Microsserviços na Nuvem : potencializando serviços em produção

Microservices in the Cloud: potentiating services in Production

Rafael Marcos Victoriano Damasceno *
Sergio Roberto Delfino †

Resumo

Este artigo tem como objetivo demonstrar como sistemas podem ser potencializados utilizando uma arquitetura de microsserviços na nuvem, resolvendo problemas de escalabilidade, monitoramento, segurança e comunicação entre serviços, utilizando de Java com *Quakus e Spring Boot*, e *Node JS* para os microsserviços, *Docker* para contêineres e *AWS* como provedor de nuvem para utilizarmos serviços como *SQS, Api Gateway, Load balancer etc.* Serão criados quatro microsserviços, login, controle de alunos, controle de cursos e um para disparo de notificações como um envio de e-mail caso um aluno se cadastre na plataforma, para realizar as configurações e solucionar os problemas descritos.

Palavras-chave: Microsserviços, Load Balacing, Escalabilidade.

Abstract

This article aims to demonstrate how systems can be leveraged using a microservices architecture in the cloud, solving scalability, monitoring and communication between services, using Java with Spring Boot and Node JS for microservices, Docker for containers and AWS as cloud provider for us to use services like SQS, Api Gateway, Load balancer etc. Four microservices will be created, login, student control, course control and one to trigger notifications such as sending an email if a student registers on the platform, to configure and solve the problems described.

Keywords: Microsservices, Load Balacing, scalability.

*"analista de sistema na empresa Rede Itaú, graduando em Análise e Desenvolvimento de Sistemas pela Faculdade de Tecnologia de Ourinhos, rafaelvictorian05@hotmail.com."

†"mestre em Ciência da Computação pelo Centro Universitário Eurípides de Marília (UNIVEM) e docente na Faculdade de Tecnologia de Ourinhos, sergio.delfino@fatecourinhos.edu.br."

1 Introdução

Segundo Muller, Meinhardt e Mendizabal (2020), a arquitetura de microsserviços está sendo cada vez mais adotada no desenvolvimento de novos sistemas pelo fato de proporcionar alta escalabilidade e facilidades no desenvolvimento.

Para compreender melhor como funciona uma arquitetura de microsserviços precisamos entender a arquitetura monolítica. Segundo Souza e Pelissari (2017) uma arquitetura monolítica geralmente é composta por um unico software central, essencialmente por três partes: uma interface de interação com o usuário, uma base de dados e uma aplicação servidor.

O modelo arquitetural de microsserviços é o oposto, e tem como objetivo desmembrar serviços em pequenos serviços independentes, com suas próprias particularidades como um banco de dados para cada serviço. AWS (2021) define que cada serviço de um microsserviço pode ser desenvolvido independente de um outro serviço sem afetar o sistema no todo e se o serviço de um sistema estiver indisponível ele não afetará os outros sistema conectados, somente a tarefa que está designado a fazer. Alguns desses benefícios são: pequenos sistemas especializados, agilidade, escalabilidade flexível, liberdade tecnológica, reutilização de código e serviços e resiliência.

Quando utilizamos o modelo arquitetural de microsserviços, consequentemente temos mais complexidade e precisamos pensar em como vamos genrenciar contêiners, como as aplicações vão se comunicar, como monitorá-las e centraliza-las de forma segura. Estes são alguns dos pontos que precisamos solucionar quando considerarmos usar este modelo arquitetural.

Dessa forma, esse projeto tem como objetivo solucionar esses problemas utilizando um modelo arquitetural de microsserviços sendo potencializado pelo ambiente em nuvem utilizando o provedor *AWS* para resolver os problemas de comunicação, escalabilidade, monitoramento de serviço, segurança e implementar *Auto Sacaling e Load Balacing*.

2 Revisão Bibliográfica

Este capítulo contém a revisão bibliográfica relacionada ao assunto que será abordado neste artigo.

2.1 Arquitetura de Microsserviços

Segundo Monte et al. (2020) microsserviços é um padrão arquitetural complexo onde serviços pequenos trabalham em conjunto para se tornar uma unica aplicação.Os serviços que compõe uma arquitetura voltada a micrsserviços, podem ser escalaveis de forma independente e permitem a diversisade de tecnologias e linguagens utilzadas para cada serviço.

Fowler e Lewis (2014) relatam que comunicação entre esses serviços ocorre de forma leve, através do protcolo HTTP como Rest APIs, mensageria como SQS e RabbitMq, eventos etc.

Aplicações que utilizam-se desta arquitetura são altamente escalaveis que segundo Santos (2020) a escalabilidade permite a adaptação de componentes individuais ou de toda a aplicação à um balancemento de carga dos serviços. Como esses serviços agem de forma

particular e desaclopada, a maioria dos recursos do sistema pode ser dimensionada para atender às suas respectivas tarefas.

O modelo de arquitetura de microsserviços define como um dos principais conceitos a gestão de descentralização dos dados, que segundo Fowler e Lewis (2014) cada serviço deve gerar sua propria base dados, mesmo que esses serviços utilizem instancia de tecnologias iguais. Mas é possível que cada serviço escolha qual opção mais adequado para armazenar os dados entre banco relacionais e não relacionais.

Este modelo arquitetural tem como principal desvantagem a sua complexidade em geral, pelo fato de que as equipes precisam ter muito cuidado no desenvolvimento e na intregação entre os serviços. Mas segue sendo o modelo mais adotado entre as empresas atualmente.

2.2 Computação em nuvem

Azure (2022) define computação em nuvem como o fornecimento de serviços de computação, que oferecem inovações imediatas, recursos flexíveis e escalaveis e economias. A computação na nuvem ajuda a reduzir os custos operacionais, e torna a infraestrutura mais fexivel, pelo fato de que o cliente não precisara mais se procucupar com hardware e poderá acessar sua infraestrutura de qualquer local.

Segundo Sousa, Moreira e Machado (2009) o modelo de computação em nuvem tem como princpal objetivo ofereçer serviços com alta disponilibilidade, baixo custo e escalabilidade.

Para Rosa (2016) temos três modelos de serviço na computação em nuvems.

Software as a Service (SaaS) este modelo concetra no do na inovação e no desenvolvimento sfotware pelo fato de que o usario não administra ou controla a infraestutura subjacente. O modelo de SaaS proporciona sistemas de software com propósitos específicos, que estão disponíveis para o usuário através da internet.

Plataform as a Service (PaaS) o PaaS oferece uma infraestrutura de alto nível de integração para implementar e testar aplicações na nuvem. Como o modelo SaaS O usuário não administra ou controla a infraestrutura subjacente. Esse modelo auxilia na implatação do sftware oferecendo sistemas operacionais e linguagens de programação.

Infrastructure as a Service (IaaS) a Infraestrutura como serviço é a parte responsável por prover toda infraestrutura para o PaaS e SaaS, com o objetivo principal de tornar mais acessível o fornecimento de recursos de computação fundamentais para se construir um ambiente sob demanda.

Na computação nuvem temos três modelos de implantação nuvem privada que seria excluisiva de uma organização, nuvem publica que é disponibilizada para qualquer usuario, nuvem comunidade quando há um compartilhamento de diversas empresas e nuvem hibrida que é a composição de duas ous mais nuvens com modelos de implantação diferentes. Sousa, Moreira e Machado (2009) salienta que os principais benefícios deste modelo são, reduzir o custo de toda construção da infraestrutura de uma empresa, flexibilidade na adição, substituição de novos recursos computacionais e ter uma abstração e facilidade de acesso de clientes e empresas aos serviços.

Atualmente computação em nuvem é uma abordagem muito adotada por empresas, grandes empresas como Microsoft, Google e Amazon possuem seus proprios ambiemntes em nuvem que frequentemente são os ambientes adotados por empresas.

2.2.1 *Amazon Web Services* (AWS)

A *Amazon Web Services* (AWS) é a plataforma de nuvem mais adotada e mais abrangente do mundo, oferece mais de 200 serviços tendo *datacenters* em todo mundo. Este provedor em nuvem tem milhões de cliente no mundo, desde startup há empresas consolidadas (AWS, 2022b).

2.2.1.1 AWS ECS

O *Amazon Elastic Container Service* é um serviços da AWS para orquestração de contêiners, que ajuda a implantar, gerenciar e escalar facilmente aplicações em contêiners. Ele gerencia o cliclo de vida do contêiner (AWS, 2020).

Neste serviço é possível habilitar o *Auto Scaling* (Auto escalabilidade) que é a capacidade de diminuir ou aumentar as *tasks definitions* (recurso que encapsula configurações de monitaramento, contêiners etc) de uma aplicação. O Amazon ECS publica métricas no *CloudWatch* (recurso de monitoramento da AWS) pelo serviço, da CPU e da memória, no CloudWatch também temos os *logs* da aplicação. Com essas informações do *CloudWatch* podemos configurar o *Auto Scaling* para aumentar as *tasks definitoins* da aplicação escalando de forma horizontal, é possível configurar o *Auto Scaling* de forma programada de acordo com a hora de pico da aplicação (AWS, 2016).

Zero Down Time é uma estratégia utilzada para que quando ocorra um deploy de uma nova versão a aplicação continue fucionando. O ECS coloca essa estratégia em prática através do *Load Balancing* (Balanceador de carga) onde ele balanceia a carga para o contêiner antigo da aplicação enquanto ele sobe o outro contêiner, após a conclusão do deploy ele deleta o contêiner antigo.

2.2.1.2 AWS Lambda

O AWS Lambda permite que você execute códigos sem provisionar e gerenciar servidores, o código só é executado quando necessário, permitindo diversos gatilhos para executa-lo. O serviço é pago apenas pelo tempo de computacão¸ que se é consumido, não há combrança para código em execução. Com o *AWS Lambda*, permite que a execução do código em diversas linguagens de programção, sendo administrada totalmente pelo própria AWS (PEREIRA; BACK; JÚNIOR, 2019).

2.3 Arquitetura Monolitica

Segundo Camelo (2020) a arquitetura monólitica é um padrão arquitetural onde se tem apenas uma aplicação, grande e completa. Uma única aplicação trata de todas as funcionalidades de um sistema. Está aplicação única é responsável por todas as camadas da solução desde o cliiente ao servidor.

Esta arquitetura acabou perdendo espaço por ser pouco escalavel e ter difícil manutenlabilidade. Existem muitos sistemas legados com esta arquitetura. Mas há empresas que defendem essa arquitetura relatando que tudo começa com monólito. Um caso de sucesso que utiliza esta arquitetura é o stack overflow.

2.4 Escalabilidade

Camelo (2020) define escalabilidade como a capacidade de acompanhar o crescimento do serviço por demanda. Isto ocorre quando o serviço cresce e começa receber muitas requisições, assim o serviço aumenta sua capacidade de uso acomodadno grande quantidades de dados.

2.4.1 Escalabilidade Vertical

A escalabilidade vertical ocorre quando um maquina não aguenta mais o volume de requisições e é aumentado a capacidade de memoria e CPUs desta maquína, este modelo é mais usado para arquiteturas monóliticas.

Felix (2020) ressalta que esta estratégia é rapída, mas que facilmente podemos chegar o limite da maquína.

2.4.2 Escalabilidade Horizontal

A escalabilidade horizontal é mais utilizada para arquitetura orientada microsserviços. Segundo Felix (2020) há medida que o servidor vai se sobrecarregando é adicionando servidores mais simples em vez de ter apenas uma máquina poderosa.

Este modelo é mais eficaz, mas mais complexo pois é preciso de um gerencionamento nos serviidores como balanceadores de carga.

2.5 Trabalhos Correlatos

Segundo o projeto abordado, a alguns exemplos de empresas que utilizam a arquitetura de microsserviços.

2.5.1 *Netflix*

Netflix é um aplicativo implantado no ambiente da nuvem, que contém um serviço de streaming de filmes, séries e documentários que são cobrado por mês uma taxa, para serem liberadas em cada cadastro de seus clientes. Em 2009 a Netflix resolveu mudar do sistema monolítico para microsserviços pelo fato da demanda de serviços cresceram grandemente (LTS, 2021). A streaming vem crescendo muito no Brasil, devido ao método de transmissão de dados de vídeo e áudio pela internet em tempo real, sem precisar baixar nada, assim fazendo com que o conteúdo sejam armazenados temporariamente na máquina e enviadas ao usuário quase que instantaneamente (COUTINHO, 2014).

2.5.2 *Amazon Prime*

Amazon Prime chegou no Brasil em 2019, a Amazon é um serviço de assinatura que oferece vários benefícios. Utilizando também a plataforma de streaming que disponibilizam serviços em seus aplicativos Amazon Prime Video e *Amazon Music*. Utilizam uma arquitetura de microsserviços implantada no seu provedor de nuvem. Possui também sistema de varejo, com muitas ofertas, descontos em produtos e streaming de jogos, séries, filmes, músicas, livros e muito mais que utilizam da mesma arquitetura (SANTOS, 2021).

2.5.3 *Spotify*

Spotify é um serviço digital de músicas, podcasts, vídeos e outros conteúdos online no mundo todo. Está disponível para muitos dispositivos, como celulares, computadores, TVs, carros e muito mais, podendo também trocar de um dispositivo para outro facilmente com a funcionalidade do aplicativo *Spotify Connet* (SPOTIFY, 2021). Durante a conferência GOTO Berlin 2015, o vice presidente Kevin Goldsmith de engenharia do aplicativo, falou sobre o uso de microsserviços e suas importâncias na descentralização da arquitetura da companhia e muito úteis em aplicações monolíticas (LINDERS, 2016). Os fundadores do *Spotify* construíram um sistema com componentes escalonáveis independentes para tornar o aplicativo com a sincronização mais fácil. A principal capacidade de prevenir falhas e usando os benefícios utilizados dos microsserviços e o ambiente da nuvem, fazendo com que mesmo que falharem os serviços simultaneamente, os demais usuários não serão afetados (LTS, 2021).

2.5.4 *Uber*

Uber é uma ferramenta disponíveis nos sistemas IOS e Android, podendo ser baixados pelo *App Store, Google Play* ou no próprio site oficial da *Uber*. Este serviço conecta usuários motoristas solicitados particularmente, para um meio de transporte econômico. Em 2009 o aplicativo foi aperfeiçoado por Garrett Camp e Travis Kalanick, quando sentiram dificuldade para pegarem um táxi que o levassem para uma conferência na França. Para a solução do problema em mente, colocaram em prática na cidade de São Francisco em 2010, que fica localizada nos EUA (UBER, 2018). No entanto, pelo fato dá demanda do serviço ter aumentado significativamente, os desenvolvedores decidiram mudar da arquitetura monolítica para microsserviços, assim podendo usar variedade de linguagens de estruturas. Contando hoje em dia, com mais de 1.300 (mil e trezentos) microsserviços focados em melhorar a escalabilidade do app (LTS, 2021). Chegando no Brasil em 2014, a *Uber* vem trabalhando constantemente ampliando suas operações e fornecendo um meio de transporte confiável e eficiente (UBER, 2018). Uber também utiiliza o ambiente em nuvem para suas aplicações.

3 Materiais e Métodos

Este tópico descreve quais materiais e procedimentos para conclusão do trabalho.

3.1 Ferramentas e Tecnologias

Atualmente existem diversas tecnologias que ajudam no desenvolvimento de software. Listaremos algumas das ferramentas e tecnologias mais utilizados para sofwares com arquitetura orientada a microsserviços:

3.1.1 Intellij IDEA

IntelliJ IDEA é um ambiente de desenvolvimento integrado para o desenvolvimento de software em Java. Desenvolvido pela JetBrains. Esta ferramento contém duas versões *Community* (versão gratuita) e Ultimate (versão paga) (SOUZA, 2022).

3.1.2 Insomnia

Segundo Ribeiro (2020) o insomnia é uma ferramenta de testes de rotas de API com uma interface intuitiva e diversas funcionalidades úteis, como a capacidade de gerenciar ambientes e variáveis, testar diferentes tipos de requisições e gerar documentação automaticamente.

3.1.3 Java

Java é uma linguagem de programação orientada a objetos. Java foi criado em 1995 pela empresa Sun Microsystem, que em 2008, foi adquirido pela Oracle. A linguagem java funciona em qualquer plataforma por conta da Maquina Virtual Java (JVM), pelo fato de que quando um progrma java é compilado é transformado em bytes e logo depois interpretado pela JVM para executa-lo em qualquer plataforma (AUGUSTO, 2021);

3.1.4 Node Js

Node JS é uma plataforma altamente escalável e de baixo nível sendo possível programar em diversos protocolos de rede e internet. Node Js foi criado em 2009 por Ryan Dahl. Node JS é(PEREIRA, 2014)

3.1.5 Amazon DynamoDB

O Amazon DynamoDB é um banco de dados de chave-valor NoSQL, sem servidor e totalmente gerenciado, projetado para executar aplicações de alta performance em qualquer escala. O DynamoDB oferece segurança integrada, backups contínuos, replicação multirregional automatizada, armazenamento em cache na memória e ferramentas de importação e exportação de dados. (AWS, 2022a).

3.1.6 Git

Criada por Linus Torvalds. GIT é uma tecnologia de versionamento de codigo distribuido que auxlia no desenvolvimento de software, sendo a tecnologia de versionamento de codigo mais adotadas atualmente. Segundo Roveda (2021) este sistema de versionamento pode resgitrar quaisquer alterações feitas no código de acordo com que o usuario quiser, armazenando essas informações, ou seja cada novo registro é criado uma nova versão do serviço que está sendo desenvolvido e caso seja necessario o desenvolvedor pode regredir de versão. Essa tecnologia ainda permite que varios desenvolvedores tenham acesso ao repositorio de codigo do sistema.

3.1.7 Spring Framework

Geekhunter (2020) define Spring Framework como uma tecnolgia desenvolvido para a plataforma Java baseada em pdrões de projeto, inversão de controle e injeção de dependência. Esta tecnologia tem um ecossistema constituído por diversos e completos módulos capazes de dar um boost na aplicação Java.

O Spring vem com intuito de preparar um ambiente de infraestrtura todo configurado, para que o desenvolvedor foque somente na logíca da aplicação. Esta tecnologia conta com muitos componentes para ajudar um desenvolvedor no desenvolvimento como, Spring Data para integrações com banco de dados, Spring Security para segurança da aplicação

desde autenticação à a autorização, Spring Cloud para integrações com a computação em nuvem, etc.

3.1.8 Quarkus

Segundo RedHat (2023) O Quarkus é um framework Java nativo do Kubernetes que foi projetado para funcionar com padrões, estruturas e bibliotecas Java conhecidas. Ele é otimizado especificamente para containers e é eficaz para ambientes serverless, de nuvem e Kubernetes.

O Quarkus vem com inutuito de ser eficiente para ambientes de containers, permitindo o desenvolvimento rápido e eficaz de aplicações Java escaláveis e distribuídas em ambientes de nuvem e Kubernetes, tornando-se uma das opções ideiais quando se trata de aplicações *cloud native*.

3.1.9 Terraform

Terraform é uma infraestrutura como ferramenta de código que permite definir recursos em provedores de nuvem em arquivos de configuração legíveis em humanos sendo possível criar, alterar e reutilizar código em diversos ambientes (TERRAFORM, 2020).

3.1.10 Docker

Segndo IBM (2021) docker é uma plataforma de conteinerização. Ela permite que aplicações sejam empocatodas em containers, onde se é configurado bibliotecas, dependências e variveis de ambiente da aplicação para que ala possa ser executada em qualuqer sistema operacional. Esses contêineres facilitam na entrega de sistemas distribuidos pelo fato de que esta tecnologia é de facil integraçaõ com ambientes de cloud.

3.1.11 Procedimentos

A primeira fase, foi baseada em pesquisas teóricas para adequação do artigo e projeto, visando uma definição simples para uma plataforma onde alunos podem se inscrever em cursos, e as melhores práticas arquiteturais de microsserviços que realizam processos assíncronos. O sistema é simples, mas tem uma arquitetura complexa, onde tem como objetivo mostrar a potencialização de microsserviços na nuvem (AWS), disponibilizando soluções para resolver problemas complexos.

A segunda fase, foi baseada em construir uma arquitetura para termos o melhor desempenho dos microsserviços e utilizar serviços da AWS que nos dão o poder de ter maior escalabilidade e resiliência então optamos por AWS lambda para ações simples (login, disparo de e-mails), ECS para realizar orquestrações de containers, DynamoDB para base dados e SNS e SQS para trabalharmoos com eventos de modo assíncrono sem perder a resiliência. A figura 1 um mostra a arquitetura final do sistema.

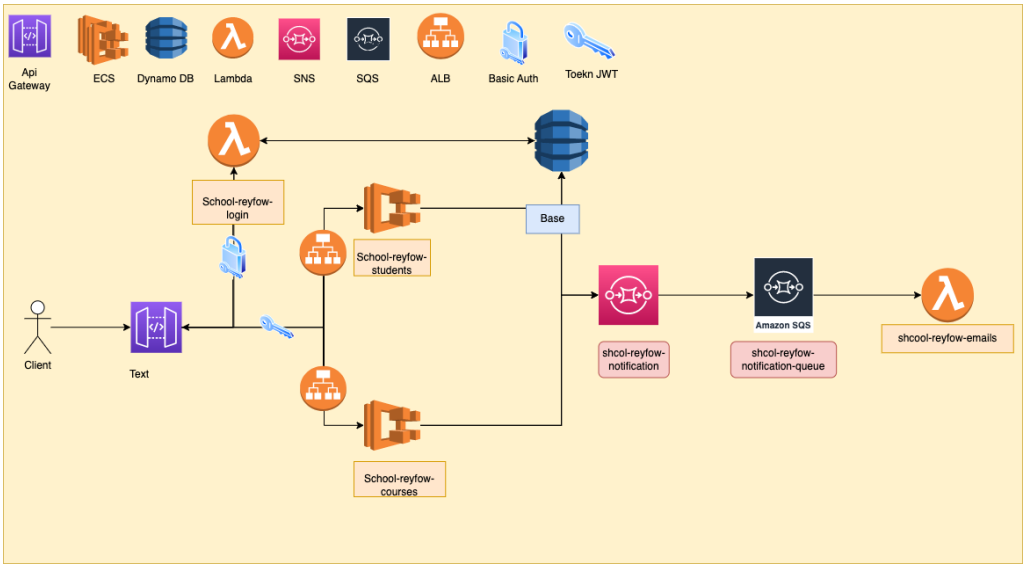

Figura 1 – Arquitetura orientada a microsserviços

Escalabilidade é resolvida de duas formas, a primeira forma é com AWS lambda onde sempre que houver uma chamada para o endpoint no *Api gateway* esse lambda irá ser acionado e instanciado em um servidor e morrerá logo em seguida, então se houver 500 chamadas em um minuto ou mais o serviço AWS Lambda atenderá as 500 chamadas de forma tranquila, por esse motivo é desejável que as AWS lambdas tenham tempo de startup curto e um tempo de execução rápido. A segunda forma é utilizar o *Auto Scaling* nos ECS onde será configurado para que quando nossa aplicação chegar a um consumo de memória maior que 35 ele irá provisionar outro container para nossa aplicação. As duas formas são para soluções de escalabilidade horizontal. As figuras 2 e 3, mostram configuração do *Auto Scaling* para aplicações do ECS e a configuração de infraestrutura de um AWS Lambda utilizando o framework serverless respectivamente.

```
#Auto scaling policy
resource "aws_appautoscaling_policy" "policy" {
  name               = "scale-out-policy"
  policy_type        = "StepScaling"
  resource_id        = aws_appautoscaling_target.target_school_courses.resource_id
  scalable_dimension = aws_appautoscaling_target.target_school_courses.scalable_dimension
  service_namespace  = aws_appautoscaling_target.target_school_courses.service_namespace

  target_tracking_scaling_policy_configuration {
    predefined_metric_specification {
      predefined_metric_type = "ECSServiceAverageMemoryUtilization"
    }

    target_value = 35.0

  }
}
```

Figura 2 – Configurando Auto Scaling

! serverless.yml ×

! serverless.yml

```yaml
service: school-reyfow-login
frameworkVersion: '3'

provider:
  name: aws
  runtime: nodejs16.x

functions:
  login:
    handler: app.lambdaHandler
plugins:
  - serverless-plugin-typescript
custom:
  serverlessPluginTypescript:
    tsConfigFileLocation: './tsconfig.build.json'

```

Figura 3 – Configurando infra AWS Lambda com framework serverless

Load Balacing é resolvido com *Application Load Balancer* é um único ponto de acesso para o cliente, que irá distribuir o tráfego para as tasks do ECS service, se recebermos 100 requisições elas serão divididas entre as tasks, evitando sobrecarregar apenas uma. A figura 4 mostra como adicionamos o Load Balancer a nosso ECS via terraform.

```
load_balancer {
  target_group_arn = var.tg-arn
  container_name   = "School-reyfow-courses"
  container_port = 8080
}
```

Figura 4 – Adicionando Load Balancer ao ECS

Comunicação é resolvida com AWS Api Gateway que disponibiliza endpoints para acesso do lambda e para o load balancer do ECS essa comunicação é feita de forma síncrona utilizando arquitetura REST. As comunicações internas dos serviços serão feitas de forma assíncrona em um modelo Pub/Sub onde publicaremos uma notificação ao SNS e ele distribuíra para os inscritos que nesta arquitetura é o SQS. A figura 5 e 6 mostram como criamos um endpoint no api gateway via terraform e como publicamos uma notificação em um tópico do SNS utilizando Java respectivamente.

```
#PATH GERAL
resource "aws_api_gateway_resource" "student_resource" {
  rest_api_id = aws_api_gateway_rest_api.school-reyfow.id
  parent_id   = aws_api_gateway_resource.geral_resource.id
  path_part   = "student"
}


resource "aws_api_gateway_method" "student_method" {
  rest_api_id   = aws_api_gateway_rest_api.school-reyfow.id
  resource_id   = aws_api_gateway_resource.student_resource.id
  http_method   = "POST"
  authorization = "NONE"

   request_parameters = {
    "method.request.header.Authorization" = true
  }
}


# Definição da integração do HTTP Proxy
resource "aws_api_gateway_integration" "student_integration" {
  rest_api_id             = aws_api_gateway_rest_api.school-reyfow.id
  resource_id             = aws_api_gateway_resource.student_resource.id
  http_method             = aws_api_gateway_method.student_method.http_method
  integration_http_method = "POST"
  type                    = "HTTP_PROXY"
  uri                     = "${var.alb-url}/student"

  request_parameters = {
    "integration.request.header.Authorization" = "method.request.header.Authorization"
  }
}
```

Figura 5 – Criando um enpoint no AWS Api gateway utilizando terraform

```
@Slf4j
@RequiredArgsConstructor
@Service
public class PublishTopic {

    private final NotificationMessagingTemplate notificationMessagingTemplate;

    @Value("${school-reyfow.topic-name}")
    private String topicName;

    1 usage  RafaelVictoriano
    public void pubTopic(Object event, Map<String, Object> headers) {
        this.notificationMessagingTemplate.convertAndSend(topicName, event, headers);
        log.info("Send message to topic, topicName:{}", topicName);
    }

}
```

Figura 6 – Publicando notificação no SNS utilizando Java

Monitoramento, será resolvido com o cloudwatch da AWS, onde é possível visualizar logs, latência e a saúde da aplicação.

Por fim, segurança é resolvido usando uma API de login onde será possível se autenticar com login e senha, se a autenticação estiver correta a API irá retornar um token JWT onde será possível acessar as outras APIs do do sistema.

A terceira fase foi provisionar os recursos ECS, AWS Api Gateway, Lambda, SQS, SNS e DynamoDB na AWS utilizando terraform e o framework serverless. A figura 7 e 8 mostram a criação de uma tabela no DynamoDB com terraform e configuração de um ECS *service*.

```
resource "aws_dynamodb_table" "student-table" {
  name           = "student"
  hash_key       = "id"
  read_capacity  = 5
  write_capacity = 5

  attribute {
    name = "id"
    type = "S"
  }

  tags = {
    Environment = "TCC"
  }
}
```

Figura 7 – Criando tabelas no DynamoDB com terraform

```
#ecs service
resource "aws_ecs_service" "school-reyfow-students-service" {
  name            = "school-reyfow-students-service"
  cluster         = aws_ecs_cluster.school_reyfow.id
  task_definition  = aws_ecs_task_definition.School-reyfow-students-task.arn
  desired_count    = 1
  launch_type      = "FARGATE"
  platform_version = "LATEST"

  network_configuration {
    subnets          = var.subnet_ids
    security_groups  = [aws_security_group.school-reyfow-students-security_group.id]
    assign_public_ip = true
  }

  load_balancer {
    target_group_arn = var.tg-arn
    container_name   = "School-reyfow-students"
    #elb_name         = "school-reyfow-alb"
    container_port = 5000
  }

}
```

Figura 8 – ECS service com terraform

A quarta fase foi desenvolver e configurar as aplicações que vão estar em uma

arquitetura AWS Lambda que são o serviço de login e serviço de e-mail, desenvolvidos com Node JS e Java com Spring Native respectivamente.

A quinta fase foi realizado o desenvolvimento e as configurações das aplicações que ficaram no ECS e a geração das imagens Docker dos serviços de controle de alunos e cursos que foram desenvolvidos utilizando Java com Spring Boot e Java com Quarkus respectivamente.

4 Resultados e Discussões

O objetivo dos resultados e discussões apresentados neste capítulo é mostrar como o uma arquitetura orientada a microsserviços pode ser pontencializada em um ambiente em nuvem e solucionando problemas relacionados a esta arquitetura.

4.1 Resultado segurança

O objetivo do teste foi inserir senhas e logins inválidos ao acessar a API de login, a fim de que a API bloqueasse o acesso, retornando o status 401, o que significa que a requisição para efetuar o login foi negada. Posteriormente, um outro teste foi realizado com credenciais válidas para que pudéssemos receber um *token* JWT, que nos permitiria acessar as APIs do sistema.

A figura 9 mostra a API de login bloqueando acesso.

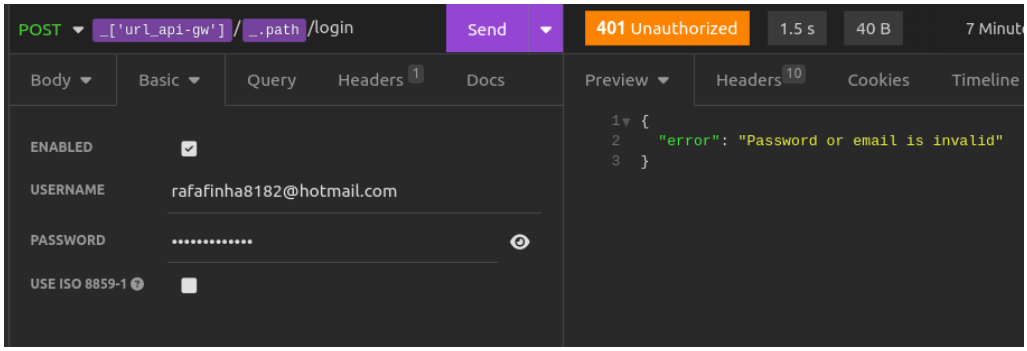

Figura 9 – Bloqueando acesso com credenciais inválidas

A figura 10 mostra a API de login gerando um token JWT, pelo fato das credenciais serem válidas.

A figura 11 mostra o acesso a API de consulta de cursos utilizando o *token* gerado pela API de login gerando um token JWT para se autenticar.

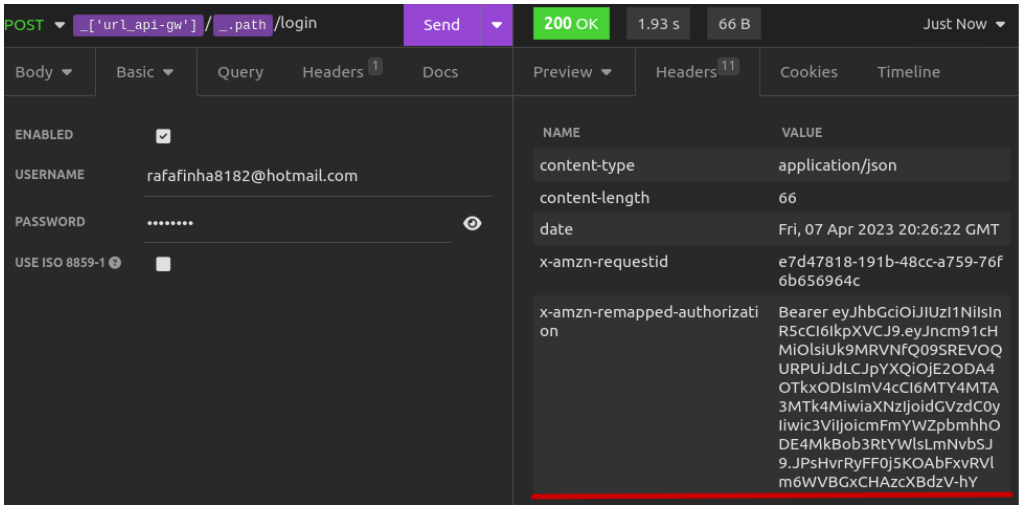

Figura 10 – Gerando um token JWT

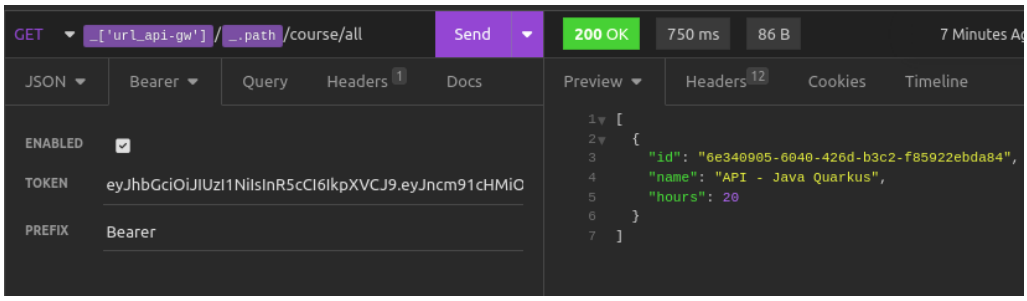

Figura 11 – Consulta de cursos

4.2 Resultado escalabilidade

O objetivo deste tópico é mostrar como a escalabilidade foi resolvida com o *Auto Scaling*, ECS e AWS Lambda.

4.2.1 Resultado escalabilidade AWS Lambda

O objetivo do teste foi realizar chamadas através do *AWS API Gateway* e enviar notificações para o SQS para que ambos acionassem nossos AWS Lambdas, para que eles pudessem escalar.

A figura 12 mostra o número de invocações do AWS Lambda para a API de login, o que significa que a AWS escalonou os recursos necessários para atender a demanda das requisições, sem que precisássemos nos preocupar com a infraestrutura do servidor, métricas coletadas pelo *CloudWatch*. Este lambda é invocado a partir de um enpoint do *AWS API Gateway*.

A figura 13 mostra o número de invocações do AWS Lambda que dispara notificações por email, com as métricas coletadas pelo *CloudWatch*. Este lambda é invocado a partir de um SQS, quando há novas mensagens no SQS, o AWS Lambda é invocado.

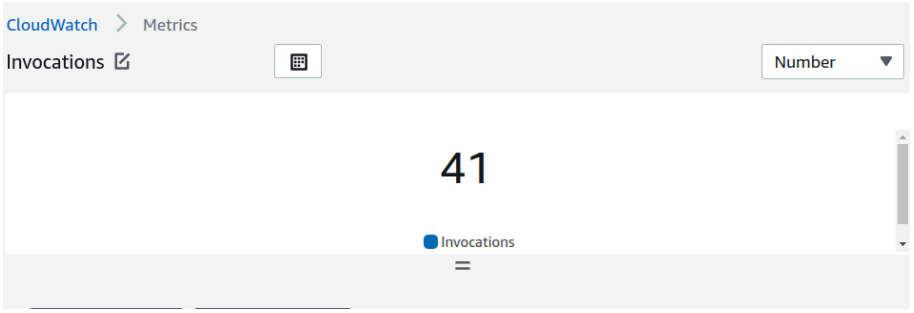

Figura 12 – Paínel do cloudwatch, número de invocações do AWS Lambda de login

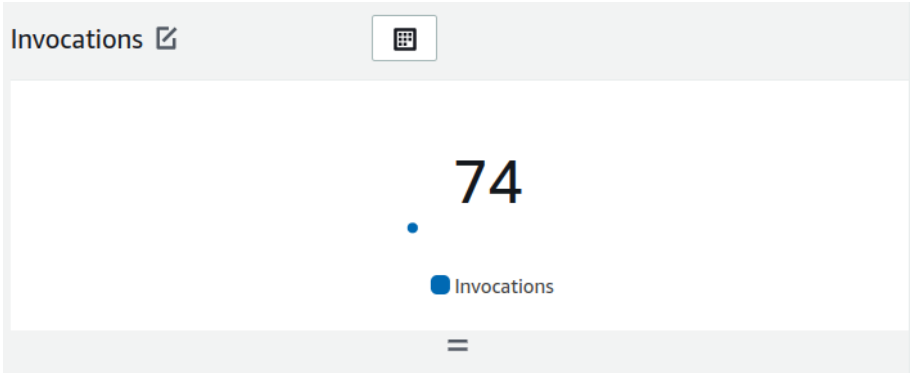

Figura 13 – Paínel do cloudwatch, número de invocações do AWS Lambda de notificação

4.2.2 Resultado escalabilidade ECS

O objetivo do teste foi estressar as APIs que estão no ECS, realizando milhares de requests por segundo, a fim de atingir um consumo de memória superior a 35%. Dessa forma, de acordo com a configuração do *Auto Scaling*, deveria provisionar mais uma tarefa (*task*), totalizando duas tarefas em execução (*running*). As figuras 14, 15 e 16 mostram, respectivamente, o ECS inicialmente com uma tarefa em execução (*running*), o consumo de memória superior a 35% e o ECS com duas tarefas em execução (*running*).

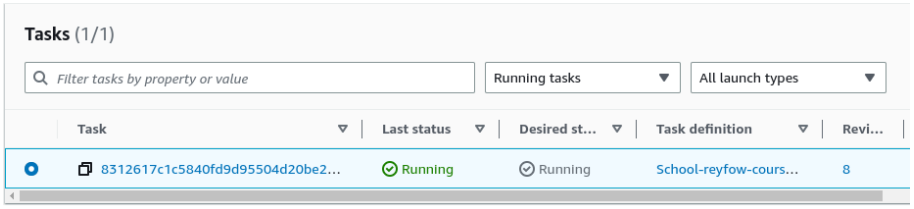

Figura 14 – ECS com uma task em running

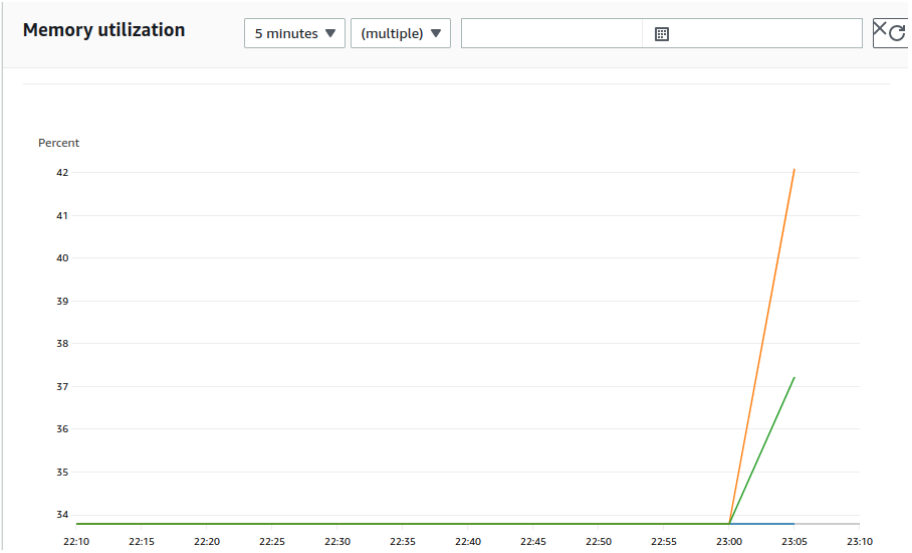

Figura 15 – Metricas do ECS com consumo de memoria acima dos 35 por cento

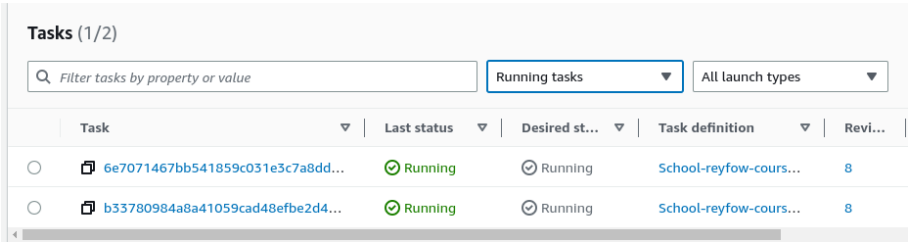

Figura 16 – ECS com duas tasks em running

Por fim quando o consumo de memória volta ser abaixo de 35% voltamos a ater apenas tarefas em execução (*running*).

4.3 Resultados monitoramento

O objetivo do teste foi utilizar os recursos do sistema para podermos monitorá-los utilizando o *CloudWatch*, que oferece um ecossistema de observabilidade como métricas e logs.

A figura 17 mostra como podemos visualizar os logs da nossa aplicação de controle de estudantes no *Cloudwatch*.

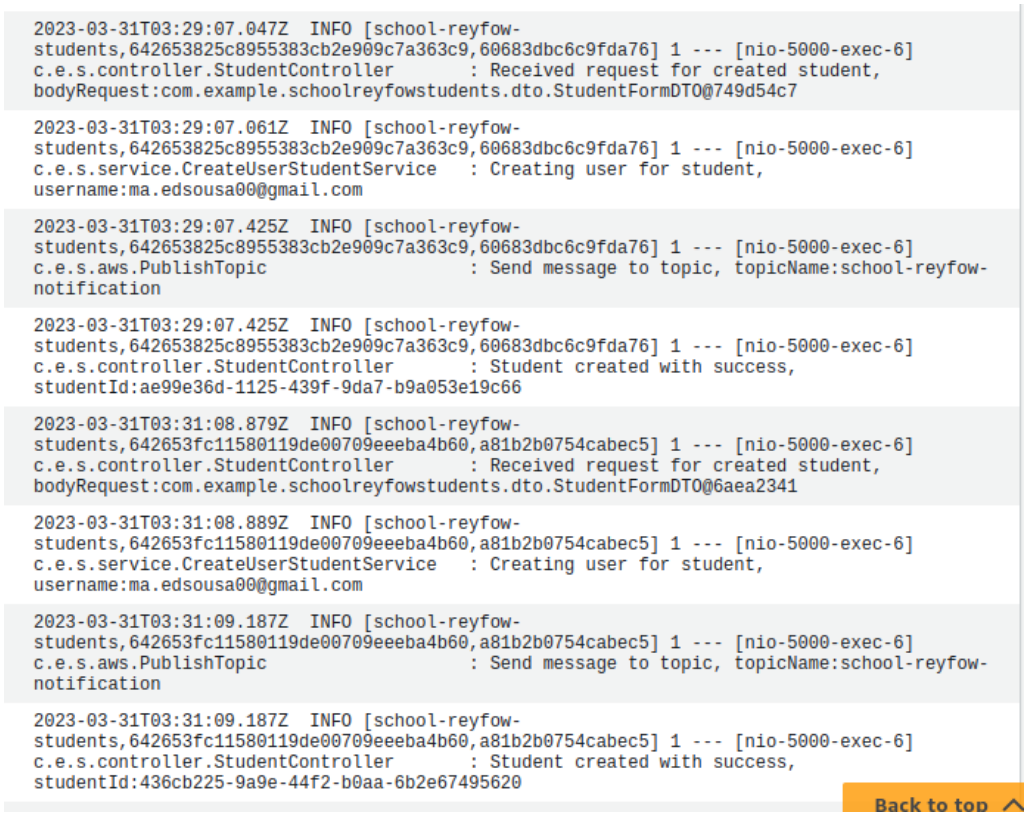

Figura 17 – Logs da API da API de controle de estudantes no CloudWatch.

A figura 18 e 19 mostram métricas de consumo de CPU e memória da aplicação de controle de estudantes.

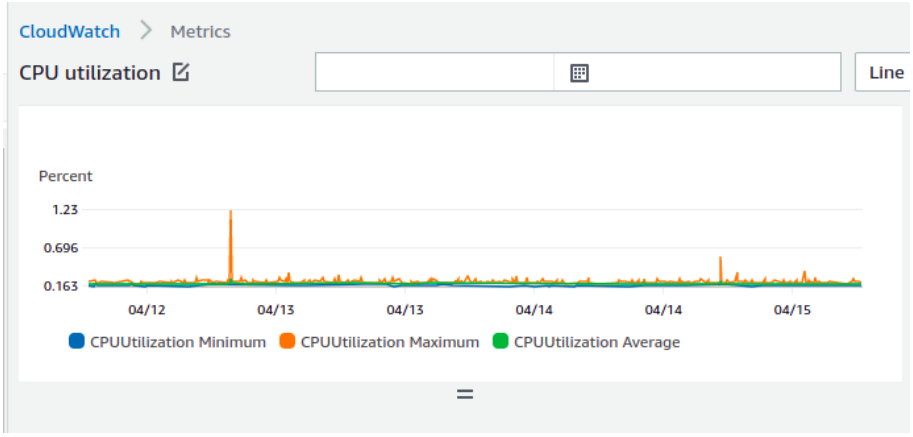

Figura 18 – Consumo de CPU da API de controle de estudantes no CloudWatch.

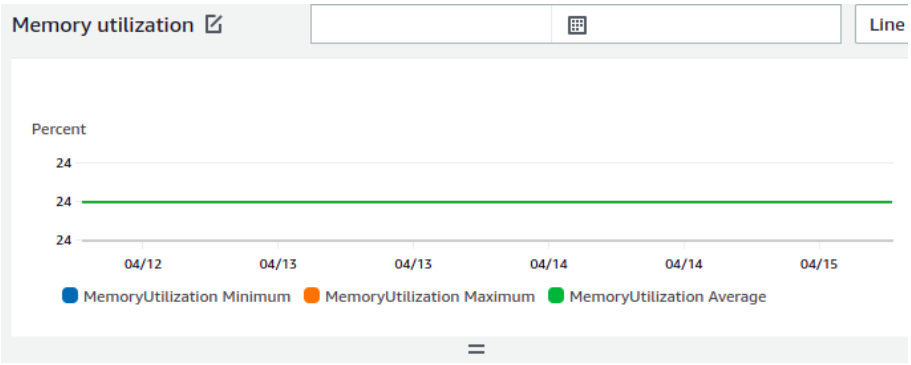

Figura 19 – Consumo de memória da API de controle de estudantes no CloudWatch.

Ao subir qualquer aplicação com ECS, a AWS já fornece essas métricas sem precisarmos realizar qualquer configuração. Mas também podemos personalizar diversos recursos. A imagem 20 mostra um alarme personalizado e a imagem 21 mostra o e-mail enviado pelo alarme após ser acionado. O intuito do alarme é disparar um e-mail caso a fila de DLQ receba pelo menos 5 mensagens na fila de DLQ de notificações.

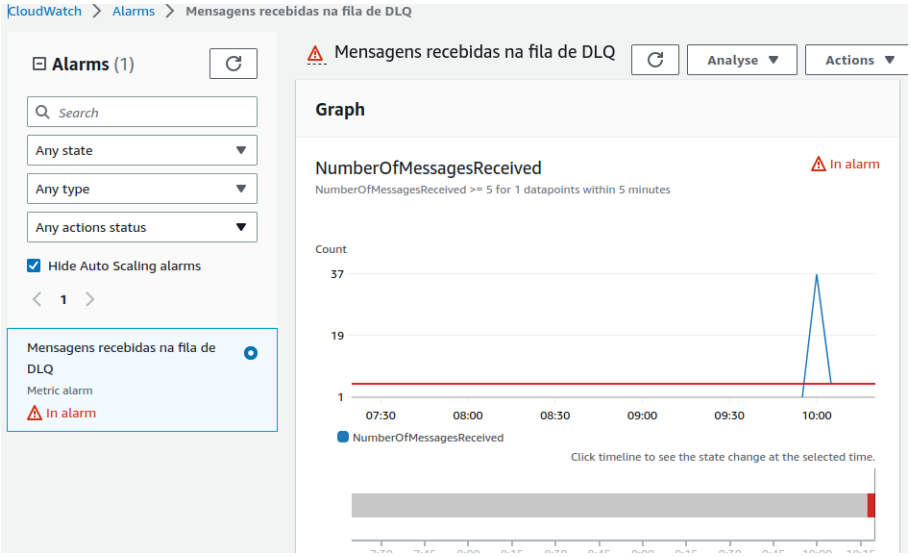

Figura 20 – Alarme configurado

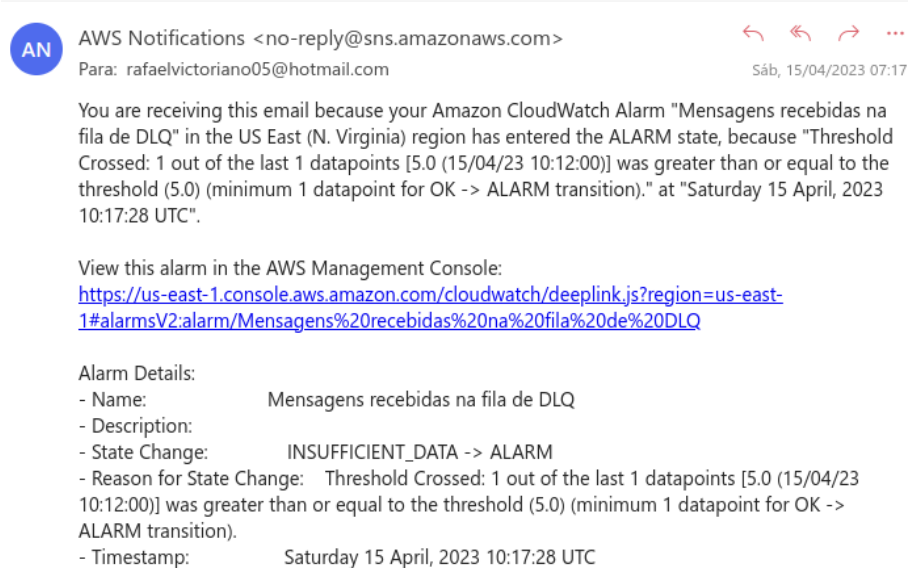

Figura 21 – Email notificado após fila DLQ de eventos receber mais de 5 mensagens

Com todas essas ferramentas de monitoramento e alertas, ganhamos agilidade na detecção de erros em nossas aplicações.

4.4 Resultado comunicação entre serviços e clientes

O objetivo do teste foi demosntrar a comunicação entre serviços e cliente funcionando.

A Figura 22 e 23 mostram logs do envio de uma notificação de que um estudante foi registrado na plataforma e que foi inscrito em um curso para um tópico que, posteriormente, passa por um SQS, sendo consumido pelo lambda de notificações que dispara um e-mail notificando o cliente de que ele foi inscrito na plataforma de cursos, além de outro e-mail

que o notifica de que ele foi inscrito no curso API - Java Quarkus. As Figuras 24 e 25 apresentam os e-mails enviados.

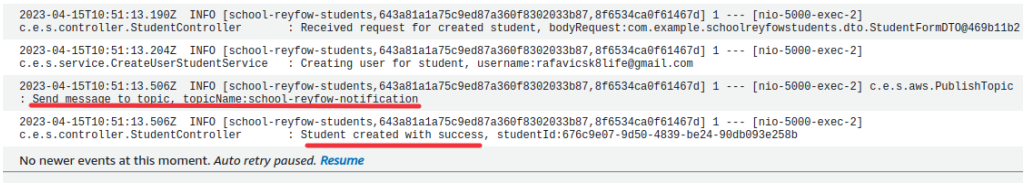

Figura 22 – Logs envio de evento para estudante registrado na plataforma

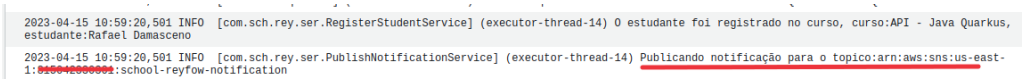

Figura 23 – Logs envio de evento para estudante registrado no curso API - Java Quarkus

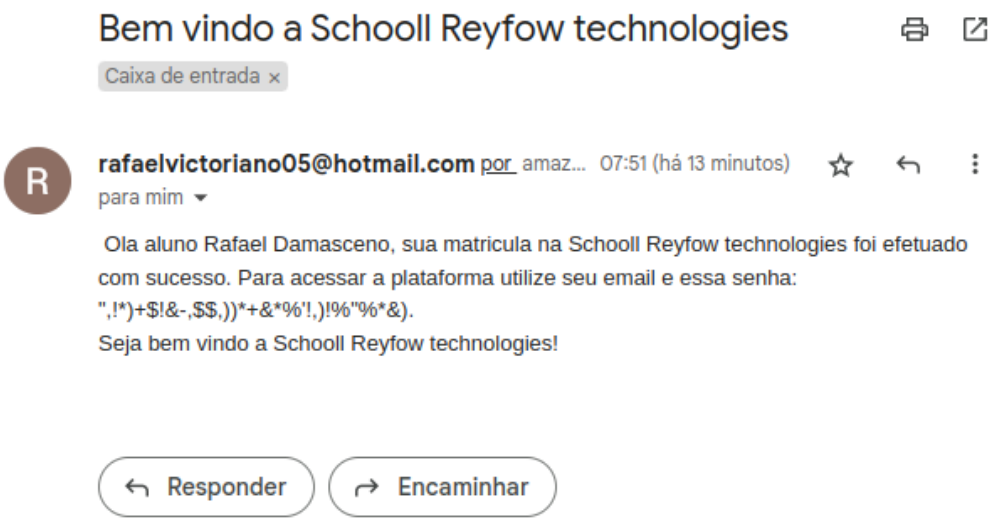

Figura 24 – Email notificando que estudante foi registrado na plataforma

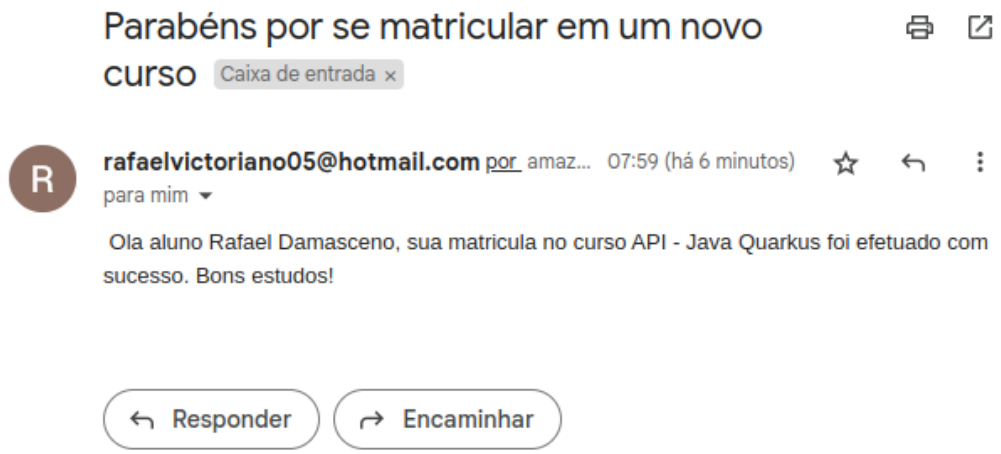

Figura 25 – Email notificando que estudante se inscreveu no curso API - Java Quarkus

Por trás das cortinas, os microsserviços de cursos e estudantes enviam uma notificação informando que o evento X ocorreu, e de forma assíncrona, nossa lambda de notificações envia notificações para os clientes.

5 Considerações finais

Com base nos resultados apresentados neste trabalho, pode-se concluir que a adoção de microsserviços em um ambiente em nuvem é uma solução muito boa para empresas que buscam escalabilidade, disponibilidade e flexibilidade em suas aplicações. A utilização de ferramentas como ECS, SQS, Docker, SNS, Lambda e *Load Balancer* proporciona um ambiente robusto e escalável para a execução de microsserviços.

Os testes realizados comprovaram a eficácia das soluções propostas para o gerenciamento de microsserviços, como o uso de orquestradores para gerenciamento de contêineres, recursos de comunicação e o uso de ferramentas de monitoramento e alerta. Além disso, foi possível verificar que a arquitetura de microsserviços pode trazer ganhos significativos em termos de desempenho e facilidade de manutenção.

Entretanto, é importante destacar que a adoção de microsserviços também pode trazer alguns desafios, como a complexidade na segurança, escalabilidade, na comunicação entre serviços e a necessidade de uma equipe especializada em sua implementação e gerenciamento. Portanto, é preciso avaliar cuidadosamente a viabilidade da adoção dessa arquitetura em cada caso específico.

Por fim, sugere-se que pesquisas futuras explorem mais profundamente os aspectos relacionados à orquestração de containers, utilizem o EKS, que é uma implementação do *Kubernetes* na AWS, utilizem *Step Functions* para gerenciar fluxos, e deem uma ênfase maior em boas práticas no desenvolvimento dos microsserviços. Também é importante destacar a necessidade de se realizar testes mais complexos e aprofundados para avaliar a eficácia dessas soluções em ambientes reais de produção.

Referências

AUGUSTO, G. **Java**: tudo o que você precisa saber para começar. 2021. Disponível em: <https://www.zup.com.br/blog/java>. Acesso em: 14 mai. 2022.

AWS. **Autoescalabilidade do serviço**. 2016. Disponível em: <https://docs.aws.amazon.com/pt_br/AmazonECS/latest/userguide/service-auto-scaling.html>. Acesso em: 08 mai. 2022.

AWS. **Implantar uma aplicação Web em contêiner no Amazon ECS**. 2020. Disponível em: <https://aws.amazon.com/pt/getting-started/guides/deploy-webapp-ecs/module-one/>. Acesso em: 08 mai. 2022.

AWS. **O que são microsserviços?** 2021. Disponível em: <https://aws.amazon.com/pt/microservices/>. Acesso em: 1 mai. 2022.

AWS. **Amazon DynamoDB**. 2022. Disponível em: <https://aws.amazon.com/pt/dynamodb/>. Acesso em: 08 Out. 2022.

AWS. **Computação em nuvem com a AWS**. 2022. Disponível em: <https://aws.amazon.com/pt/what-is-aws/>. Acesso em: 08 mai. 2022.

AZURE. **O que é computação em nuvem?** 2022. Disponível em: <https://azure.microsoft.com/pt-br/overview/what-is-cloud-computing/#benefits>. Acesso em: 05 mai. 2022.

CAMELO, R. **Monólitos, serviços e microsserviços: impactos nos negócios**. 2020. Disponível em: <https://softdesign.com.br/blog/monolitos-servicos-e-microsservicos-impactos-nos-negocios>. Acesso em: 07 mai. 2022.

COUTINHO, M. Saiba mais sobre streaming, a tecnologia que se popularizou na web 2.0. setembro 2014. Disponível em: <https://www.techtudo.com.br/noticias/2013/05/conheca-o-streaming-tecnologia-que-se-popularizou-na-web.ghtml>. Acesso em: 09 mai. 2022.

FELIX, W. **Escalabilidade vertical vs escalabilidade horizontal**. 2020. Disponível em: <https://waldyrfelix.com.br/escalabilidade-vertical-vs-escalabilidade-horizontal-6a3981783477>. Acesso em: 07 mai. 2022.

FOWLER, M.; LEWIS, J. **Microservices**. 2014. Disponível em: <https://martinfowler.com/articles/microservices.html>. Acesso em: 08 mai. 2022.

GEEKHUNTER. **Spring Framework: o que é, seus módulos e exemplos!** 2020. Disponível em: <https://blog.geekhunter.com.br/spring-framework/>. Acesso em: 06 mai. 2022.

IBM. **Docker**. 2021. Disponível em: <https://www.ibm.com/br-pt/cloud/learn/docker?msclkid=6c5a8467cf2411ec847d2cdc7ad1046e>. Acesso em: 05 mai. 2022.

LINDERS, B. **Microservices no Spotify**. 2016. Disponível em: <https://www.infoq.com/br/news/2016/03/microservices-spotify/>. Acesso em: 08 mai. 2022.

LTS. Arquitetura de microsserviÇos: Exemplos escalÁveis. janeiro 2021. Disponível em: <https://ltsconsulting.com.br/arquitetura-de-microsservicos-exemplos-escalaveis/>. Acesso em: 10 mai. 2022.

MONTE, D. P. R. d. et al. Arquitetura de microsserviços: quando vale a pena migrar? Jaboatão dos Guararapes, 2020.

MULLER, R. H.; MEINHARDT, C.; MENDIZABAL, O. M. Microsserviços aplicados no gerenciamento de dados de vistorias imobiliárias: um estudo de caso. In: SBC. **Anais do XVIII Workshop em Clouds e Aplicações**. [S.l.], 2020. p. 41–54.

PEREIRA, C. R. **Aplicações web real-time com Node. js**. [S.l.]: Editora Casa do Código, 2014.

PEREIRA, M. M.; BACK, G.; JÚNIOR, N. W. Arquitetura baseada em microserviço. Junho 2019.

REDHAT. **O que é Quakus?** 2023. Disponível em: <https://www.redhat.com/pt-br/topics/cloud-native-apps/what-is-quarkus>. Acesso em: 02 abr. 2023.

RIBEIRO, L. **Insomnia, um poderoso testador de rotas**. 2020. Disponível em: <https://lucassr.medium.com/insomnia-um-poderoso-testador-de-rotas-3d77d2cd8e89>. Acesso em: 02 abr. 2023.

ROSA, T. P. Um método para o desenvolvimento de software baseado em microsserviços. 2016.

ROVEDA, U. **O que é git e github, como usar e quais são as vantagens?** 2021. Disponível em: <https://kenzie.com.br/blog/o-que-e-git/>. Acesso em: 07 mai. 2022.

SANTOS, L. Saiba mais sobre streaming, a tecnologia que se popularizou na web 2.0. dezembro 2021. Disponível em: <https://www.techtudo.com.br/listas/2021/12/o-que-e-amazon-prime-veja-5-perguntas-e-respostas-sobre-o-servico.ghtml>. Acesso em: 10 mai. 2022.

SANTOS, W. R. M. **Adaptação de aplicações baseadas em microsserviços usando aprendizagem de máquina**. Dissertação (Mestrado) — Universidade Federal de Pernambuco, 2020.

SOUSA, F. R.; MOREIRA, L. O.; MACHADO, J. C. Computação em nuvem: Conceitos, tecnologias, aplicações e desafios. **II Escola Regional de Computação Ceará, Maranhão e Piauí (ERCEMAPI)**, p. 150–175, 2009.

SOUZA, A. V. F. de. **O que é intellij idea?** 2022. Disponível em: <https://eng-linear.blogspot.com/2019/08/o-que-e-intellij-idea.html>. Acesso em: 12 mai. 2022.

SOUZA, Í. A. de; PELISSARI, W. R. Microserviços como alternativa de arquitetura monolítica. **DIVERSITÀ: Revista Multidisciplinar do Centro Universitário Cidade Verde**, v. 3, n. 2, 2017.

SPOTIFY. https://support.spotify.com/br/article/what-is-spotify/. Novembro 2021. Disponível em: <https://support.spotify.com/br/article/what-is-spotify/>. Acesso em: 09 mai. 2022.

TERRAFORM. **O que é Terraform?** 2020. Disponível em: <https://www.terraform.io/intro>. Acesso em: 15 mai. 2022.

UBER. **Descubra o que é o Uber e saiba como ele funciona**. 2018. Disponível em: <https://www.uber.com/pt-BR/blog/o-que-e-uber/>. Acesso em: 08 mai. 2022.