de conceito.

O serviço utilizado foi o da Google Cloud e na modalidade inicial de “avaliação gratuita”, na qual a Google disponibiliza um valor (€) em créditos iniciais e com um tempo limitado (em dias) para sua utilização.

	Status	Nome ↑	Local	Número de nós	Total de vCPUs	Memória total
☐	✔	cluster-university	us-central1-c	3	6	12 GB

Figura 24 - Cluster do Google Kubernetes Engine da Google Cloud

Resumo

Região	europe-west2 (Londres)
Versão do banco de dados	MySQL 5.7
vCPUs	4 vCPU
Memória	16 GB
Armazenamento	20 GB
Capacidade da rede (MB/s)	1.000 de 2.000
Capacidade do disco (MB/s)	Ler: 9,6 de 240,0 Gravar: 9,6 de 240,0
IOPS	Ler: 600 de 15.000 Gravar: 600 de 15.000
Conexões	IP público
Backup	Automatizado
Disponibilidade	Várias zonas (altamente disponível)
Recuperação pontual	Ativada

Figura 25 - Configuração da instância do Cloud SQL da Google Cloud

As figuras 24 e 25 acima demonstram os recursos do serviço de computação em nuvem utilizado na modalidade de "avaliação gratuita". Cloud SQL, que é uma instância de Bases de Dados Relacionais, e o cluster do GKE - *Google Kubernetes Engine*, que é o orquestrador de contêineres.

A Google Cloud possui uma imensa e extensa gama de recursos, funcionalidades e configurações com muitos detalhes e que exigem um longo tempo de estudo, pesquisa e testes para conhecer e entender todas as possibilidades, melhores práticas e também a correta utilização de seus serviços.

Como o autor deste trabalho não possuía nenhum conhecimento e experiência sobre a criação, configuração e/ou manutenção de nenhum ambiente de computação em nuvem anteriormente é preciso realçar que, apesar de ter sido possível executar os testes de desempenho dos sistemas descritos, é provável e possível que as configurações feitas e utilizadas possam não ter sido as ideais para um melhor proveito de ambas arquiteturas em comparação. Um exemplo desse fato é que o teste de estresse de nível alto sobre ambos os sistemas começou a apresentar muitas dificuldades com 300 ou mais utilizadores simultaneamente, quando era expectável testar um número bem maior do que este.

Espera-se, entretanto, que qualquer falta de conhecimento para uma melhor preparação e configuração do ambiente da Google Cloud, ou uma melhor preparação e configuração dos sistemas para serem executados sobre esse ambiente especificamente, tenha ocorrido sobre ambos sistemas testados sem nenhuma influência acidental positiva ou negativa a mais no resultado dos testes.

7.3.2 - Teste 1: teste de sistema "Frio"

Este primeiro teste foi planejado para ser feito sem muitas expectativas sobre os resultados de desempenho dos sistemas testados, e sua intenção foi a ter em conta a "inércia de inicialização" dos sistemas e permitir que as tecnologias utilizadas, como por exemplo o Sistema operativo, a *JVM - Java Virtual Machine* ou a própria Google Cloud, pudessem realizar as suas estratégias de alocação de recursos computacionais para a melhor execução aplicacional, como por exemplo gestão de memória, escalonamento de processos, gestão de recursos I/O e etc.

A métrica utilizada para este teste foi a de tempo de duração de execução da API.

Vejamos os resultados dos testes para comparação.

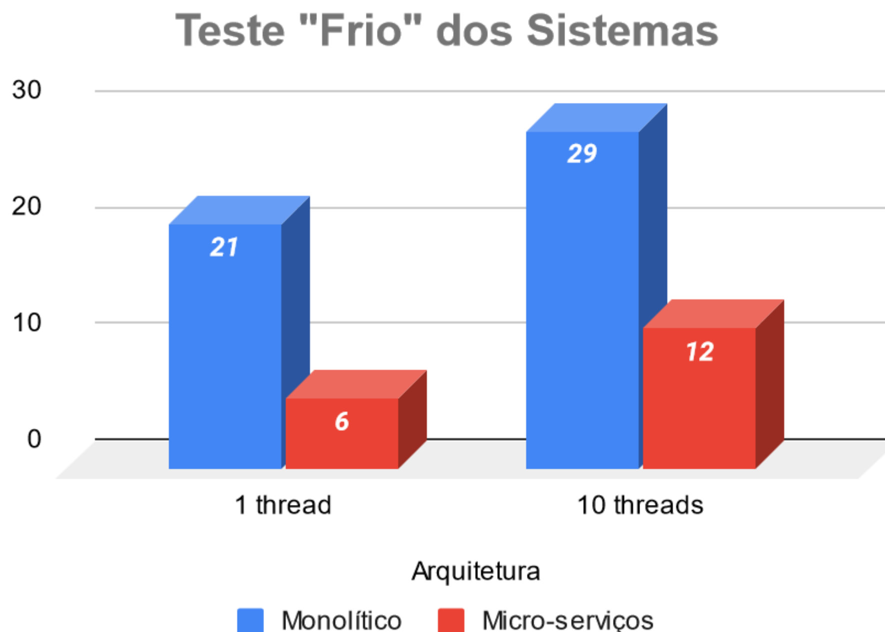


Figura 26 - Tempo em segundos de execução da API REST

O que ressalta daqui é que, embora para 1 utilizador os micro-serviços tenham um muito melhor desempenho, em 2,5 vezes (ou 250%) mais rápido, já com 10 utilizadores a melhora de desempenho diminui, caindo para 1,41 vezes (ou 141%) mais rápido.

Este resultado poderia levar a crer que a arquitectura de micro-serviços não escala tão bem face ao aumento de carga significativa como a monolítica. A explicação aparentemente mais pertinente para esse fato pode ser o fato de que para sair do estado “cold-start” para um estado “warm up” a arquitetura de micro-serviços necessite um pouco mais de estímulo de requisições, pois além dos micro-serviços de fato existem ainda componentes responsáveis pela comunicação síncrona e assíncrona, o service discovery e o message broker respectivamente.

Há de se ponderar ainda nos testes de “Sistema frio” ao aumentar em 10 vezes o número de utilizadores que o desempenho do sistema monolítico teve um aumento no tempo de execução em 38% e o desempenho do sistema de micro-serviços um aumento de 100%, e ainda assim o sistema de micro-serviços apresentou um desempenho consideravelmente e relevantemente maior e melhor de 141,66%.

O percentual de erros sobre as requisições foi de 0% para ambos os sistemas.

Vejamos agora o desempenho de ambos sistemas para 10 utilizadores sob o gráfico de “Threads Ativas ao Longo do Tempo” (*Active Threads Over Time*) retirado do dashboard gerado pelo JMeter

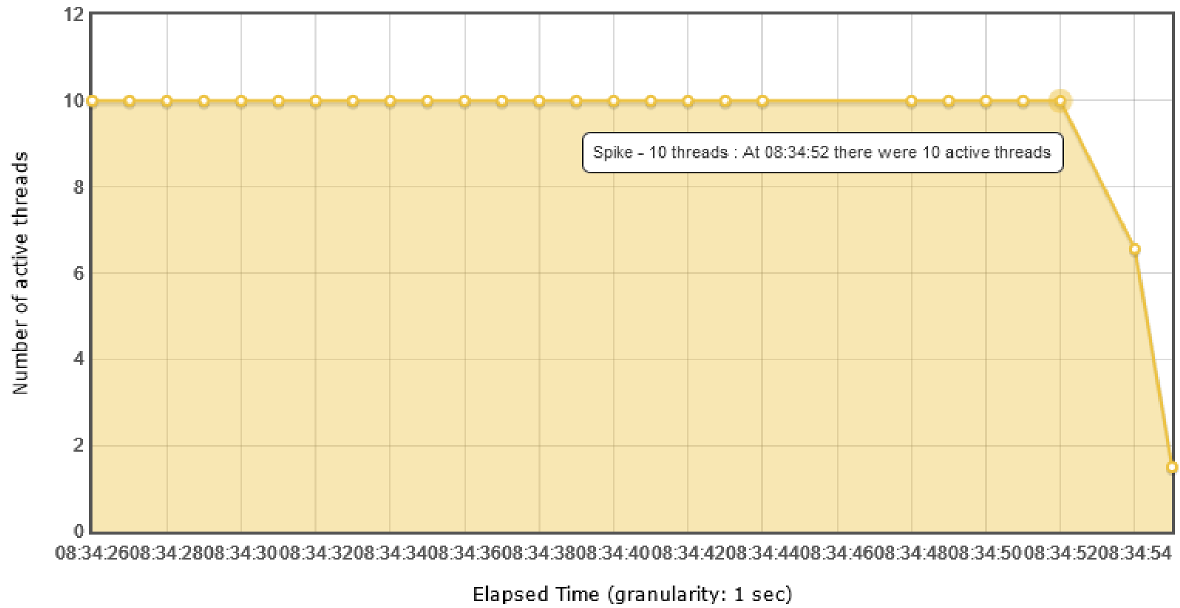


Figura 27 - Execução da API por 10 threads sobre sistema monolítico

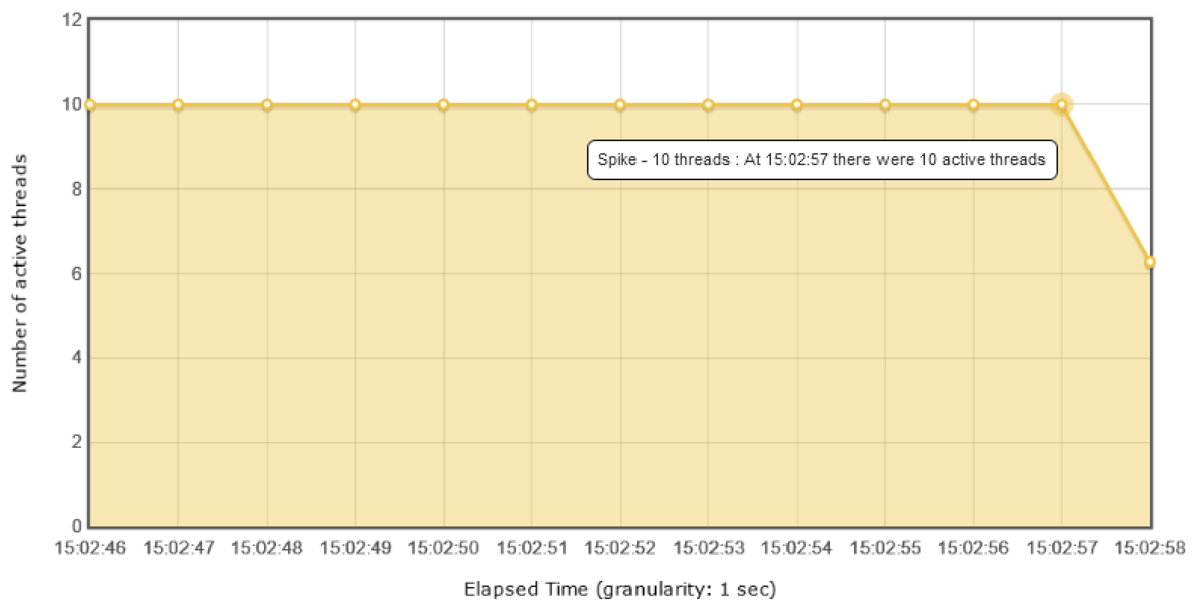


Figura 28 - Execução da API por 10 threads sobre sistema de micro-serviços

Estes dois gráficos acima, além de serem exemplos de como se comporta o teste do tipo Spike, ajudam a perceber o fato de que: ambos os sistemas inicializaram os 10 utilizadores no mesmo instante (o 1º segundo) entretanto o monolítico levou 3 vezes (300%) mais tempo do que os micro-serviços, 3/1 segundos, para encerrar todos os utilizadores da memória.

Como referido, estes são resultados de execuções com os ambientes em estado de "cold-start". A seguir seguem os resultados dos testes sobre os ambiente em estado "warm up" e sob um nível de estresse plausível, com o intuito de verificar se os micro-serviços tendem realmente a ter um melhor desempenho do que o monolítico sob um ambiente já inicializado e melhor preparado pelos sistemas gestores de recursos, como S.O. e a JVM.

7.3.3 - Teste 2: Teste de desempenho sequencial

Os testes de desempenho para ambos os sistemas tiveram neste caso 10 execuções sequenciais sobre os dois cenários de estresse: com 50 e com 250 utilizadores simultaneamente.

É importante ressaltar que o cluster Google Kubernetes Engine - GKE, na figura 25 não foi utilizado da mesma forma pelas duas arquiteturas:

- sobre o sistema monolítico - seus recursos computacionais estiveram dedicados somente ao software aplicacional, pois a base de dados relacional esteve a utilizar os recursos do serviço Cloud SQL, demonstrado na figura 24.
- sobre o sistema de micro-serviços - seus recursos computacionais foram compartilhados entre o software aplicacional, a base de dados do MongoDB, o message broker RabbitMQ e o Service Discovery.

As métricas analisadas nos testes de desempenho foram.
Latência - Utilizada para medir o tempo gasto entre o envio de um Request e o recebimento de seu Response. Quanto menor melhor.
Throughput (Rendimento**)** - Utilizado para medir o número de requisições processadas simultaneamente em uma unidade de tempo. Quanto maior melhor.
Erros - Percentual do Total de Requests na qual ocorreu algum erro durante a execução do Request ou do Response, ou a requisição nem mesmo foi executada.
O total de requisições é o cálculo da quantidade de requisições da API (26) multiplicado pelo número de utilizadores, 26x50=1300 ou 26x250=6500.
Os resultados apresentados para os testes com 50 utilizadores e, posteriormente, com 250 utilizadores são números consolidados como a média de todos os Requests da API, conforme a figura 23, para cada execução de teste.

Testes 2.1 - Testes com 50 utilizadores simulando um cenário de baixo estresse

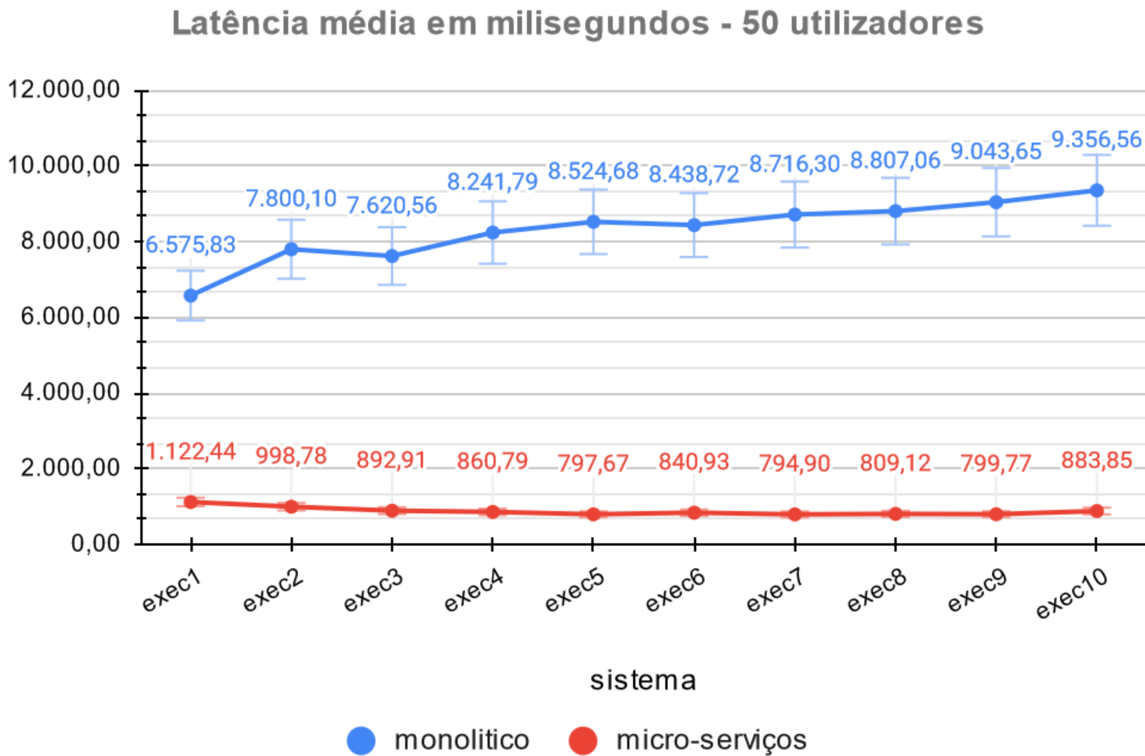


Figura 29 - Resultado da latência média por execução para 50 utilizadores

Através do gráfico na figura 29 acima pode ser verificado que:

- mesmo com poucas execuções, houve uma rápida tendência na perda de desempenho com o aumento da latência no sistema monolítico, enquanto que o sistema de micro-serviços apresentou uma estabilidade de sua latência;
- O sistema de micro-serviços apresentou um expressivo melhor desempenho do que o sistema monolítico que foi de 485%, como na execução 1, chegando a mais de 1000%, como na execução 9.

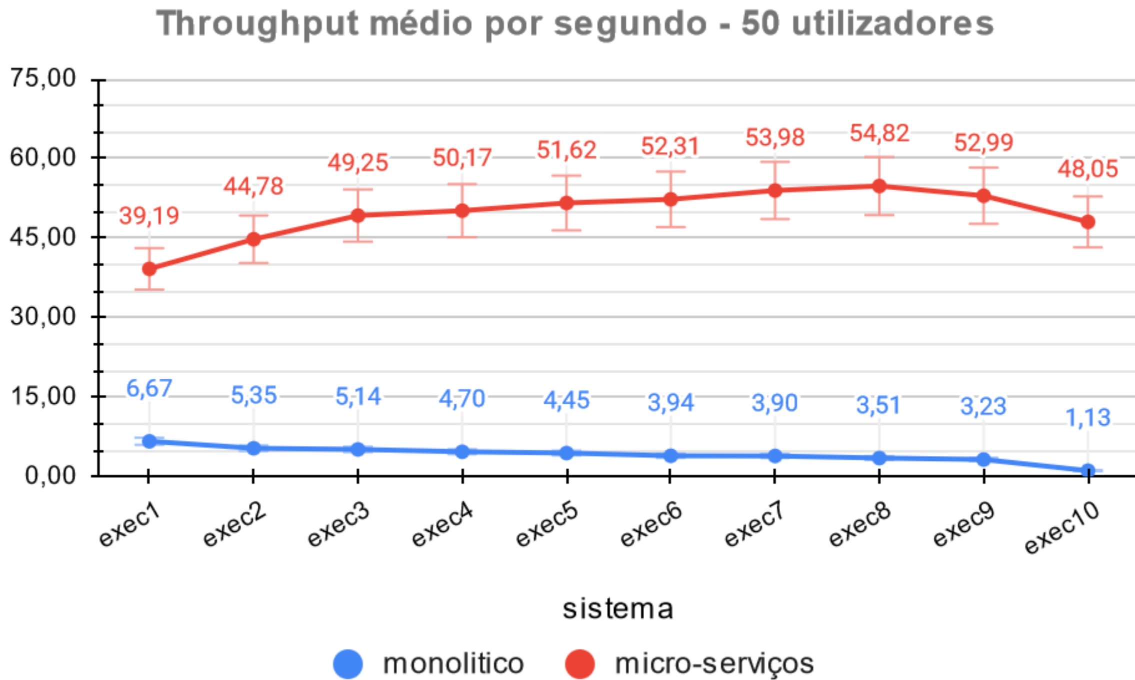


Figura 30 - Resultado de throughput médio por execução para 50 utilizadores

Através do gráfico na figura 30 acima pode ser verificado que;

- mesmo que nas primeiras execuções o sistema de micro-serviços apresente uma tendência de ganho no desempenho do throughput, diferentemente do sistema monolítico, em que há desde o início uma perda, após a 8ª execução começa haver uma tendência de perda de desempenho também no sistema de micro-serviços;
- Assim como na métrica de latência, o sistema de micro-serviços apresentou um expressivo melhor desempenho relativamente ao sistema monolítico, que foi de 487%, na execução 1, ultrapassando mais de 4100%, na execução 10. Entretanto, a queda do desempenho do sistema monolítico em -65% na 10ª execução pode ter sido um fato pontual, uma vez que nem mesmo nos testes com 250 utilizadores ocorreu um resultado parecido com este.

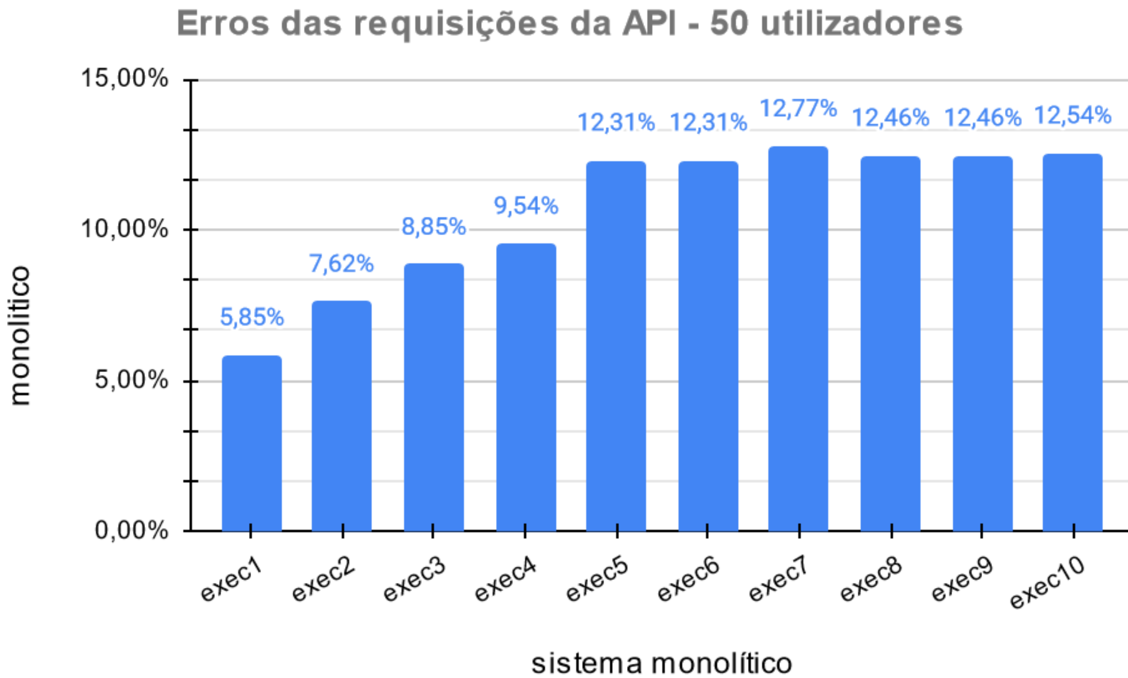


Figura 31 - Resultado de % de erros por requisição para 50 utilizadores

Através do gráfico, na figura 31 acima, pode ser verificado o percentual de erros do sistema monolítico que começa na execução1 a 5,85%, apresenta um crescimento linear até a execução5 a 12,31% (aumento de 110,42% até este ponto) e mantém uma estabilidade no percentual de erros.

A única comparação possível aqui é que o sistema de micro-serviços não apresentou erros, sendo assim 100% mais escalável e confiável do que o monolítico.

A seguir, apresentamos uma comparação dos valores das médias dos resultados de testes com 50 utilizadores.

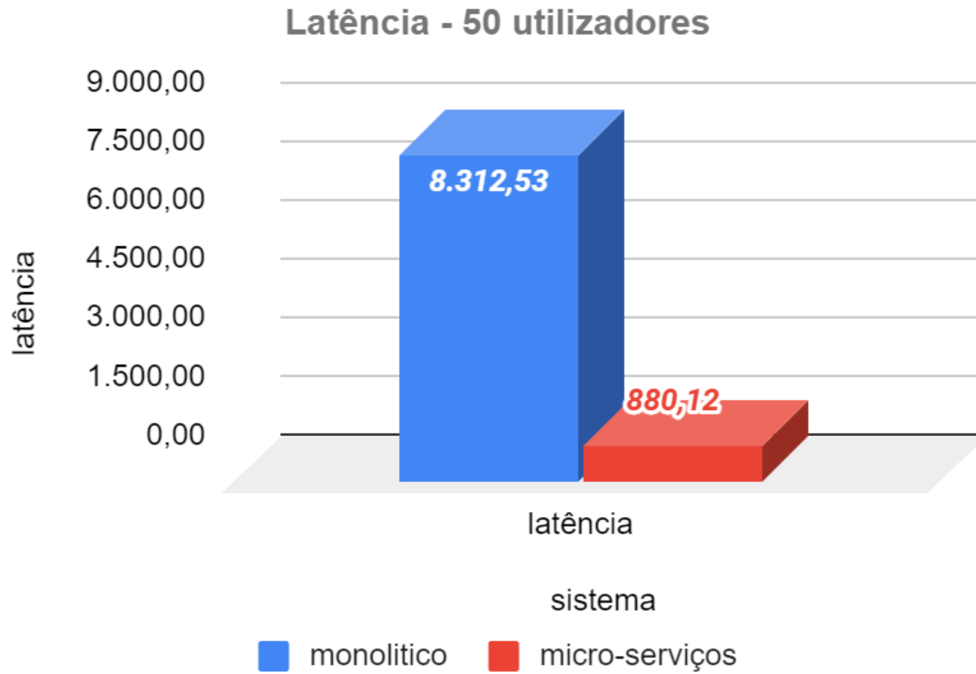


Figura 32 - Resultado médio da latência para 50 utilizadores

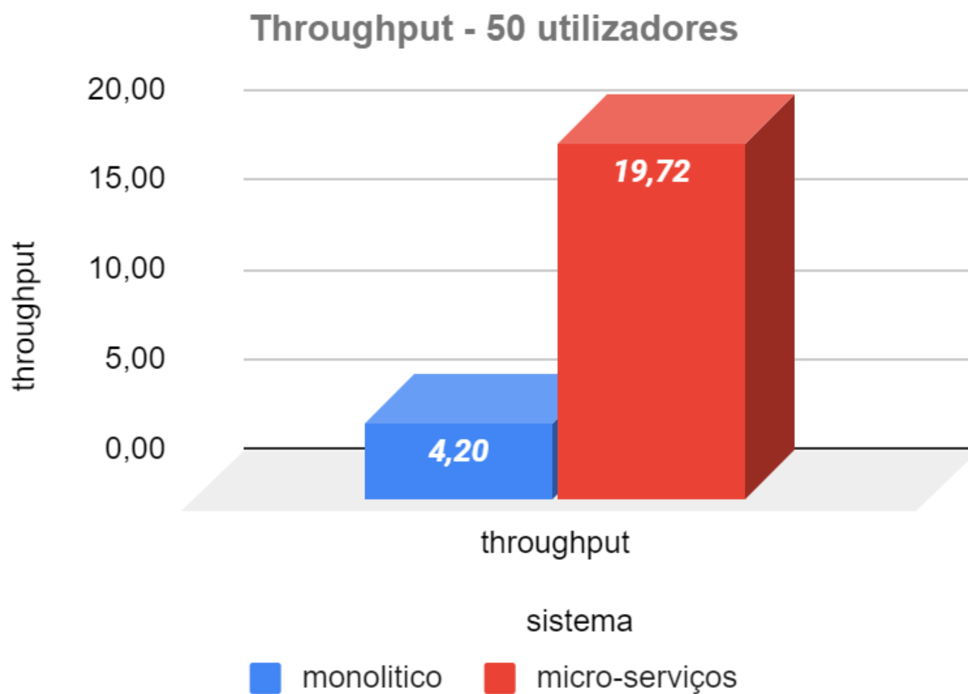


Figura 33 - Resultado médio do throughput para 50 utilizadores

Pode-se concluir, a partir dos gráficos acima (figuras 32 e 33), que o desempenho do sistema de micro-serviços, nesta sequência de testes, foi bem superior ao desempenho do sistema monolítico:

- a nível da latência, o sistema micro-serviços obteve um resultado 844,47% melhor do que o monolítico.
- a nível do throughput, o sistema micro-serviços obteve um resultado 369,52% melhor do que o monolítico.

Testes 2.2 - Teste com 250 utilizadores simulando um cenário de alto estresse;

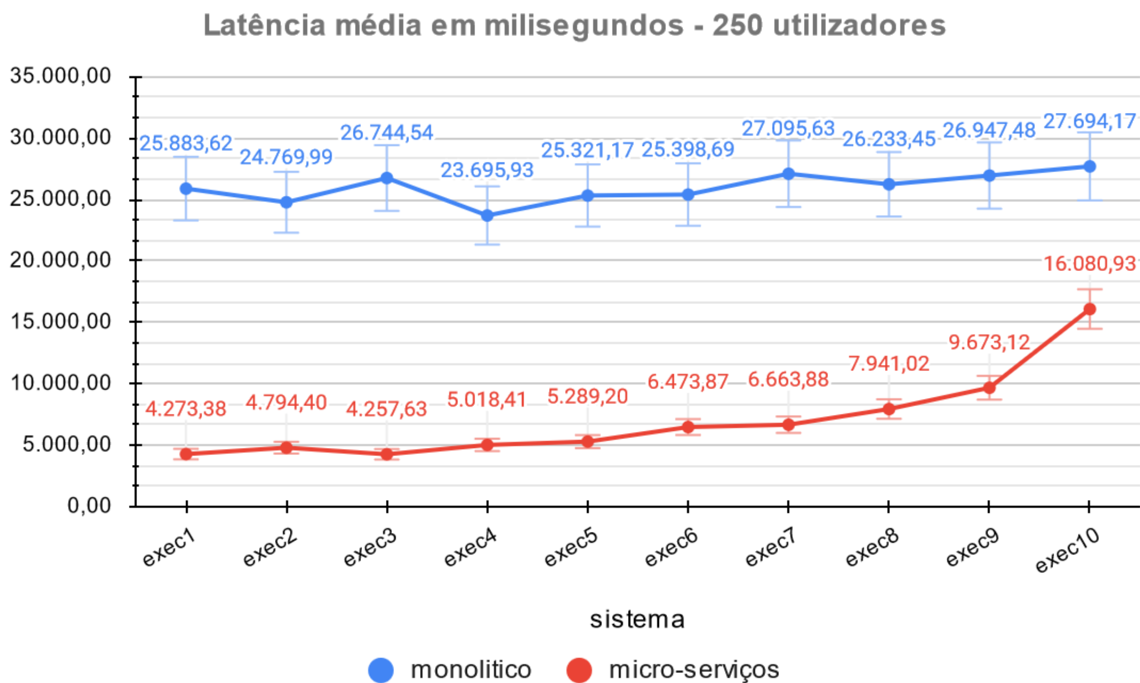


Figura 34 - Resultado da latência média por execução para 250 utilizadores

Através do gráfico na figura 34 acima pode ser verificado que;

- o sistema de micro-serviços apresentou um melhor desempenho da latência do que o sistema monolítico, iniciando a uma diferença de 505%, entretanto a cada nova execução sequencial os micro-serviços apresentavam uma tendência de perda de desempenho mais acentuada do que o monolítico e dessa forma o resultado chegou na 10ª execução com uma diferença de 72%.

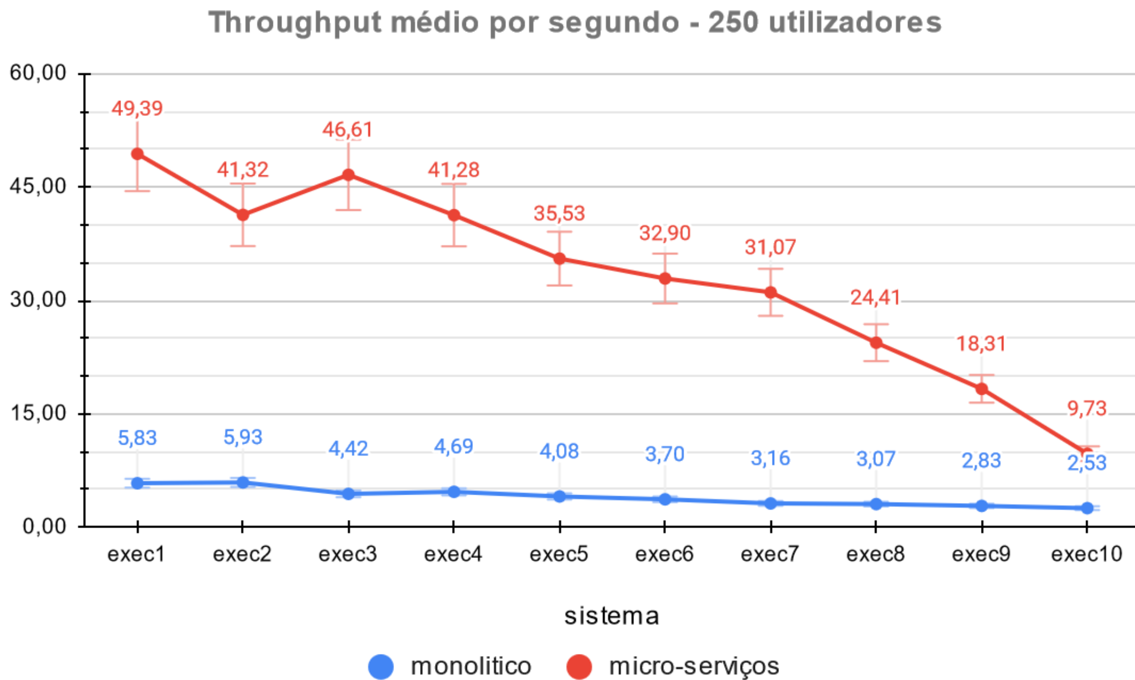


Figura 35 - Resultado de throughput médio por execução para 250 utilizadores

Através do gráfico na figura 35 acima pode ser verificado que;

- o sistema de micro-serviços apresentou um melhor desempenho de throughput do que o sistema monolítico, iniciando com uma diferença de 747,16%, entretanto a cada nova execução os micro-serviços apresentaram uma tendência de perda de desempenho mais acentuada do que o monolítico e dessa forma o resultado chegou na 10ª execução com uma diferença de 284,58%.
- A queda de rendimento dos micro-serviços está provavelmente relacionada ao facto verificado de que o consumo de memória no cluster do GKE não passava por um processo de “regeneração” imediatamente após a finalização de cada execução dos testes.

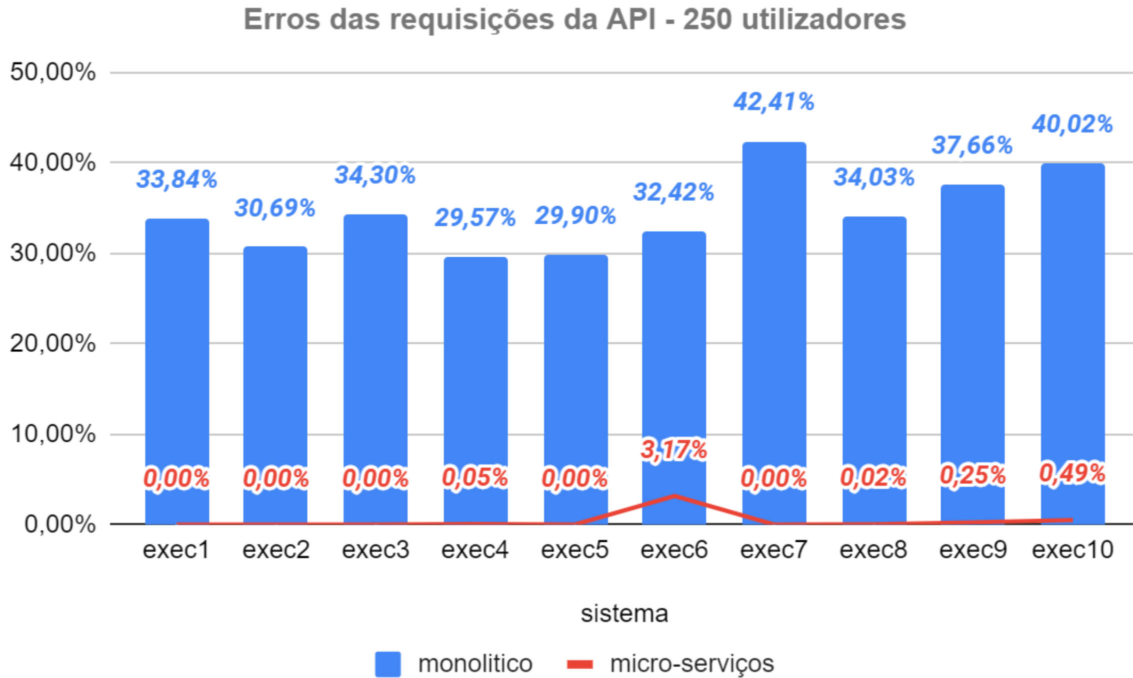


Figura 36 - Resultado de % de erros por execução para 250 utilizadores

Através do gráfico na figura 36 acima pode ser verificado que;

- o sistema de micro-serviços passou a apresentar em algumas execuções um pequeno percentual de erros, o que não ocorreu com apenas 50 utilizadores;
- A taxa de percentual de erros do sistema de micro-serviços na execução6, a de menor desempenho, foi ainda assim 10 vezes mais confiável do que o sistema monolítico.

A seguir, apresentamos uma comparação dos valores das médias dos resultados de testes com 250 utilizadores.

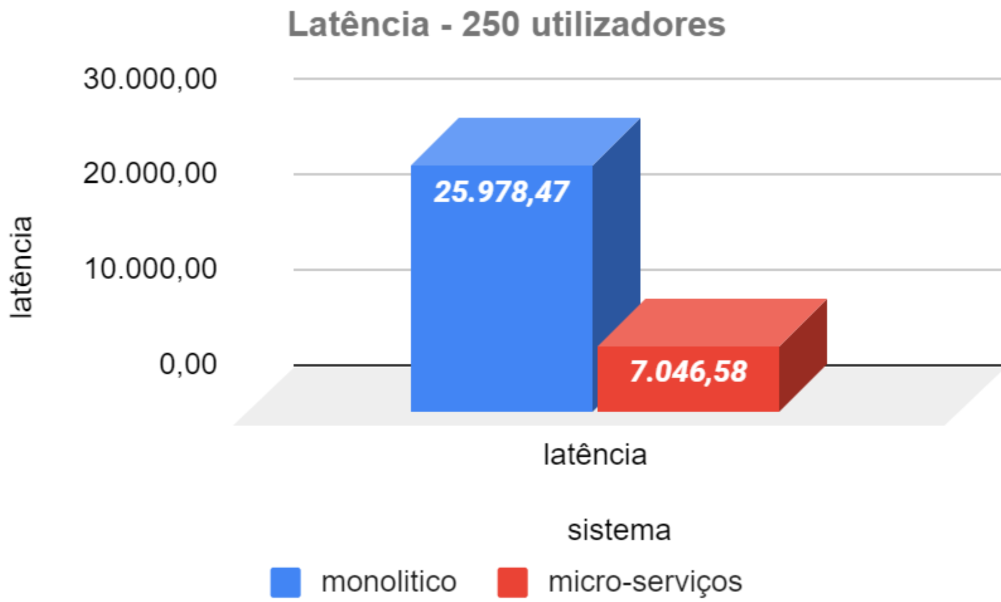


Figura 37 - Resultado médio da latência para 250 utilizadores

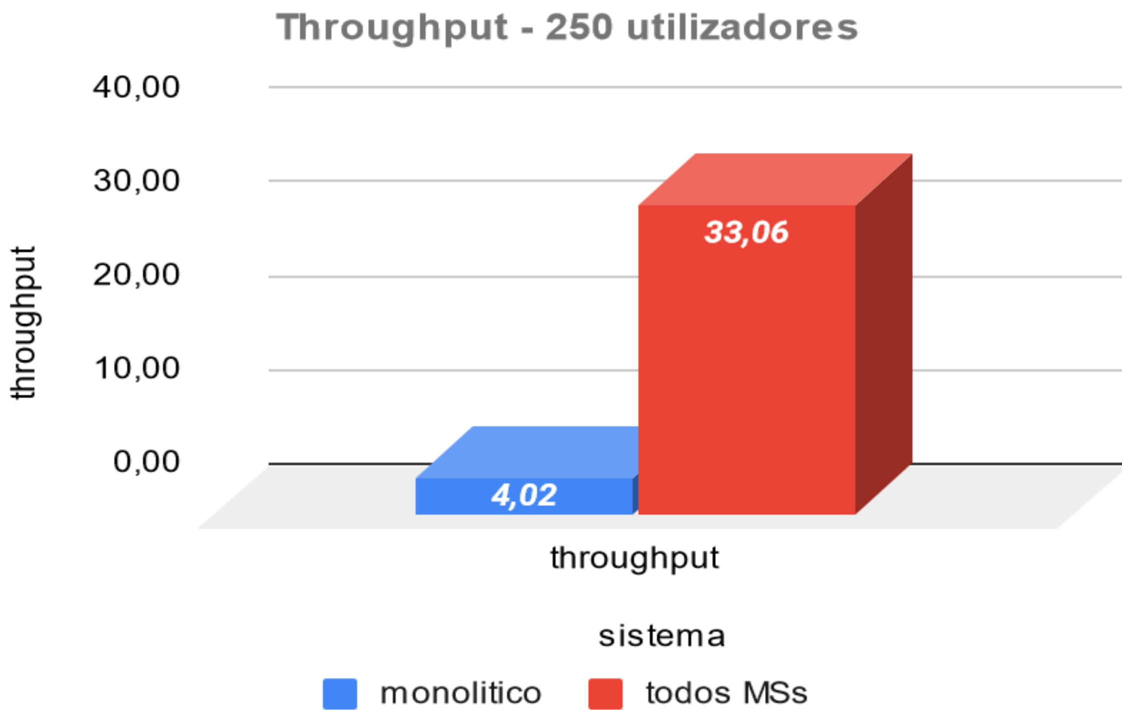


Figura 38 - Resultado médio do throughput para 250 utilizadores

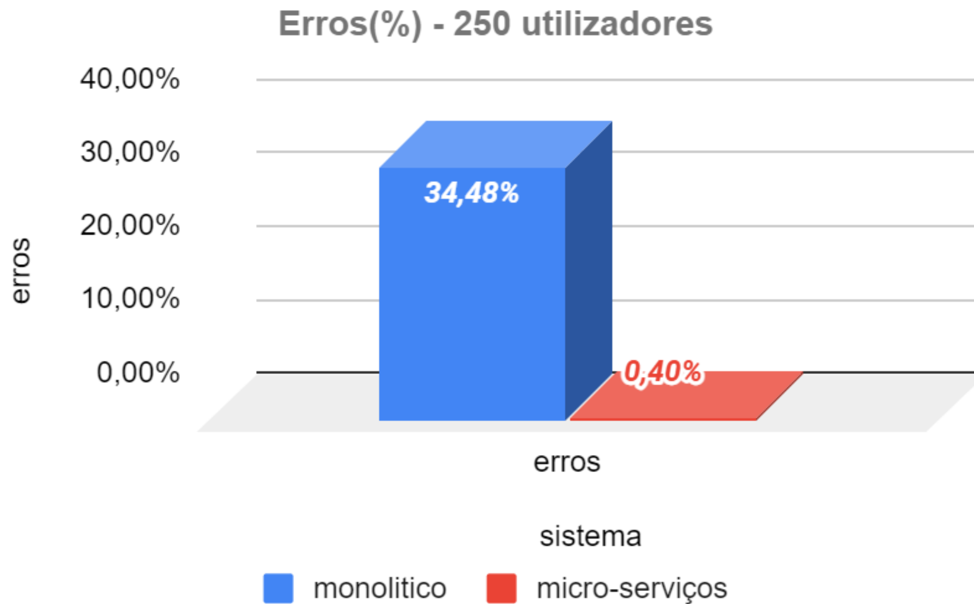


Figura 39 - Resultado médio do % de erros para 250 utilizadores

Pode ser concluído, a partir dos gráficos das figuras 37, 38 e 39, que o desempenho do sistema de micro-serviços foi bem superior ao desempenho do sistema monolítico:

- A latência dos micro-serviços obteve um resultado 268,66% melhor do que a latência do monolítico.
- O throughput dos micro-serviços obteve um resultado 722,38% melhor do que o throughput do monolítico.
- A média do percentual de erros ocorridos demonstra que o sistema de micro-serviços foi quase 100% confiável, enquanto o sistema monolítico teve uma confiabilidade de apenas 65,52%.

Todavia verifica-se que o sistema de micro-serviços teve uma tendência a perder esta vantagem a cada nova execução de testes. Esta degradação está possivelmente relacionada com o facto do cluster do GKE estar a compartilhar os seus recursos computacionais entre os micro-serviços em si, bem como as bases de dados MongoDB e o RabbitMQ, o que não acontece no caso do sistema monolítico. Outro fato é o analisado, e já mencionado junto a figura 35, da não “regeneração” do consumo de memória pelo GKE.

7.3.4 - Teste 3 - Teste de desempenho com 3 micro-serviços desativados

A intenção deste teste foi analisar o comportamento e desempenho do sistema de micro-serviços no caso em que algum destes não estejam em pleno funcionamento. Este tipo de teste tem como fundamento a seção 4.3 - Resiliência de software.

Um micro-serviço pode estar desativado por inúmeras razões: mau funcionamento por um problema qualquer, devido a manutenção, ou a uma decisão por economia de recursos computacionais/financeiros, ou pode não estar realmente desativado e entretanto haver retornos de erros de exceção por problemas na programação ou mesmo a falha na comunicação por problemas de Rede.
Neste caso específico, a intenção foi simular uma decisão por economia de recursos computacionais/financeiros e os micro-serviços escolhidos foram Courses, Applicants e Applications.

As razões para a escolha da desativação destes micro-serviços levou em consideração obviamente o modelo de negócio do sistema de gestão de alunos no âmbito universitário, e também cenários hipotéticos.

- Courses foi desativado porque é possível que uma universidade não fique a criar novos cursos, e a atualizar os dados dos cursos já existentes, constantemente. Applicants e Applications foram desativados levando em consideração que uma universidade possua um tempo limite (data de início e data de fim) sobre o processo de candidatura com aceitação de novos candidatos. Ou seja, isso significa que não haveria a necessidade destes micro-serviços consumirem recursos computacionais/financeiros constantemente.
- Para os cursos, essa estratégia pode ser adotada porque o micro-serviço Disciplines poderia ser utilizado para a pesquisa de cursos uma vez que uma disciplina armazena os dados do curso pertencente. Para os candidatos e candidaturas, o micro-serviço Students poderia ser utilizado para a pesquisa de candidatos e de candidaturas uma vez que um aluno armazena dados do seu respectivo processo de candidatura e seu registro como candidato.

Vejamos então os resultados do teste com 3 micro-serviços desativados em comparação aos outros testes anteriores com 250 utilizadores.

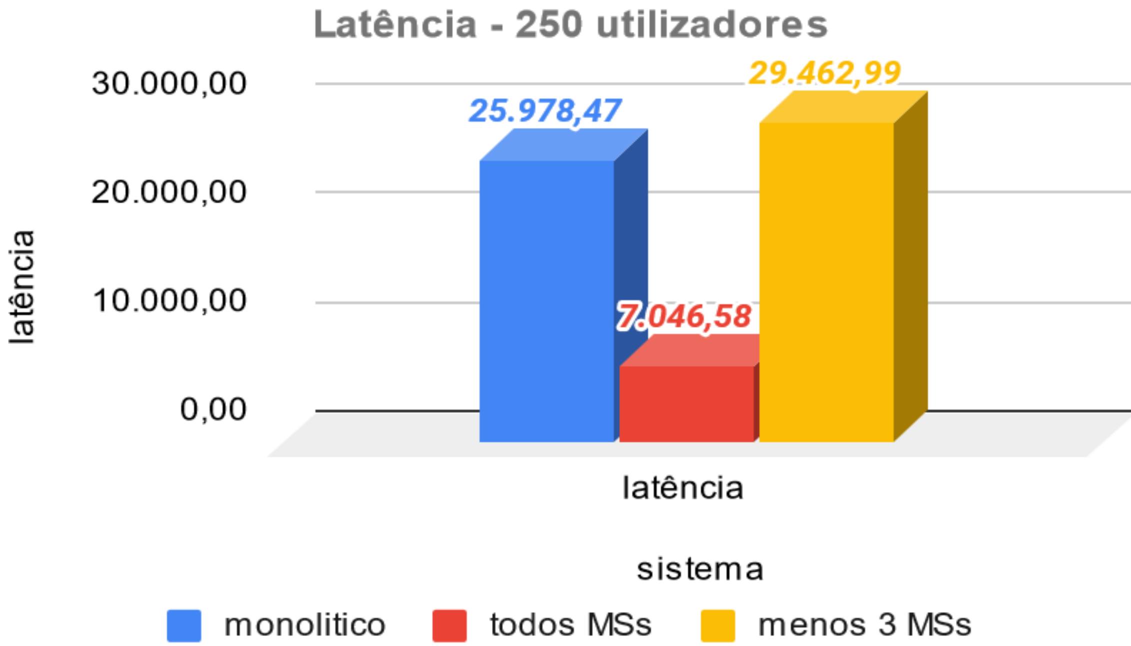


Figura 40 - Resultado médio da latência para 250 utilizadores

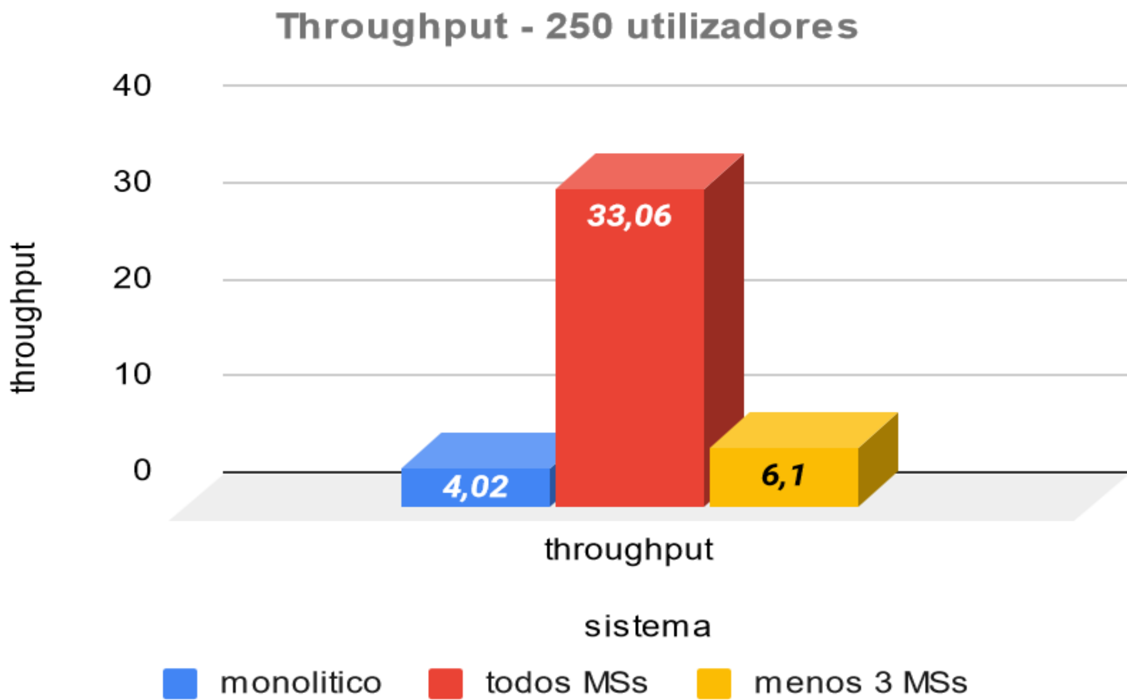


Figura 41 - Resultado médio do throughput para 250 utilizadores

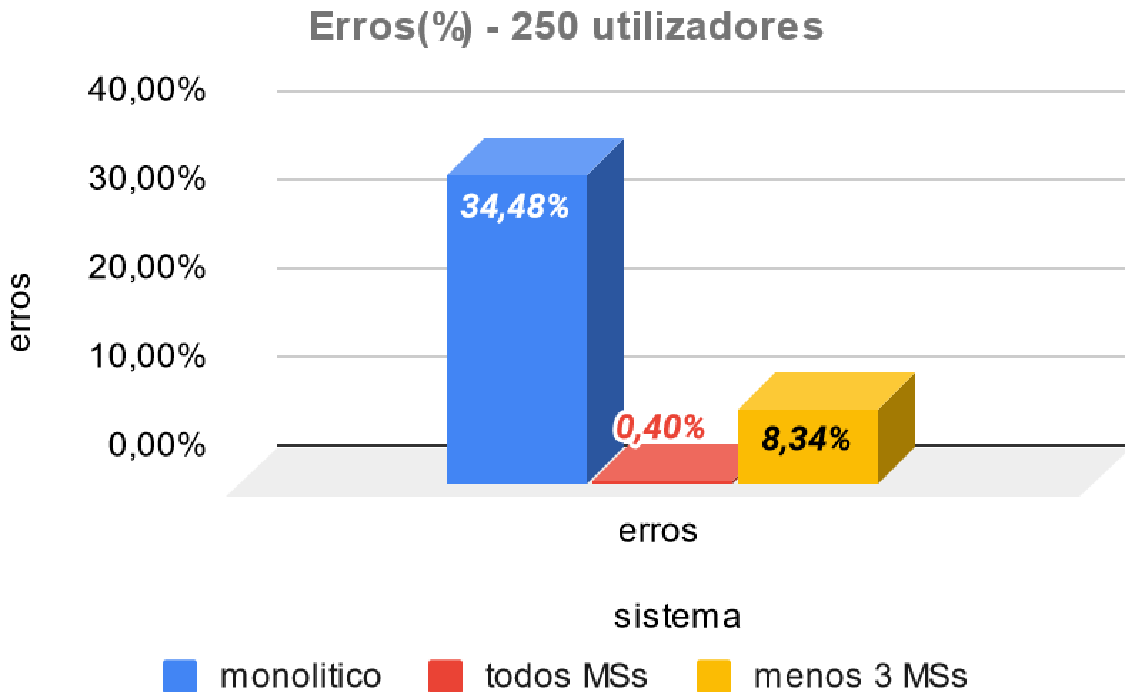


Figura 42 - Resultado médio do % de erros para 250 utilizadores

Pode ser analisado a partir dos resultados nos gráficos das figuras 40, 41 e 42, que:

1. *entre ambos os testes de micro-serviços*: o desempenho do sistema com menos 3 serviços foi inferior aos resultados anteriores do mesmo sistema.

   Esse resultado talvez possa ter ocorrido devido aos seguintes fatores: 1.1- o sistema de micro-serviços já demonstrava uma perda de desempenho a cada nova execução, como descrito junto a figura 35, por motivo da não regeneração dos recursos de memória do GKE; 1.2- buscar todos os cursos através da busca de disciplinas e buscar todos os candidatos e candidaturas através da busca de alunos tornou as requisições de busca de Cursos, de Candidatos e de Candidaturas um pouco mais lentas uma vez que estas passavam a consumir cada vez vez mais recursos de memória do GKE e; 1.3- a ocorrência de "*timeouts*" na tentativa de comunicação com os micro-serviços desativados também é um provável ponto de impacto no resultado dos testes.

2. *entre os testes de resiliência (3 micro-serviços desativados) e o sistema monolítico*: o desempenho do sistema com menos 3 serviços não foi superior no critério latência sobre o sistema monolítico, entretanto foi superior nos critérios throughout e de erros.

   Esse resultado talvez possa ser explicado pela combinação dos seguintes fatores: 2.1- O throughput dos testes com menos 3 micro-serviços, mesmo com perda desempenho

considerável, ainda obteve um resultado 51,74% melhor do que o throughput do monolítico; 2.2- A média do percentual de erros apresenta os micro-serviços com quase 91,66% confiável, enquanto o sistema monolítico teve uma confiabilidade de apenas 65,52% e neste ponto há ainda a necessidade de realçar que muitos destes erros não foram durante a execução do Request ou do Response da API, mas são erros reportados pela não execução do total expectável de Requests, e; 2.3- A latência dos micro-serviços obteve um resultado 13,41% pior ao da latência do monolítico, possivelmente impactado pela combinação da não "regeneração" dos recursos de memória pelo GKE, como já mencionado, e também pelo alto percentual de erros do sistema monolítico que ao não executar todos os Requests que eram expectáveis reduziu assim o impacto de sobrecarga dos recursos computacionais para os Requests que foram completados com sucesso.

Após apresentar os resultados e comparação dos testes entre ambos sistemas, a seguir é descrito um pouco sobre o que foi percebido sobre o comportamento do serviço de computação em nuvem durante os testes

7.3.5 - Comportamento da computação em nuvem

Após os testes executados, os resultados obtidos, colhidos e os gráficos de comparação feitos, o autor deste trabalho tentou identificar no ambiente da computação em nuvem alguma informação que pudesse auxiliar na explicação, da forma mais plausível possível, dos resultados dos testes.

Como descrito algumas vezes na seção 7.3.4, foi analisado e verificado que o GKE não realizava um processo de "regeneração" do seus recursos de memória logo após o fim de uma execução de testes. E as execuções dos testes foram feitas sequencialmente com tempo de intervalo de segundos ou minutos entre cada uma delas.

Vejamos o gráfico a seguir para perceber um pouco o cenário dos testes e o comportamento ocorrido no ambiente de computação em nuvem.

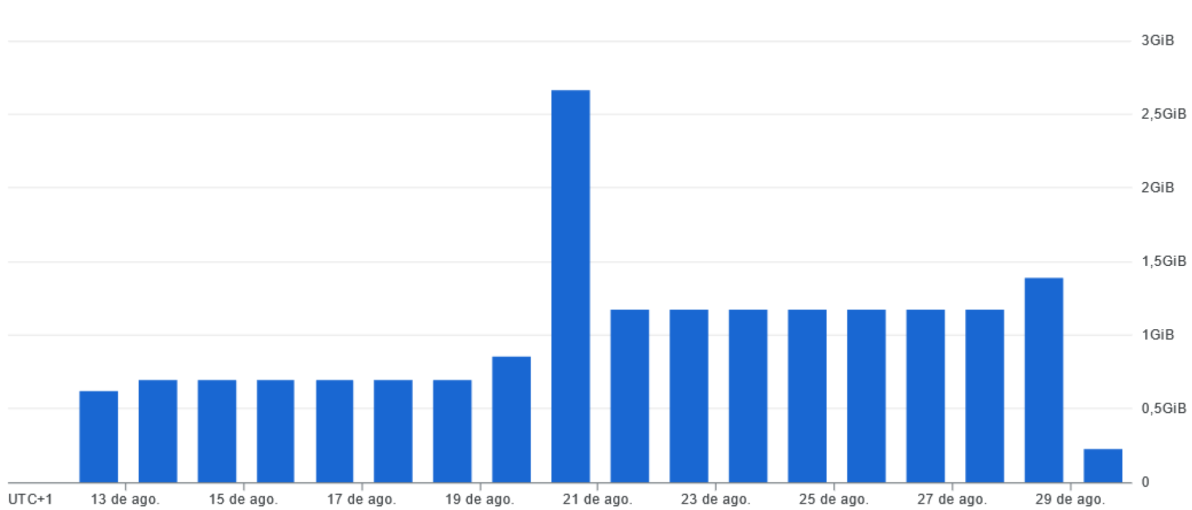


Figura 43 – Uso de memória do micro-serviços disciplines no GKE

O gráfico na figura 43 acima apresenta o comportamento e consumo de memória pelo micro-serviço de turmas no GKE. O mesmo comportamento ocorreu também com os micro-serviços de alunos e professores.

A primeira coluna é a data de 12/08/2021, quando foram executados os testes de desempenho do sistema de micro-serviços e pode ser verificado que o GKE não executou nenhum procedimento de “limpeza” ou “regeneração” de sua memória durante 8 dias.
A coluna maior é a data de 20/08/2021, quando foram realizados os testes com 3 micro-serviços desativados, seguindo assim como descrito na seção 7.3, entre as etapas 2 e 3 da execução dos testes. Pode ser verificado que o GKE levou 9 dias para executar um procedimento de “limpeza” ou “regeneração” de sua memória.

Este não era o comportamento expectável da gestão dos recursos pelo GKE. Imaginava-se que ao fim de cada execução dos testes no JMeter esta “regeneração” dos recursos computacionais fizessem uma “limpeza” da memória consumida em poucos minutos.
Fica neste ponto a dúvida se as configurações utilizadas para a criação dos serviços no Kubernetes foram feitas corretamente ou se este é um comportamento padrão da plataforma para estar preparada e tentar otimizar o seu uso novamente.

7.3.6 - Custos da computação em nuvem

Após todos os testes executados e os resultados consolidados, foi possível então obter as informações de custos financeiros da prova de conceito para ambas as arquitecturas no ambiente da computação em nuvem.

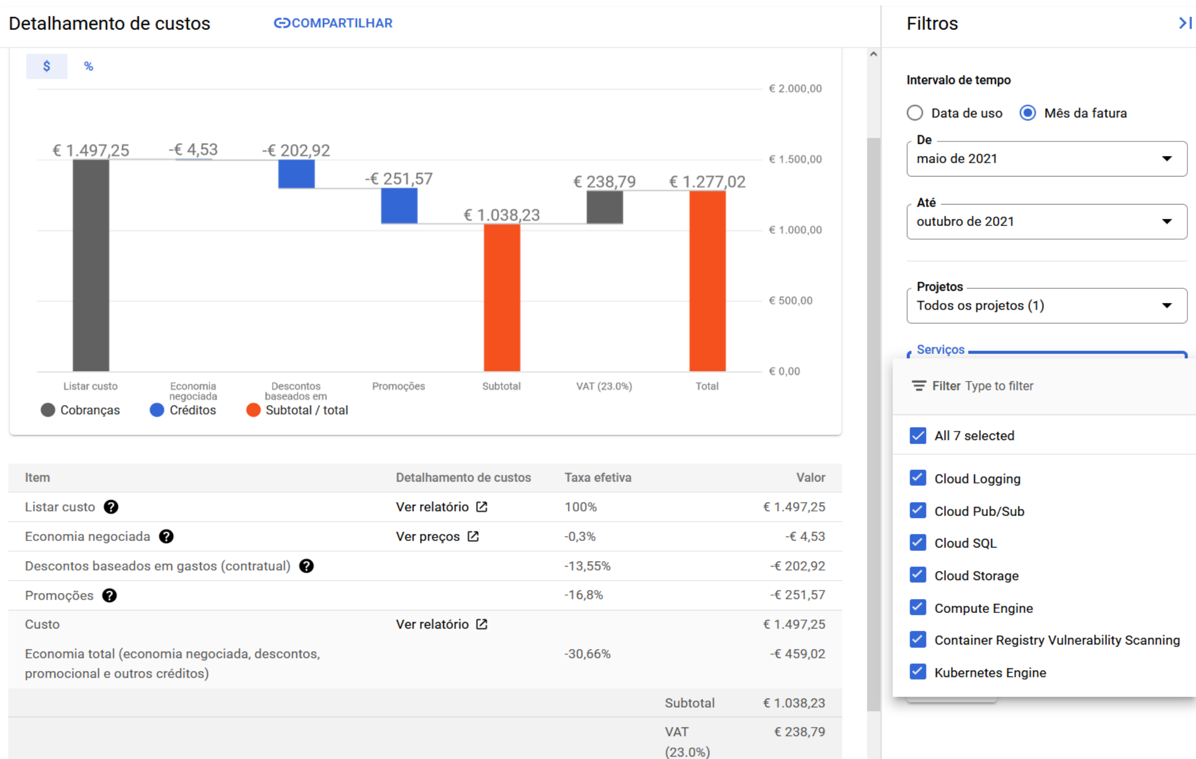
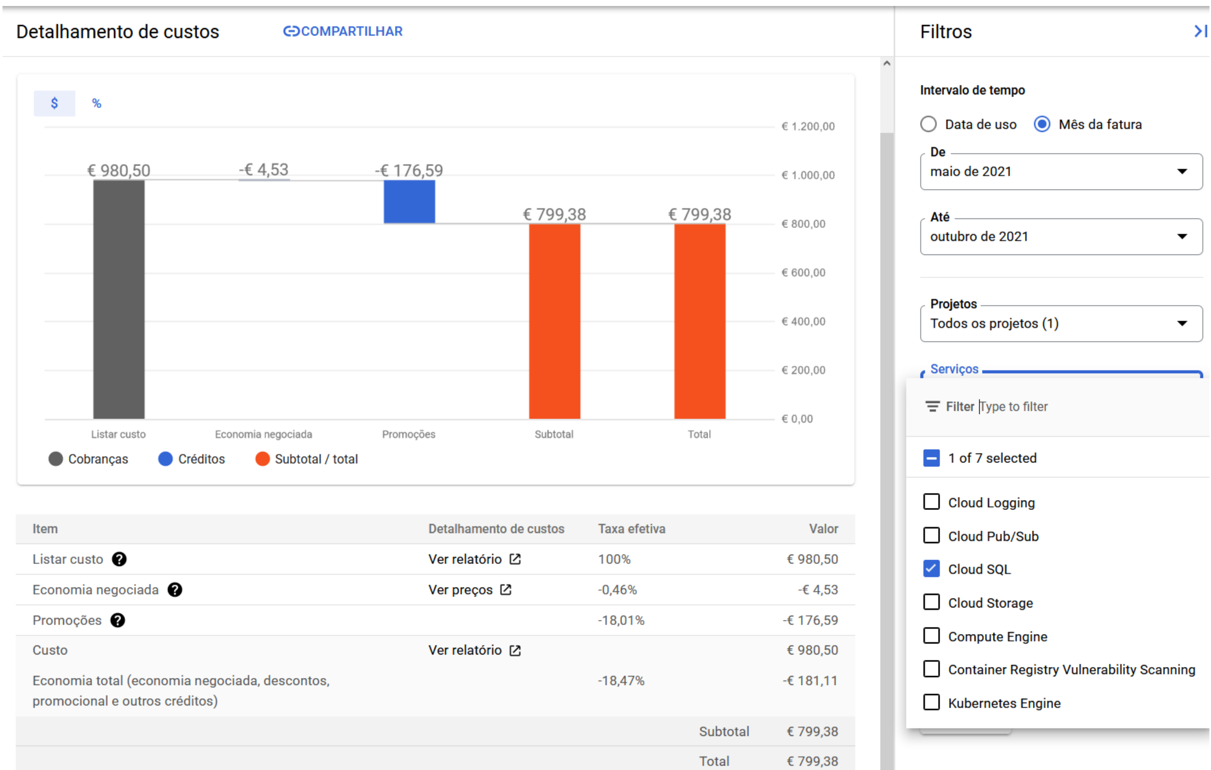


Figura 44 – Custo total da prova de conceito na Google Cloud

A figura 44 acima apresenta o custo total da prova de conceito em €1.277,02 (€ 1.038,23 + € 238.79 de IVA) para ambas as arquitecturas no ambiente da Google Cloud.
Uma vez que os Sistemas Monolítico e de Micro-serviços utilizaram quase totalmente os mesmos recursos de Serviços utilizados da Google Cloud, como podem ser vistos selecionados na imagem 44 acima, não é possível identificar com precisão o custo específico de cada Sistema separadamente.

Entretanto, é possível identificar o custo separado para o Sistema Monolítico relativo ao serviço da Cloud SQL para a Base de Dados Relacional na Google Cloud.
Este custo foi de € 799,38, conforme a imagem 45 abaixo.



Item	Detalhamento de custos	Taxa efetiva	Valor
Listar custo	Ver relatório	100%	€ 980,50
Economia negociada	Ver preços	-0,46%	-€ 4,53
Promoções		-18,01%	-€ 176,59
Custo	Ver relatório		€ 980,50
Economia total (economia negociada, descontos, promocional e outros créditos)		-18,47%	-€ 181,11
		Subtotal	€ 799,38
		Total	€ 799,38

Figura 45 – Custo da Base de Dados Relacional com a Cloud SQL na Google Cloud

Este custo somente com a Cloud SQL, consequentemente com a Base de Dados Relacional em MySQL, representou aproximadamente 77% do custo total (sem IVA) com a Google Cloud.

Capítulo 8 - Conclusão e trabalhos futuros

A primeira conclusão que pode ser apresentada sobre este trabalho de dissertação é que, como percebido ao longo de seu conteúdo, a diferença entre as arquiteturas de software monolítico e de micro-serviços não são poucas e não são triviais.

De tal forma, também pode ser concluído que as decisões de projetos organizacionais que envolvam, e/ou tenham como base, a área de tecnologia da informação precisarão ser apresentados de forma mais clara possível pela equipe de negócios, os requisitos para responder às questões: 1- Qual a demanda prevista ou projetada de utilizadores do sistema? 2- As regras de negócio podem ser disponibilizadas parcialmente, em etapas? 3- Quais são os critérios e parâmetros de aceitação sobre o desempenho e confiabilidade do sistema? Estas questões são importantes para que as equipes de tecnologia de arquitetura, infraestrutura e de desenvolvimento possam avaliar e mensurar da melhor forma possível qual arquitetura de software adotar para atender o(s) projeto(s) idealizado(s) pela equipe de negócios. E também é importante um "*feedback*" da equipe de tecnologia apresentando e explicando para a equipe de negócios a arquitetura de software decidida.

As respostas para as 3 questões irão resultar em diferentes tipos possíveis de sistemas. Como exemplo podemos ter um sistema de acesso restrito aos colaboradores de uma organização, com base em funcionalidades do tipo "CRUD", sem muitas regras de negócio e com centenas de utilizadores simultaneamente ou podemos ter um sistema para processamento batch de um alto volume de dados, no qual não haverá nenhum utilizador e que os dados processados deverão ser disponibilizados para outros aplicativos internos e externos.

É nítido que nos dias atuais as organizações tenham mudado sua forma de planejar e agir sobre seus projetos de negócio suportados pela tecnologia, saindo de *sistemas robustos e completos com grande quantidade e complexidade de regras de negócio, planejados para um longo prazo de criação até à disponibilização em ambiente produtivo*, e indo em direção para *sistemas com menor quantidade e complexidade de regras de negócio que podem ser incrementadas ao longo do tempo, e planejados para um curto prazo até a disponibilização em ambiente produtivo*. E essa mudança de mentalidade das organizações, que ocorreu para acompanhar uma maior competitividade, e de alcance global, também é um dos fatores

que têm influenciado a evolução do desenvolvimento de software para responder às questões levantadas no parágrafo anterior e para ao mesmo tempo otimizar os recursos financeiros das organizações.

Todavia, ainda sobre o fato da diferença entre as arquiteturas monolítica e de micro-serviços não serem poucas e triviais, a decisão de qual arquitetura de software ser adotada para um, ou mais, projeto(s) deve ter em conta um fator inerente, para além do entendimento entre as equipes técnicas e de negócio. Esse fator é o "*Know-How*" da equipe de tecnologia para desenvolver e manter os sistemas. Desta forma, há dois pontos tecnicamente a serem levantados.
1º- DE: desenvolvimento de sistema monolítico com programação imperativa e base de dados relacional. PARA: um sistema de micro-serviços com programação reativa, com mais de uma base de dados do modelo NoSql e ainda orientados a eventos é de fato algo que exige uma mudança na forma de pensar da equipe de desenvolvimento.
2º- As configurações e parametrizações necessárias no serviço de computação em nuvem sobre as arquiteturas desenvolvidas não foram simples e triviais. E a arquitetura de micro-serviços tornou as configurações do serviço de computação em nuvem um pouco mais complexas e trabalhosas do que para a arquitetura monolítica, vide suas diferenças estruturais nas figuras 3 e 5. Sob a ótica das equipes de arquitetura e de infraestrutura a manutenção de um sistema monolítico realmente é mais fácil e menos "doloroso" do que manter sistema de micro-serviços e seus componentes associados, entretanto a tecnologia também vem evoluindo para oferecer soluções que auxiliem e facilitem este trabalho o mais automatizado e eficientemente possível.

Pondo isto, para o autor desta dissertação, pode ser concluído de qualquer forma que estes dois pontos levantados não deveriam ser fatores avaliados como limitadores pelas equipes de arquitetura, infraestrutura e de desenvolvimento, sobre a decisão de qual tipo de arquitetura adotar. Uma vez obtido o conhecimento necessário para desenvolver e manter um tipo de arquitetura de software, este conhecimento tende apenas a crescer e melhorar.

Os testes de desempenho que foram realizados atenderam ao objetivo de fornecer dados para análise e comparação de ambos sistemas.
A ferramenta do JMeter apresentou vários desafios técnicos mas no final, após inúmeras tentativas e erros, foi possível chegar a configuração do plano de testes para execução de

forma dinâmica da API Rest e também concluir e obter os resultados de todos os testes executados.

É preciso ressaltar que os resultados dos testes, para ambos os sistemas, apresentaram desempenhos surpreendentemente inferiores aos expectáveis pelo autor desta dissertação. Exemplos para isso são: 1º- não era expectável que o sistema monolítico apresentasse um percentual em volta de 10% de erros na execução das requisições da API Rest para o cenário de baixo estresse com apenas 50 utilizadores; 2º- não era expectável que o sistema de micro-serviços começasse a perder tanto desempenho de uma execução para outra mesmo no cenário de alto estresse com 250 utilizadores.

Os maiores impactos negativos apresentados nos testes de ambos os sistemas foram as requisições da API do tipo de “Busca Entidades/Objetos de negócio”, que são “Search Courses”, “Search Disciplines”, “Search Applicants”, “Search Applications”, “Search Professors” and “Search Students”, vide figura 22.

De fato já era expectável que estas pudessem ser as requisições com maiores problemas de desempenho na API Rest para ambos os sistemas, mas há a dúvida se o desempenho menos eficiente do que o expectável foi causado por alguma má configuração e uso da ferramenta JMeter ou do serviço Google Cloud.

Ainda é importante ressaltar que o facto do custo total com a Google Cloud ter sido aproximadamente 77% com somente a Base de Dados Relacional, sob o serviço Cloud SQL, traz ao aluno autor deste trabalho uma avaliação de que relação de custo/benefício do sistema de micro-serviços é superior ao sistema monolítico. Há aqui neste ponto também alguma margem para dúvida se a configuração dos serviços da Cloud impactaram de alguma forma neste resultado de eficiência financeira.

Por este motivo, o autor desta dissertação conclui que os testes foram realizados com sucesso, mas com ressalvas por falta de opinião e conhecimento especializados na ferramenta do JMeter e no serviço Google Cloud para corroborar um cenário bem configurado e definido para o uso de ambos recursos.

No estudo de caso deste trabalho, sobre o modelo de negócio de um sistema de gestão de alunos no âmbito universitário, foi possível apresentar e perceber como a tomada de

decisão sobre qual arquitetura de software utilizar poderia realmente fazer sentido e ser útil para obter uma possível redução de custos ou obter uma resiliência e confiabilidade maior na utilização do sistema por seus utilizadores (internos e externos).

É importante ressaltar que, propositalmente, não foram realizadas análises sobre a quantidade de recursos de processamento, de memória, de rede ou leitura e escrita de bases de dados, que cada arquitetura de software consumia durante a execução dos testes. Os recursos computacionais disponíveis foram, aproximadamente, os mesmos para ambas arquiteturas, pois a intenção foi a de analisar a questão: dado um cenário de recursos computacionais disponíveis para a arquitetura monolítica, poderia a arquitetura de micro-serviços obter um desempenho de melhor custo/benefício para atender ao modelo de negócio proposto? A resposta para o autor deste trabalho é Sim. E será justificada na seção seguinte (9.1).

8.1 - Conclusão Final

A intenção do autor desta dissertação de contribuir para a comunidade acadêmica e tecnológica pode ser considerada como alcançada com sucesso, uma vez que foi possível através deste trabalho adquirir o conhecimento, e compartilhar este conhecimento adquirido, sobre as arquiteturas de software monolítica e de micro-serviços, detalhando e comparando suas estruturas e suas diferenças bem como apresentando e comparando os resultados dos seus testes de desempenho.

A conclusão de que a contribuição deste trabalho pode ser considerada como realizada com sucesso tem como suporte o contexto teórico desenvolvido através de muitos estudos e pesquisas que foram compiladas e aqui repassadas. Contudo, o que pode ser considerado como o fator preponderante para a contribuição deste trabalho, é o contexto prático através de um método empírico de concepção, desenvolvimento e realização dos testes de ambas as arquiteturas.

O principal motivo deste trabalho de dissertação foi demonstrar que a arquitetura de micro-serviços pode ser mais eficiente do que a arquitetura monolítica, mesmo que seja mais complexa e possa exigir um pouco mais de investimento no capital intelectual humano.

Os resultados dos testes realizados sobre ambos cenários de baixo e de alto estresse apresentaram resultados sólidos, quanto às métricas analisadas de Latência, Throughput e Erros de Request, como foram demonstrados nas seções 7.3.3 e 7.3.4. Resultados estes que permitem ao autor concluir com ressalvas que a arquitetura de micro-serviços é bem mais eficiente do que a arquitetura monolítica. Alguns fatos que suportam essa conclusão são:

1º- o modelo de negócio abordado e desenvolvido, de um sistema de gestão de alunos, pode ser considerado como simples e portanto era expectável que tivesse melhores resultados de desempenho do sistema monolítico perante o sistema de micro-serviços sobre o cenário de baixo estresse com apenas 50 utilizadores. E também não era expectável que ocorresse 12% de erros das requisições para o sistema monolítico.

2º- no cenário de alto estresse com 250 utilizadores, o sistema de micro-serviços aparentemente teve uma queda significativa de desempenho da latência e do throughput a cada execução ao ser comparado ao sistema monolítico, entretanto o sistema monolítico apresentou uma altíssima taxa percentual de erros, como pode ser revisto no gráfico da figura 35, indo de 30% a 42% com uma média de 34%. Muitos destes erros representam não somente erros de Request ou Response, mas também representam requisições que não foram realizadas, o que pode ser concluído como uma ineficiência com alta possibilidade de má experiência dos utilizadores.

3º- dos 8 micro-serviços desenvolvidos 3 foram desativados para a execução de testes de resiliência. Estes testes foram realizados com sucesso demonstrando que mesmo com 37,5% dos micro-serviços sem responder, o sistema de gestão de alunos no modelo de micro-serviços continuava a funcionar e responder às requisições da API que não dependiam diretamente destes 3 serviços desativados. Uma vez que para estes testes o desempenho dos micro-serviços tenha sido mais eficiente do que o monolítico pode permitir concluir que os recursos computacionais utilizados no serviço de computação em nuvem poderiam ser reduzidos durante alguns meses do ano, reduzindo assim os custos do sistema de gestão de alunos.

4º- o custo financeiro total com os serviços fornecidos pela plataforma de cloud computing, a Google Cloud, foi composta em 77% somente com a Base de Dados Relacional do MySQL e este recurso foi utilizado somente pelo sistema monolítico, o que permite afirmar assim

que a relação custo/benefício do sistema de micro-serviços é superior a relação custo/benefício do sistema monolítico.

8.2 - Trabalhos Futuros

Como possibilidade de trabalho futuro após esta dissertação, sugere-se a implementação de uma estratégia importante, todavia não obrigatória, na arquitetura de micro-serviços, que é a implementação do “Event Sourcing” e em conjunto com este, a implementação do CQRS - Command Query Responsibility Segregation.

Enquanto as bases de dados relacionais possuem as propriedades ACID, como descrito no tópico 2.5.3, nas bases de dados NoSQL esse conceito não existe. Mas a equipe de arquitetura pode decidir implementar algum controle sobre as operações atómicas, ou seja, aquelas operações que alteram o estado de um objeto na base de dados NoSQL através dos processos de criação, alteração ou remoção de objetos.
Para isso utiliza-se o “Event Sourcing”, que é a implementação de uma estratégia para armazenar todas as operações de mudança de estado, de qualquer objeto em qualquer micro-serviço, como um evento numa base de dados específica. Isso irá prover a possibilidade de obter um tipo de "atomicidade” dos objetos pela capacidade de recuperar todas as mudanças de estado e funcionando como um *Event Log* através do histórico das mudanças de estado.

Conceitualmente esta base de dados é chamada de *Event Store*, e não deveria salvar o objeto alterado do micro-serviço em si, mas deveria salvar um conjunto de informações úteis para identificar e obter, o máximo possível, a atomicidade histórica de um objeto como por exemplo o tipo do objeto (ou entidade) salvo, o micro-serviço, o estado do objeto na operação executada - neste ponto entraria a entidade salva no micro-serviço - quando ocorreu a operação, o tipo da transação - criar, atualizar, apagar - e também o motivo da operação. Exemplo: adicionar professor para lecionar disciplina ou registrar nota de aluno em disciplina e etc.

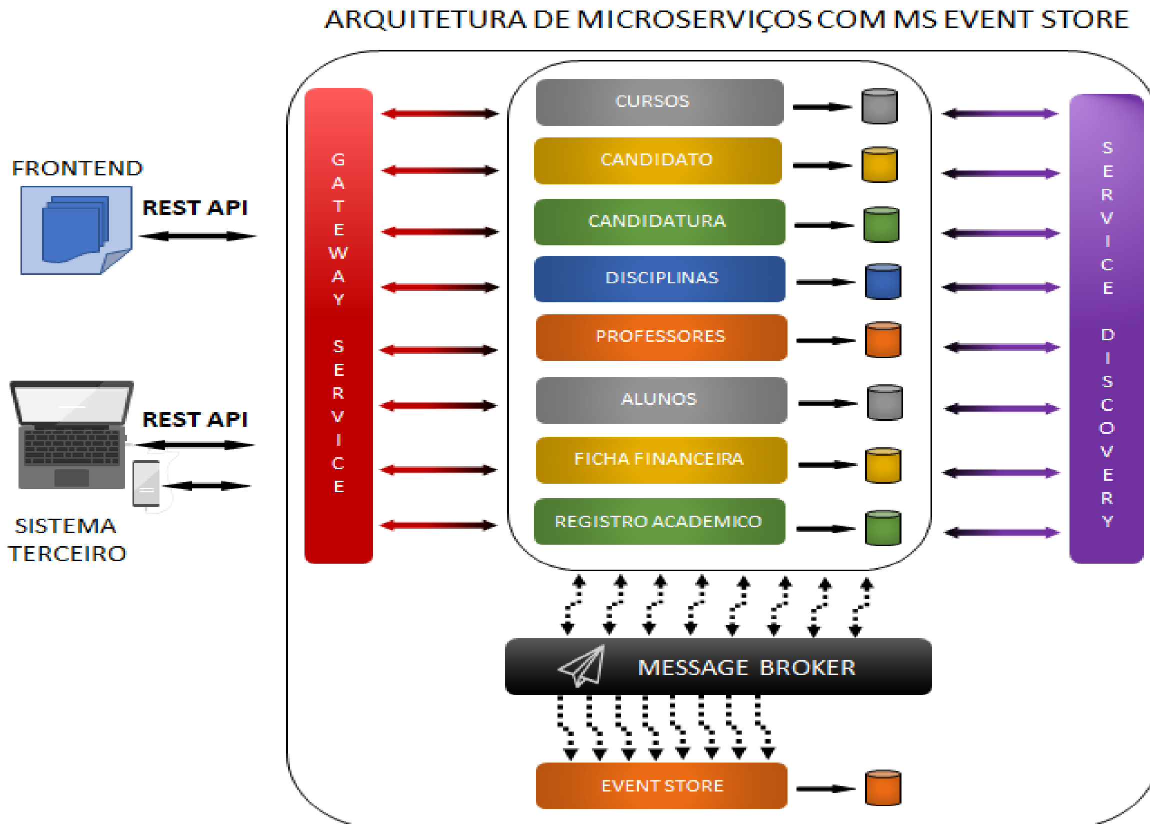


Figura 46 – Arquitetura de micro-serviços com o micro-serviço do EventStore do event sourcing

Na figura 46 acima, a estratégia do Event Sourcing é representada por um micro-serviço chamado de EventStore, e este irá ler e armazenar os eventos do message broker, devidamente identificados, por motivo de mudança de estado de algum objeto.

> “We can query an application's state to find out the current state of the world, and this answers many questions. However there are times when we don't just want to see where we are, we also want to know how we got there.” (Fowler, 2005)

Como pode ser percebido, o EventStore terá uma enorme base de dados com o tempo, e isso irá possivelmente fazer com que as consultas dos dados históricos dos objetos possam impactar diretamente o desempenho do sistema.

Para resolver ao máximo possível esse desafio, é sugerido implementar o padrão CQRS - *Command Query Responsibility Segregation*, no qual a idéia é separar os *Commands* (são transações de mudança de estado) das *Querys* (as consultas dos objetos nas bases de dados), ou seja, separar e otimizar as execuções de escrita e de leitura de dados.

Também fica como sugestão de trabalho futuro a participação de dois especialistas para aprimorar os testes de desempenho. Um especialista sobre a ferramenta Apache JMeter e outro sobre o serviço do Google Cloud.
A intenção é ter a certeza de que ambos estão configurados e preparados corretamente para não impactar de forma negativa no resultado dos testes de desempenho..

Apêndices

Apêndice I – Base de dados NoSQL de alunos


```
{
  "_id" : "8806b33c-1f59-4d2a-807c-d29d99d16c1e",
  "name" : "Students Name 1062853",
  "email" : "student.1062853@email.com",
  "phone" : "98566-4578",
  "applicant" : {
    "registration" : "A532508094"
  },
  "registryAcademic" : {
    "registration" : "RA862751754",
    "disciplines" : [{
        "registration" : "D347806005",
        "name" : "Discipline Name 778",
        "startDate" : ISODate("2020-09-25T23:00:00Z"),
        "endDate" : ISODate("2021-08-24T23:00:00Z"),
        "approved" : false,
        "closed" : false,
        "course" : {
          "registration" : "C657512865",
          "name" : "Course's name 115"
        }
      }]
  },
  "financialStatement" : {
    "_id" : "250a9080-a0cd-40b1-8b4f-a6ccd80bad71",
    "registration" : "FS149253059",
    "listInvoices" : [{
        "referenceNumber" : "312478822",
        "amount" : "37.50",
        "creationDate" : ISODate("2021-05-09T17:03:08.986Z"),
        "expirationDate" : ISODate("2021-05-08T23:00:00Z"),
        "invoiceType" : "TUITION_FEE",
        "status" : "OPEN",
        "description" : "Tuition Fee in 2021-05-09T18:03:08.981"
      }, {
        "referenceNumber" : "723431715",
        "amount" : "37.50",
        "creationDate" : ISODate("2021-05-09T17:03:08.986Z"),
        "expirationDate" : ISODate("2021-06-08T23:00:00Z"),
        "invoiceType" : "TUITION_FEE",
        "status" : "OPEN",
        "description" : "Tuition Fee in 2021-05-09T18:03:08.981"
      }, {
        "referenceNumber" : "388276586",
        "amount" : "37.50",
        "creationDate" : ISODate("2021-05-09T17:03:08.986Z"),
        "expirationDate" : ISODate("2021-07-08T23:00:00Z"),
```

```
      "invoiceType" : "TUITION_FEE",
      "status" : "OPEN",
      "description" : "Tuition Fee in 2021-05-09T18:03:08.981"
    }, {
      "referenceNumber" : "682803363",
      "amount" : "37.50",
      "creationDate" : ISODate("2021-05-09T17:03:08.987Z"),
      "expirationDate" : ISODate("2021-08-08T23:00:00Z"),
      "invoiceType" : "TUITION_FEE",
      "status" : "OPEN",
      "description" : "Tuition Fee in 2021-05-09T18:03:08.981"
    }]
  },
  "creationDate" : ISODate("2021-05-09T17:02:06.601Z"),
  "registration" : "S405098299",
  "_class" : "com.university.dissertation.students.domain.model.Students"
}
```


Apêndice II – Base de dados NoSQL de fichas financeiras

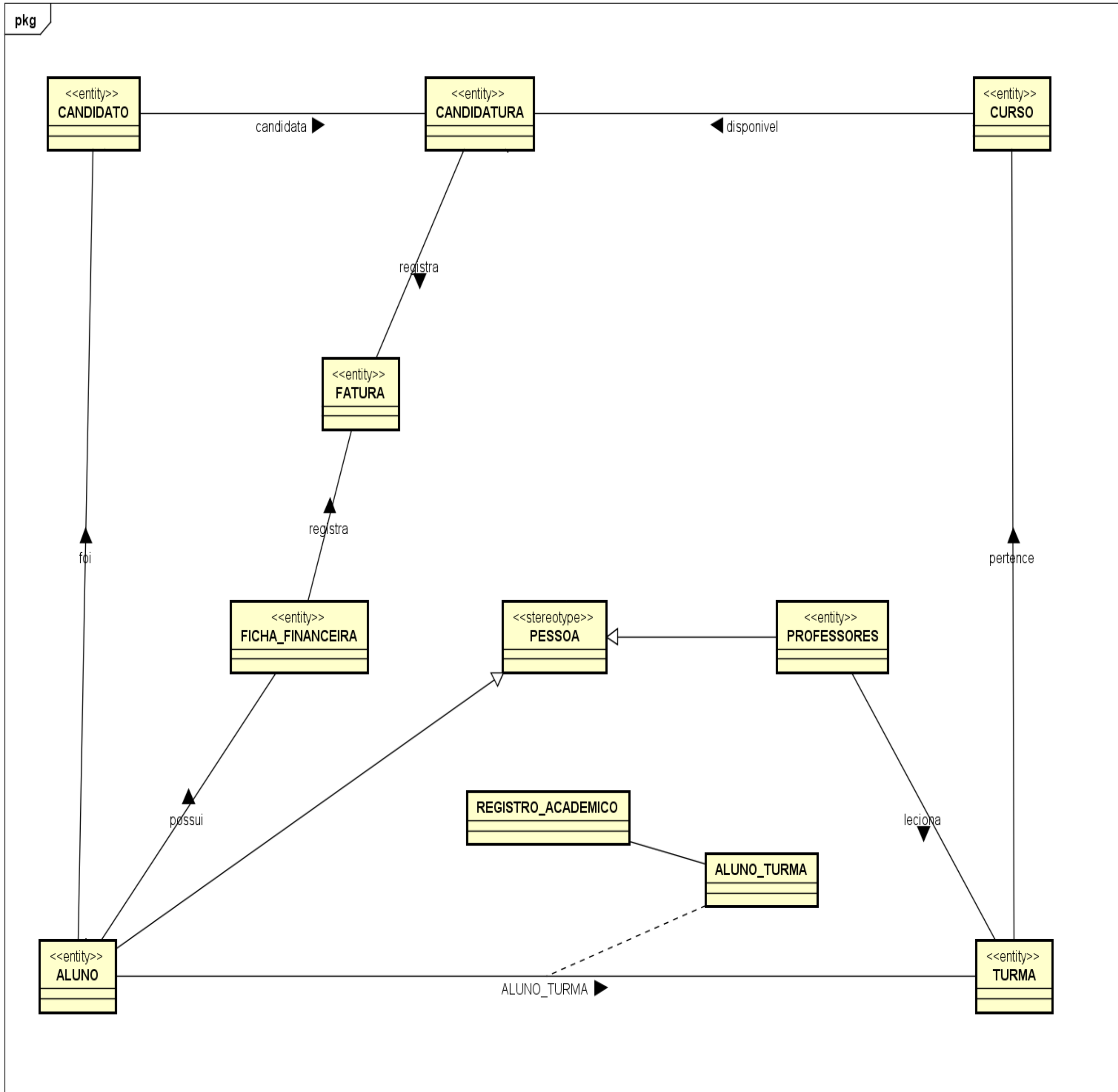

```
{
  "_id" : "250a9080-a0cd-40b1-8b4f-a6ccd80bad71",
  "student" : {
    "registration" : "S405098299",
    "name" : "Students Name 1062853"
  },
  "listInvoices" : [{
      "_id" : "20210509312478822",
      "referenceNumber" : "312478822",
      "amount" : "37.50",
      "creationDate" : ISODate("2021-05-09T17:03:08.986Z"),
      "expirationDate" : ISODate("2021-05-08T23:00:00Z"),
      "invoiceType" : "TUITION_FEE",
      "status" : "OPEN",
      "description" : "Tuition Fee in 2021-05-09T18:03:08.981"
    }, {
      "_id" : "20210509723431715",
      "referenceNumber" : "723431715",
      "amount" : "37.50",
      "creationDate" : ISODate("2021-05-09T17:03:08.986Z"),
      "expirationDate" : ISODate("2021-06-08T23:00:00Z"),
      "invoiceType" : "TUITION_FEE",
      "status" : "OPEN",
      "description" : "Tuition Fee in 2021-05-09T18:03:08.981"
    }, {
      "_id" : "20210509388276586",
      "referenceNumber" : "388276586",
      "amount" : "37.50",
      "creationDate" : ISODate("2021-05-09T17:03:08.986Z"),
      "expirationDate" : ISODate("2021-07-08T23:00:00Z"),
      "invoiceType" : "TUITION_FEE",
      "status" : "OPEN",
      "description" : "Tuition Fee in 2021-05-09T18:03:08.981"
    }, {
```

```
        "_id" : "20210509682803363",
        "referenceNumber" : "682803363",
        "amount" : "37.50",
        "creationDate" : ISODate("2021-05-09T17:03:08.987Z"),
        "expirationDate" : ISODate("2021-08-08T23:00:00Z"),
        "invoiceType" : "TUITION_FEE",
        "status" : "OPEN",
        "description" : "Tuition Fee in 2021-05-09T18:03:08.981"
    }],
  "creationDate" : ISODate("2021-05-09T17:02:14.689Z"),
  "registration" : "FS149253059",
  "_class" :
"com.university.dissertation.financialstatement.domain.model.FinancialStateme
nt"
}
```

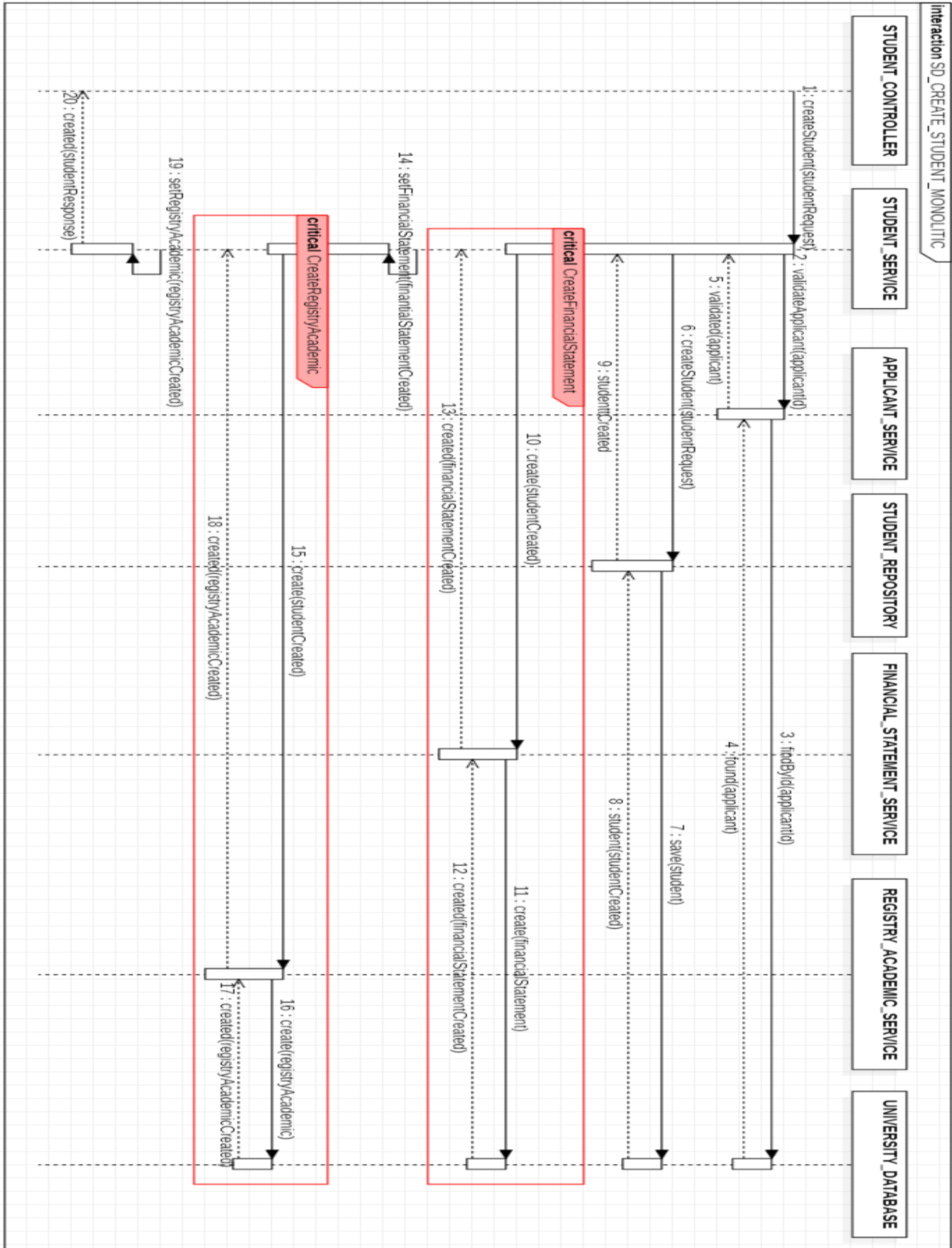

Apêndice III – Base de dados NoSQL de registros acadêmicos


```
{
  "_id" : "76b63d67-a2ce-4ac6-b38b-57cdebde75da",
  "student" : {
    "_id" : "8806b33c-1f59-4d2a-807c-d29d99d16c1e",
    "registration" : "S405098299",
    "name" : "Students Name 1062853",
    "email" : "student.1062853@email.com",
    "phone" : "98566-4578"
  },
  "disciplines" : [{
      "registration" : "D347806005",
      "name" : "Discipline Name 778",
      "startDate" : ISODate("2020-09-25T23:00:00Z"),
      "endDate" : ISODate("2021-08-24T23:00:00Z"),
      "closed" : false,
      "course" : {
        "registration" : "C657512865",
        "name" : "Course's name 115"
      },
      "amount" : "150.00",
      "approved" : false
    }],
  "creationDate" : ISODate("2021-05-09T17:02:20.182Z"),
  "registration" : "RA862751754",
  "_class" :
"com.university.dissertation.registryacademics.domain.model.RegistryAcademics
"
}
```

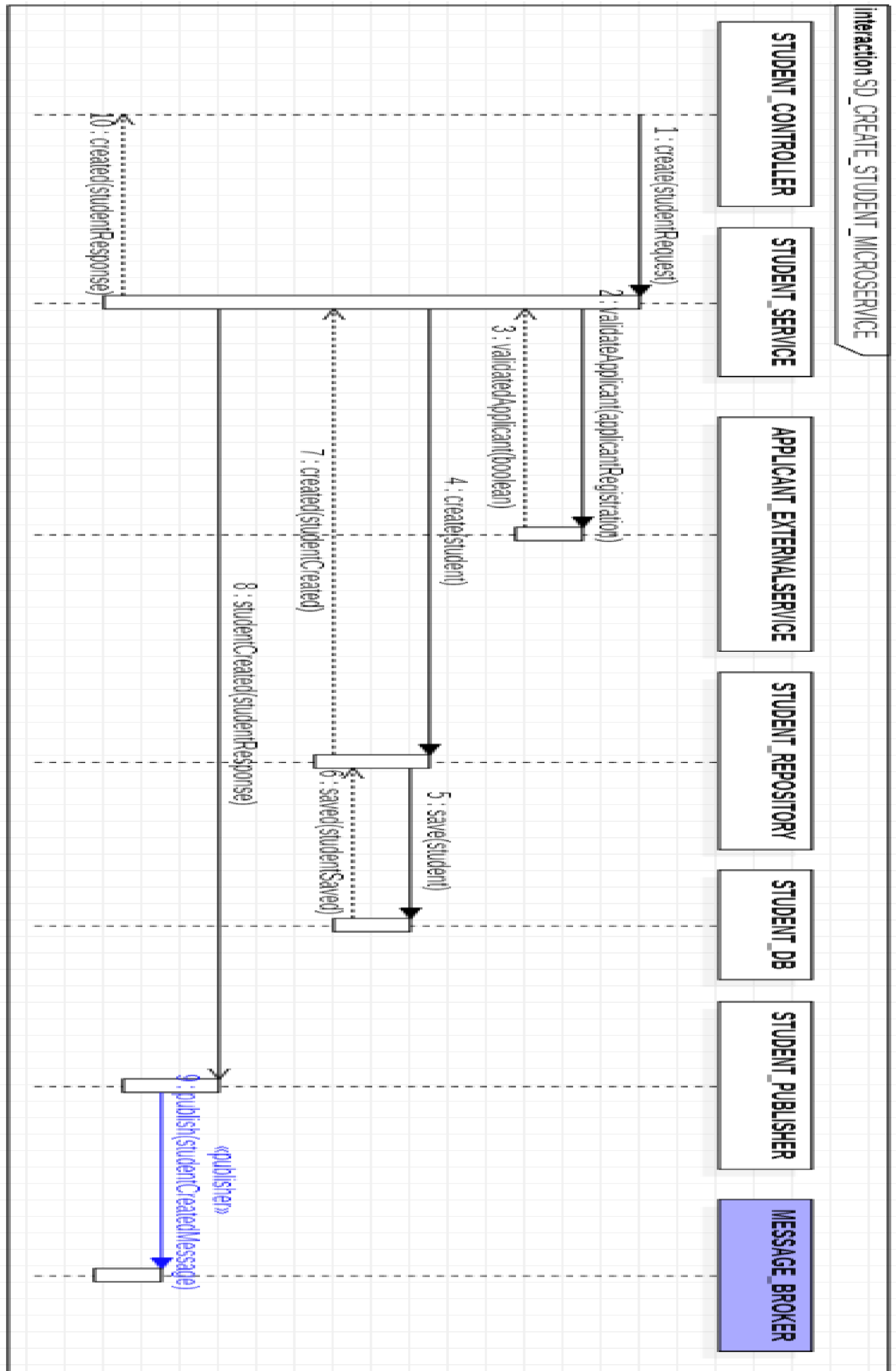

Apêndice IV - Ampliação da figura 4 - Diagrama de Classes da arquitetura monolítica


pkg
<<entity>>
CANDIDATO
candidata
<<entity>>
CANDIDATURA
disponivel
<<entity>>
CURSO
registra
<<entity>>
FATURA
registra
foi
pertence
<<entity>>
FICHA_FINANCEIRA
<<stereotype>>
PESSOA
<<entity>>
PROFESSORES
REGISTRO_ACADEMICO
possui
leciona
ALUNO_TURMA
<<entity>>
ALUNO
ALUNO_TURMA
<<entity>>
TURMA

Apêndice V - Ampliação da figura 8 - Diagrama de Sequência para criar um novo aluno em arquitetura monolítica



Apêndice VI - Ampliação da figura 9 - Diagrama de Sequência para criar um novo aluno em arquitetura de micro-serviços



Referências Bibliográficas

Alura. (2021, January 28). *MongoDB: O banco baseado em documentos*. https://www.alura.com.br/artigos/mongodb-o-banco-baseado-em-documentos, Recuperado em 10/07/2021

(Amazon, 2020) Amazon. (2020). *What are NoSQL databases?* AWS, Inc. https://aws.amazon.com/nosql/ , Recuperado em 22/05/2021

Andriel, W. A. (2017, November 30). *Microserviços com Spring Cloud: Parte 1*. Medium. https://medium.com/codeshare/microservi%C3%A7os-com-spring-cloud-parte-1-aeea162e1e33, Recuperado em 11/06/2021

(Ankur, 2021) Ankur, A. (2019a, September 26). *Spring Reactive Stream Basic concepts (Mono or Flux) (part 1)*. Medium.Com. https://medium.com/@AnkurRatra/spring-reactive-stream-basic-concepts-mono-or-flux-part-1-baed4b432977, Recuperado em 17/05/2021

Ankur, A. (2019b, September 26). *Spring reactive stream basic concepts (Mono or Flux) (part 2)*. Medium.Com. https://medium.com/@AnkurRatra/spring-reactive-stream-basic-concepts-mono-or-flux-part-2-3877cc1fda5a, Recuperado em 17/05/2021

((Apache, 2020) *Apache Foundation - Apache JMeter*. (n.d.). Apache JMeter. https://jmeter.apache.org/ Recuperado em 13/07/2021

(Baragry & Reed, 1999) Baragry, J., & Reed, K. (1998, December). Why is it so hard to define software architecture? *Asia-Pacific*. software Engineering Conference, Bundoora, Vic 3083, Australia. https://www.researchgate.net/publication/3781314 , 25/06/2021

(Bonér et al., 2014) Bonér, J., Farley, D., Kuhn, R., & Thompson, M. (2014, September 16). *The Reactive Manifesto*. The Reactive Manifesto. https://www.reactivemanifesto.org/, Recuperado em 11/05/2021

Bushnev, Y. B. (2017, May 24). *JMeter Ramp-Up - The Ultimate Guide*. Blazemeter. https://www.blazemeter.com/blog/jmeter-ramp-up-the-ultimate-guide, Recuperado em 10/06/2021

(Dijkstra, 1968) Dijkstra, E. W. (1968). The Structure of the “THE”-Multiprogramming System. *Communications of the ACM*, *II*(Number 5). https://www.eecs.ucf.edu/~eurip/papers/dijkstra-the68.pdf, Recuperado em 14/06/2021 Technological University, Eindhoven, The Netherlands

Dolphine, T. (2017). *Microservices reativos* [Slides]. QCon São Paulo. https://qconsp.com/sp2017/system/files/presentation-slides/tiago.dolphine.qcon_sp_2017_-_reactive_microservices.pdf, Recuperado em 11/06/2021

(ECPI Univercity, 2020) ECPI Univercity. (2020, November 10). *A Brief History of Cloud Computing*. ECPI Univercity. https://www.ecpi.edu/blog/a-brief-history-of-cloud-computing ,, Recuperado em 30/05/2021

(Fowler, 2005) Fowler, M. (2005, December). *Event sourcing*. Martinfowler.Com. https://martinfowler.com/eaaDev/EventSourcing.html Recuperado em 10/04/2021

(Gamma et al., 1994) Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software* (1a ed.). Addison-Wesley Professional. https://www.oreilly.com/library/view/design-patterns-elements/0201633612/, Recuperado em 06/04/2021

(Garfinkel, 2011) Garfinkel, S. (2011, October). *The cloud imperative*. MIT Technology Review. https://www.technologyreview.com/2011/10/03/190237/the-cloud-imperative/ , Recuperado em 13/06/2021

Guevara, I. G. (2019, June 26). *O que é Cloud Computing e como funciona?* Kinghost. https://king.host/blog/2019/06/o-que-e-o-cloud-computing/, Recuperado em 14/06/2021

História da computação em nuvem: como surgiu a cloud computing? (2020, May 27). IPM. https://www.ipm.com.br/administracao-geral/historia-da-computacao-em-nuvem-como-surgiu-a-cloud-computing/ Recuperado em 12/06/2021

(Imran et al., 2016) Imran, M., Alghamdi, A. A., & Ahmad, B. (2016). SOFTWARE ENGINEERING: ARCHITECTURE, DESIGN, AND FRAMEWORKS. *International Journal of*

Computer Science and Mobile Computing - IJCSMC, *5*(3), 801–815, https://www.researchgate.net/publication/311233381, Recuperado em 17/06/2021

Iundarigun. (2018, March 26). *Controle transacional*. Medium. https://medium.com/dev-cave/controle-transacional-34b9de948b8d, Recuperado em 11/06/2021

(Kubernetes.io, 2021) Kubernetes.io. (2021, May). *What is Kubernetes?* Kubernetes. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/, Recuperado em 03/007/2021

Lambda Expressions. (n.d.). Oracle. https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html, Recuperado em 11/06/2021

Lesson: Aggregate Operations. (n.d.). Oracle. https://docs.oracle.com/javase/tutorial/collections/streams/, Recuperado em 11/06/2021

Magalhães, E. (2018). *Programação reativa com spring boot, WebFlux e MongoDB: Chega de sofrer!* Medium. https://medium.com/nstech/programa%C3%A7%C3%A3o-reativa-com-spring-boot-webflux-e-mongodb-chega-de-sofrer-f92fb64517c3 Recuperado em 11/06/2021

Marcus Carvalho, O. M. C. (2016). *arquitetura de Micro Serviços: uma Comparação com Sistemas Monolíticos*. https://si.dcx.ufpb.br/wp-content/uploads/2017/08/arquitetura-de-Micro-Servic%CC%A7os-uma-Comparac%CC%A7a%CC%83o-com-Sistemas-Monoli%CC%81ticos.pdf, Recuperado em 09/06/2021

(McIlroy et al., 1978) McIlroy, M. D., Pinson, E. N., & Tague, B. A. (1978). Unix time-sharing system: Foreword. *The Bell System Technical Journal*, *57*, 1899–1904. https://archive.org/details/bstj57-6-1899/mode/2up, Recuperado em 10/04/2021

(Microsoft, 2020) Microsoft. (2020). *O que é a computação na cloud? Manual para principiantes | Microsoft.* Microsoft Azure. https://azure.microsoft.com/pt-pt/overview/what-is-cloud-computing/, Recuperado em 10/04/2021

Mishra, P. (2021, May 24). *Streams in Computer Programming: Definition & Examples*. Study.Com. https://study.com/academy/lesson/streams-in-computer-programming-definition-examples.html, Recuperado em 22/06/2021

(MongoDB, 2021) MongoDB. (2021). *What is mongodb*. https://www.mongodb.com/, Recuperado em 10/06/2021

(NSTech, 2019) NSTech. (2019). *Lições após o primeiro ano reativo!* Medium. https://medium.com/nstech/li%C3%A7%C3%B5es-ap%C3%B3s-o-primeiro-ano-reativo-786ab7785227 , Recuperado em 11/06/2021

O que é a computação em nuvem? (n.d.). AWS Amazon. https://aws.amazon.com/pt/what-is-cloud-computing/, Recuperado em 03/06/2021

O que é a computação na cloud? (n.d.). Azure Microsoft. https://azure.microsoft.com/pt-pt/overview/what-is-cloud-computing/#benefits, Recuperado em 03/06/2021

O que é cloud computing. (n.d.). Mandic. https://www.mandic.com.br/cloud/, Recuperado em 14/06/2021

O que é e como funciona o cloud computing? (n.d.). CentralServer. https://blog.centralserver.com.br/o-que-e-e-como-funciona-o-cloud-computing/, Recuperado em 13/06/2021

(Oracle, 2020) Oracle Inc. (2020). *What is a relational database*. Oracle. https://www.oracle.com/pt/database/what-is-a-relational-database/, Recuperado em 10/04/2021

(Parnas, 1979) Parnas, D. L. (1979). Designing software for ease of extension and contraction. *IEEE Transactions on software Engineering*. Published. https://www.researchgate.net/publication/3189208, Recuperado em 11/06/2021

(Perry & Wolf, 1992) Perry, D. E., & Wolf, A. L. (1992). Foundations for the Study of software Architecture. *ACM SIGSOFT - SOFTWARE ENGINEERING NOTES*, *17*, 40–52. https://users.ece.utexas.edu/~perry/work/papers/swa-sen.pdf , Recuperado em 11/06/2021

(Priberam, 03/04/2021) Priberam. *https://dicionario.priberam.org/resiliencia* , Recuperado em 03/04/2021

Produtos e serviços. (n.d.). Google Cloud. https://cloud.google.com/products?hl=pt, Recuperado em 11/06/2021

Project Reactor. (n.d.). Project Reactor. https://projectreactor.io/, Recuperado em 11/06/2021

(Richards, 2015) Richards, M. (2015). *software architecture patterns* [E-book]. O'Reilly Media, Inc.

Ronaldo. (2015). *Conheça o Spring Transactional Annotations*. Devmedia. https://www.devmedia.com.br/conheca-o-spring-transactional-annotations/32472,, Recuperado em 08/06/2021

(Shaw, 1989) Shaw, M. (1989). Large scale systems require Higher-Level abstractions. *ACM SIGSOFT software Engineering Notes*, *14*(3), 143–146., https://ur.booksc.eu/book/44933458/af4a45, Recuperado em 18/05/2021

(Spooner, 1971) Spooner, C. R. (1971). A software architecture for the 70's: Part I - the general approach. *software - Practice and Experience*, *1*, 5–37. https://ur.booksc.eu/ireader/11923097, Recuperado em 18/05/2021

(Spring, 2020) Spring. (2020). *Why Spring? Spring.* https://spring.io/why-spring/, Recuperado em 10/04/2021

Spring. (2020). *Spring Cloud Gateway*. Spring. https://cloud.spring.io/spring-cloud-gateway/reference/html/, Recuperado em 10/04/2021

What is kubernetes. (2021). Kubernetes. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/, Recuperado em 13/07/2021

Glossário

ARPANET - Advanced Research Projects Agency Network. projeto do Departamento de Defesa dos EUA.

API – Application Programming Interface.

CRUD - Create, Read, Update and Delete.

CSV - Comma-Separated Values.

DLL - Dynamic Link Library.

GKE - Google Kubernetes Engine.

GPL - General Public License, que pode ser consultada através do website http://www.gnu.org/licenses/license-list.html.

HTTP – Hyper Text Transfer Protocol, protocolo utilizado para comunicação na internet.

HTTPS – Hyper Text Transfer Protocol Security, protocolo utilizado para comunicação na internet que utiliza certificados digitais para segurança.

REQUEST – Requisição de um pedido, muito utilizado sobre métodos HTTP.

RESPONSE - Resposta de uma Request, muito utilizado sobre métodos HTTP.

REST – Representational State Transfer, tipo de comunicação utilizada entre sistemas.