

Microsserviços na Nuvem : potencializando serviços em produção

Rafael Marcos Victoriano Damasceno¹, Sergio Roberto Delfino²

¹Departamento de Segurança da Informação – Faculdade de Tecnologia de Ourinhos
Av. Vitalina Marcusso, 1440, Campus Universitário – Ourinhos – SP – Brasil

`rafael.damasceno@fatec.sp.gov.br, sergio.delfino@fatec.sp.gov.br`

Abstract. *This article aims to demonstrate how systems can be empowered using a cloud microservices architecture, addressing issues such as scalability, monitoring, security, and communication between services. Java with Quarkus and Spring Boot, Node JS for microservices, Docker for containers, and AWS as a cloud provider to use services such as SQS, Api Gateway, Load balancer, etc. were utilized. Four microservices were created: login, student control, course control, and a notification service that sends an email when a student registers on the platform, to perform configurations and solve the described problems.*

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 Quarkus 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. Foram 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.*

1. Introdução

Segundo [Muller et al. \(2020\)](#), 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 é necessário entender a arquitetura monolítica. Segundo [de Souza and 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 como um todo e se o serviço de um sistema estiver indisponível ele não afetará outros sistema conectados, somente a tarefa que está designado a fazer. Arquitetura de microsserviços traz muitos benefícios, como: pequenos sistemas especializados, agilidade, escalabilidade flexível, liberdade tecnológica, reutilização de código e serviços e resiliência.

Quando o modelo arquitetural de microsserviços é adotado, consequentemente se tem mais complexidade e é necessário pensar em como gerenciar contêineres, como as aplicações vão se comunicar, como monitorá-las e descentraliza-las de forma segura. Esses são alguns dos pontos que precisam ser solucionados ao considerar o uso desse modelo arquitetural.

Dessa forma, o objetivo do projeto é solucionar esses problemas utilizando um modelo arquitetural de microsserviços sendo potencializado pelo ambiente em nuvem utilizando o provedor *Amazon Web Services* (AWS) para resolver os problemas de comunicação, escalabilidade, monitoramento de serviço, segurança e implementar *Auto Scaling e Load Balancing*.

2. Revisão da Literatura

Este capítulo contém a revisão da literatura 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 tornarem uma única aplicação. Os serviços que compõe uma arquitetura voltada a microsserviços, podem ser escaláveis de forma independente e permitem a diversidade de tecnologias e linguagens utilizadas para cada serviço.

[Fowler and Lewis \(2014\)](#) relatam que comunicação entre serviços ocorre de forma leve, através do protocolo HTTP como Rest APIs, mensageria como SQS e RabbitMQ, eventos etc.

Aplicações que utilizam-se desta arquitetura são altamente escaláveis, segundo [Santos \(2020\)](#) a escalabilidade permite a adaptação de componentes individuais ou de toda

a aplicação à um balanceamento de carga dos serviços. Como esses serviços agem de forma particular e desacoplada, 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 and Lewis \(2014\)](#) cada serviço deve gerar sua própria base dados, mesmo que esses serviços utilizem instância 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 sua complexidade em geral, pelo fato de que as equipes precisam ter muito cuidado no desenvolvimento e na integração entre os serviços.

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 escaláveis e economias. A computação na nuvem ajuda a reduzir os custos operacionais, e torna a infraestrutura mais flexível, pelo fato de que o cliente não precisará mais se preocupar com hardware e poderá acessar sua infraestrutura de qualquer local.

Segundo [Sousa et al. \(2009\)](#) o modelo de computação em nuvem tem como principal objetivo oferecer serviços com alta disponibilidade, baixo custo e escalabilidade.

Para [Rosa \(2016\)](#) se tem três modelos de serviço na computação em nuvens:

- *Software as a Service* (SaaS) este modelo concentra na inovação e no desenvolvimento software pelo fato de que o usuário não administra ou controla a infraestrutura subjacente. O modelo de SaaS proporciona sistemas de software com propósitos específicos, que estão disponíveis para o usuário através da internet.
- *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.
- *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 exclusiva de uma organização, nuvem pública que é disponibilizada para qualquer usuário, nuvem comunidade quando há um compartilhamento de diversas empresas e nuvem híbrida que é a composição de duas ou mais nuvens com modelos de implantação diferentes. [Sousa et al. \(2009\)](#) 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.

A computação em nuvem é uma abordagem amplamente adotada por empresas. Grandes empresas como Microsoft, Google e Amazon possuem seus próprios ambientes em nuvem, que são frequentemente utilizados por outras 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 startups há empresas consolidadas [AWS \(2022b\)](#).

O *Amazon Elastic Container Service* (ECS) é um serviço da AWS para orquestração de contêineres, que ajuda a implantar, gerenciar e escalar facilmente aplicações em contêineres. Ele gerencia o ciclo 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 monitoramento, contêineres 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 há *logs* de aplicação. Com essas informações do *Cloudwatch* pode-se configurar o *Auto Scaling* para aumentar as *tasks definitions* da aplicação escalando de forma horizontal, sendo 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 utilizada para que quando ocorra um deploy de uma nova versão a aplicação continue funcionando. 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.

O AWS Lambda permite que você execute códigos sem provisionar e gerenciar servidores, o código só é executado quando necessário, permitindo diversos gatilhos para executá-lo. O serviço é pago apenas pelo tempo de computação, que se é consumido, não há cobrança para código em execução. Este recurso permite que a execução do código seja feita em diversas linguagens de programação, sendo administrada totalmente pela própria AWS [Pereira et al. \(2019\)](#).

2.3. Arquitetura monolítica

Segundo [Camelo \(2020\)](#) arquitetura monolítica é um padrão arquitetural onde se tem apenas uma aplicação, grande e completa. Uma única aplicação trata de todas as funcionalidades de um sistema. Esta aplicação é responsável por todas as camadas da solução desde o cliente ao servidor.

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 acomodando grande quantidades de dados.

2.4.1. Escalabilidade vertical

A escalabilidade vertical ocorre quando uma máquina não aguenta mais o volume de requisições e é aumentado a capacidade de memória e CPUs, este modelo é mais usado

para arquiteturas monolíticas.

É importante destacar que, segundo [Felix \(2020\)](#), essa estratégia é rápida, mas pode facilmente atingir o limite da máquina.

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 adicionado servidores mais simples em vez de ter apenas uma máquina poderosa.

2.5. Trabalhos Correlatos

Segundo o projeto abordado, há 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 cobrados 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 Connect* [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 disponível 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 da demanda do serviço ter aumentado significativamente, os desenvolvedores decidiram mudar da arquitetura monolítica para microsserviços, assim podendo usar variedade de linguagens de estruturas. Em 2021 a empresa contava com mais de 1.300 (mil e trezentos) microsserviços focados em melhorar a escalabilidade do app [LTS \(2021\)](#). Uber também utiliza o ambiente em nuvem para suas aplicações.

3. Metodologia

Este tópico descreve quais materiais e metodologias para conclusão desse trabalho.

3.1. Materiais

Nesse capítulo será apresentado os materiais e ferramentas que foram utilizados para criação desse trabalho.

3.1.1. IntelliJ IDEA

IntelliJ IDEA é um ambiente de desenvolvimento integrado para o desenvolvimento de software em Java. Desenvolvido pela JetBrains. Esta ferramenta contém duas versões *Community* (versão gratuita) e *Ultimate* (versão paga) [de 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 Microsystems, que em 2008, foi adquirido pela Oracle. A linguagem java

funciona em qualquer plataforma por conta da Máquina Virtual Java (JVM), pelo fato de que quando um programa java é compilado é transformado em bytes e logo depois interpretado pela JVM para executá-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 [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 código distribuído que auxilia no desenvolvimento de software, sendo a tecnologia de versionamento de código mais adotada atualmente. Segundo [Roveda \(2021\)](#) este sistema de versionamento pode registrar quaisquer alterações feitas no código de acordo com que o usuário quiser, armazenando essas informações, ou seja cada novo registro é criado uma nova versão do serviço que está sendo desenvolvido e caso seja necessário o desenvolvedor pode regressar de versão. Essa tecnologia ainda permite que vários desenvolvedores tenham acesso ao repositório de código do sistema.

3.1.7. Spring Framework

[Geekhunter \(2020\)](#) define Spring Framework como uma tecnologia desenvolvida para a plataforma Java baseada em padrões 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 infraestrutura todo configurado, para que o desenvolvedor foque somente na lógica 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 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 contêineres e é eficaz para ambientes serverless, de nuvem e Kubernetes.

O Quarkus vem com intuito de ser eficiente para ambientes de contêineres, permitindo o desenvolvimento rápido e eficaz de aplicações Java escaláveis e distribuídas em ambientes de nuvem e Kubernetes, tornando-se uma das opções ideais 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 para humanos sendo possível criar, alterar e reutilizar código em diversos ambientes [Terraform \(2020\)](#).

3.1.10. Docker

Segundo [IBM \(2021\)](#) Docker é uma plataforma de containerização. Ela permite que aplicações sejam empacotadas em contêineres, onde se é configurado bibliotecas, dependências e variáveis de ambiente da aplicação para que possa ser executada em qualquer sistema operacional. Esses contêineres facilitam na entrega de sistemas distribuídos pelo fato de que esta tecnologia é de fácil integração com ambientes de cloud.

3.2. Métodos

Nesse capítulo será apresentado os procedimentos e etapas que foram utilizados para criação desse trabalho.

3.2.1. Pesquisas teóricas

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.

3.2.2. Construção de arquitetura

A segunda fase, foi baseada em construir uma arquitetura para o melhor desempenho dos microsserviços e utilizar serviços da AWS que impulsionam em maior escalabilidade e resiliência. Então foi utilizado AWS lambda para ações simples (login, disparo de e-mails), ECS para realizar orquestrações de contêineres, DynamoDB para base dados e SNS

e SQS para eventos de modo assíncrono sem perder a resiliência. A figura 1 um mostra a arquitetura final do sistema.

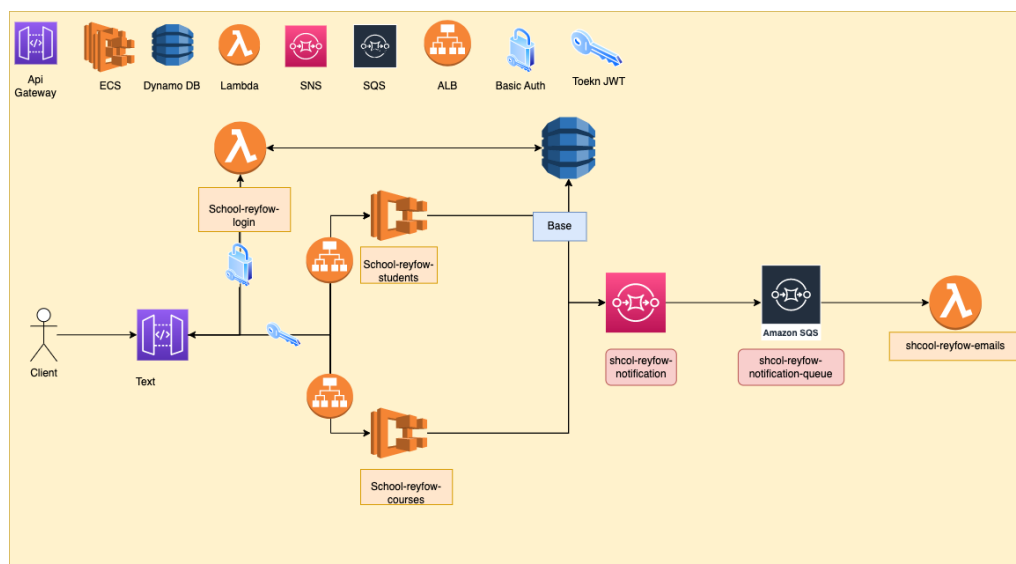


Figura 1. Arquitetura orientada a microsserviços. Fonte: [Elaborado pelo Autor.](#)

3.2.3. Modelagem de dados

Na terceira fase do projeto, foi realizada a construção da modelagem de dados, utilizando o DynamoDB como banco de dados. Foram elaboradas três tabelas, uma para o serviço de login, uma para o serviço de estudante e outra para o serviço de cursos. Vale ressaltar que, mesmo não havendo relação direta entre as tabelas, é possível obter informações de uma tabela em outra, como por exemplo o ID do estudante presente em duas tabelas diferentes. A figura 2 mostra a estrutura das tabelas.

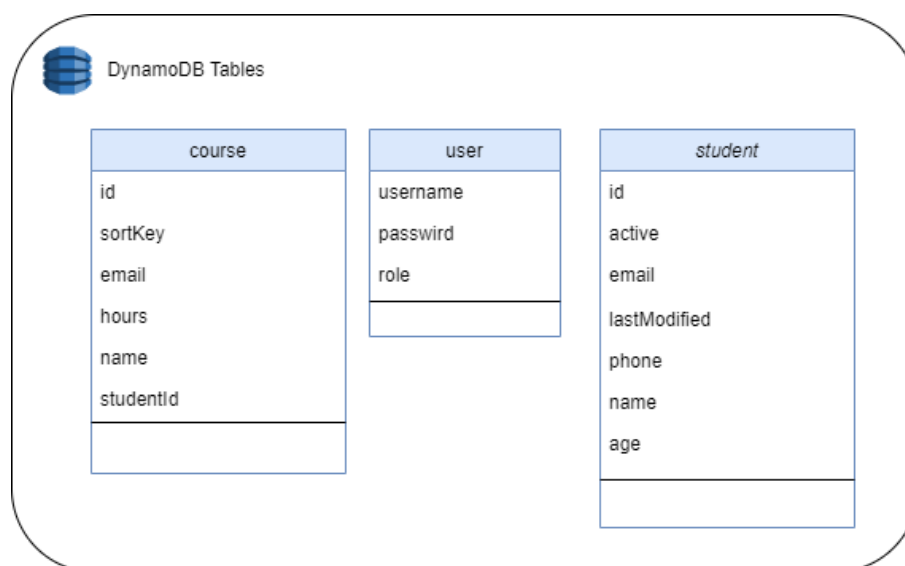


Figura 2. Modelagem da base de dados. Fonte: [Elaborado pelo Autor.](#)

3.2.4. Provisionamento de infraestrutura

A terceira fase, foi provisionar os recursos ECS, AWS Api Gateway, Lambda, SQS, SNS e DynamoDB na AWS utilizando terraform e o *framework serverless*. E iniciar configurações para resolver problemas descritos anteriormente.

3.2.5. Resolvendo escalabilidade

Escalabilidade é resolvida de duas formas. A primeira forma é com o AWS Lambda, onde sempre que houver uma chamada para o *endpoint* no API Gateway, esse lambda será 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. Por esse motivo, é desejável que as AWS Lambdas tenham um tempo de *startup* curto e um tempo de execução rápido. A segunda forma é utilizar o *Auto Scaling* nos ECS, onde será configurado para quando as aplicações chegarem a um consumo de memória superior a 35%, será provisionado outro contêiner para aplicação. Os códigos fontes 1 e 2 mostram a configuração do *Auto Scaling* para aplicações do ECS e a configuração de infraestrutura de um AWS Lambda utilizando o *framework Serverless*, respectivamente.

```

1 resource "aws_appautoscaling_policy" "policy" {
2   name           = "scale-out-policy"
3   policy_type    = "StepScaling"
4   resource_id    = aws_appautoscaling_target.tg_sc.resource_id
5   scalable_dimension = aws_appautoscaling_target.tg_sc.
     scalable_dimension
6   service_namespace = aws_appautoscaling_target.tg_sc.service_namespace
7
8   target_tracking_scaling_policy_configuration {
9     predefined_metric_specification {
10      predefined_metric_type = "ECSServiceAverageMemoryUtilization"
11    }
12    target_value = 35
13  }
14 }

```

Código-fonte 1. Configurando Auto Scaling. Fonte: [Elaborado pelo Autor](#).

```

1 service: school-reyfow-login
2 frameworkVersion: '3'
3
4 provider:
5   name: aws
6   runtime: nodejs16.x
7
8 functions:
9   login:
10    handler: app.lambdaHandler
11 plugins:
12   - serverless-plugin-typescript
13 custom:
14   serverlessPluginTypescript:
15     tsConfigFileLocation: './tsconfig.build.json'

```

Código-fonte 2. Configurando AWS Lambda. Fonte: [Elaborado pelo Autor](#).

3.2.6. Resolvendo *Load Balancer*

Load Balancing é resolvido com *Application Load Balancer* da AWS, que é um único ponto de acesso para o cliente, que irá distribuir o tráfego para as tasks do ECS *service*, se receber 100 requisições elas serão divididas entre as tasks, evitando sobrecarregar apenas uma. O código fonte 3 apresenta como adicionar o Load Balancer ao ECS via terraform.

```
1 load_balancer {
2   target_group_arn = var.tg-arn
3   container_name   = "School-reyfow-courses"
4   container_port   = 8080
5 }
```

Código-fonte 3. Configuração infra AWS Lambda com framework serverless.
Fonte: [Elaborado pelo Autor.](#)

3.2.7. Resolvendo *comunicação*

Comunicação é resolvida com o AWS Api Gateway que disponibiliza *endpoints* para acesso do Lambda e para o *load balancer* do ECS. A 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 se publica uma notificação ao SNS e ele distribuirá para os inscritos, que nesta arquitetura é o SQS. Os códigos-fonte 4 e 5 demonstram como criar um endpoint no API Gateway via Terraform e como publicar uma notificação em um tópico do SNS utilizando Java, respectivamente.

```
1
2 resource "aws_api_gateway_resource" "student_resource" {
3   rest_api_id = aws_api_gateway_rest_api.school-reyfow.id
4   parent_id   = aws_api_gateway_resource.geral_resource.id
5   path_part   = "student"
6 }
7
8 resource "aws_api_gateway_method" "student_method" {
9   rest_api_id   = aws_api_gateway_rest_api.school-reyfow.id
10  resource_id    = aws_api_gateway_resource.student_resource.id
11  http_method    = "POST"
12  authorization  = "NONE"
13
14  request_parameters = {
15    "method.request.header.Authorization" = true
16  }
17 }
18
19 resource "aws_api_gateway_integration" "student_integration" {
20   rest_api_id      = aws_api_gateway_rest_api.school-reyfow.id
21   resource_id      = aws_api_gateway_resource.student_resource.
22     id
23   http_method      = aws_api_gateway_method.student_method.
24     http_method
25   integration_http_method = "POST"
26   type              = "HTTP_PROXY"
27   uri                = "${var.alb-url}/student"
```

```
26
27 request_parameters = {
28     "integration.request.header.Authorization" = "method.request.header
29     .Authorization"
30 }
```

Código-fonte 4. Criando um endpoint no AWS Api gateway utilizando terraform.
Fonte: [Elaborado pelo Autor.](#)

```
1
2 @Slf4j
3 @RequiredArgsConstructor
4 @Service
5 public class PublishTopic {
6
7     private final NotificationMessagingTemplate
8     notificationMessagingTemplate;
9
10    @Value("${school-reyfow.topic-name}")
11    private String topicName;
12
13    public void pubTopic(Object event, Map<String, Object> headers) {
14        this.notificationMessagingTemplate.convertAndSend(topicName,
15        event, headers);
16        log.info("Send message to topic, topicName:{}", topicName);
17    }
18 }
```

Código-fonte 5. Publicando notificação no SNS utilizando Java. Fonte: [Elaborado pelo Autor.](#)

3.2.8. Resolvendo monitoramento

Monitoramento, foi resolvido com o *Cloudwatch* da AWS, onde é possível visualizar *logs*, latência e a saúde das aplicações.

3.2.9. Resolvendo segurança

Segurança é resolvido usando uma API de login sendo 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.

3.2.10. Aplicações

A ultima 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. E o desenvolvimento das aplicações do AWS lambda como API de login e de notificações.

4. Resultados

O objetivo dos resultados apresentados neste capítulo foi apresentar como uma arquitetura orientada a microsserviços pode ser pontencializada em um ambiente em nuvem, solucionando problemas comuns relacionados a essa abordagem arquitetural.

4.1. Resultado segurança

O objetivo dos testes foi avaliar a segurança da API de login por meio da inserção de logins e senhas inválidos, a fim de verificar se a API bloquearia o acesso, retornando o status 401, que indica que a requisição para efetuar o login foi negada. Em seguida, foi realizado um segundo teste utilizando credenciais válidas, a fim de obter um token JWT que permitiria o acesso às APIs do sistema.

A figura 3 apresenta a API de login bloqueando acesso.

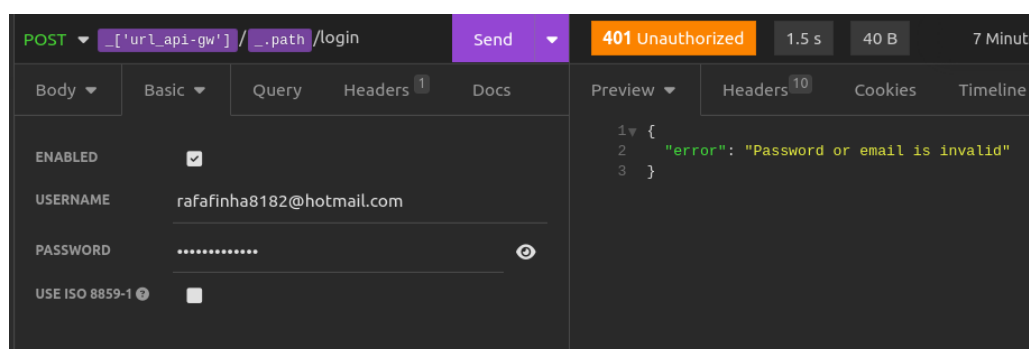


Figura 3. Bloqueando acesso com credenciais inválidas. Fonte: Elaborado pelo Autor.

A figura 4 demonstra a API de login gerando um token JWT, pelo fato das credenciais serem válidas.

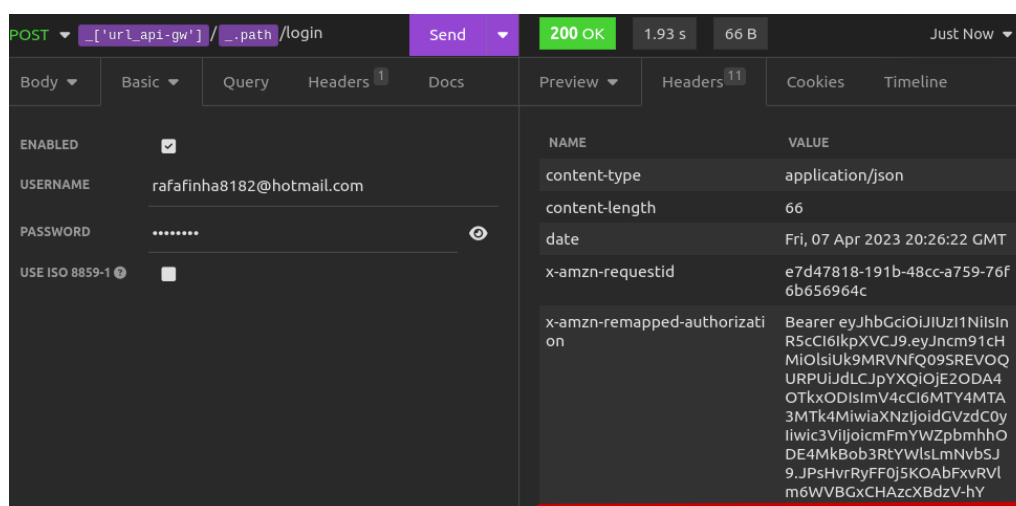


Figura 4. Gerando um token JWT. Fonte: Elaborado pelo Autor.

A figura 5 apresenta o acesso a API de consulta de cursos utilizando o token gerado pela API de login gerando um token JWT para se autenticar.

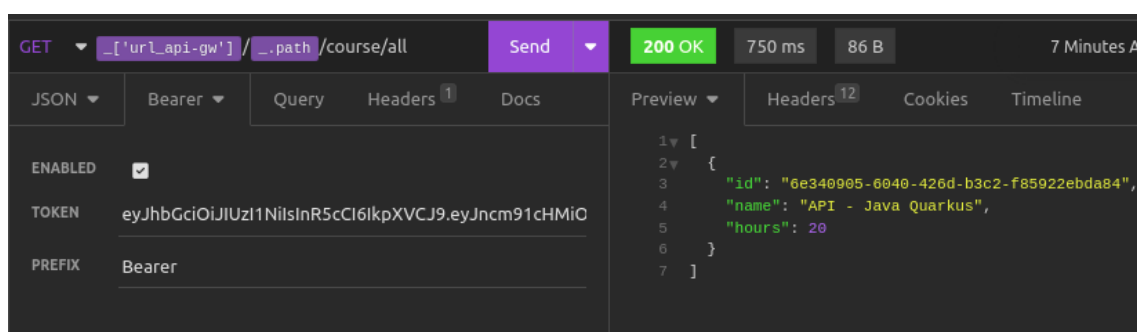


Figura 5. Consulta de cursos. Fonte: [Elaborado pelo Autor](#).

4.2. Resultado escalabilidade

O objetivo deste tópico foi demonstrar como a escalabilidade foi abordada por meio da utilização do *Auto Scaling*, ECS e AWS Lambda, solucionando assim possíveis problemas relacionados à demanda de recursos e ao gerenciamento de cargas de trabalho em uma arquitetura de microsserviços.

4.2.1. Resultado escalabilidade AWS Lambda

O objetivo do teste consistiu em realizar chamadas ao AWS API Gateway e enviar notificações para o SQS. Desta forma, foi possível acionar os AWS Lambdas, possibilitando a realização da escalabilidade necessária.

Através da Figura 6, é possível visualizar o número de invocações do AWS Lambda para a API de login, demonstrando a capacidade da AWS em escalar os recursos de acordo com a demanda das requisições. Tudo isso foi gerenciado pela plataforma, sem que houvesse a necessidade de nos preocuparmos com a infraestrutura subjacente. As métricas foram coletadas pelo *Cloudwatch*, e o lambda foi invocado através de um *endpoint* do AWS API Gateway.

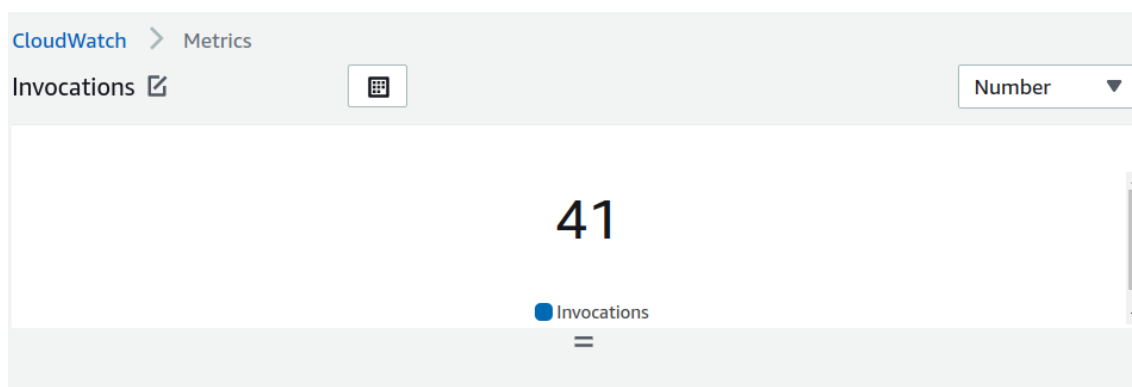


Figura 6. Painel do Cloudwatch, número de invocações do AWS Lambda de login. Fonte: [Elaborado pelo Autor](#).

A figura 7 apresenta o número de invocações do AWS Lambda responsável pelo envio de notificações por e-mail, cujas métricas são coletadas pelo *Cloudwatch*. Esse lambda é acionado a partir de um SQS, ou seja, sempre que há novas mensagens na fila do SQS, o AWS Lambda é invocado.

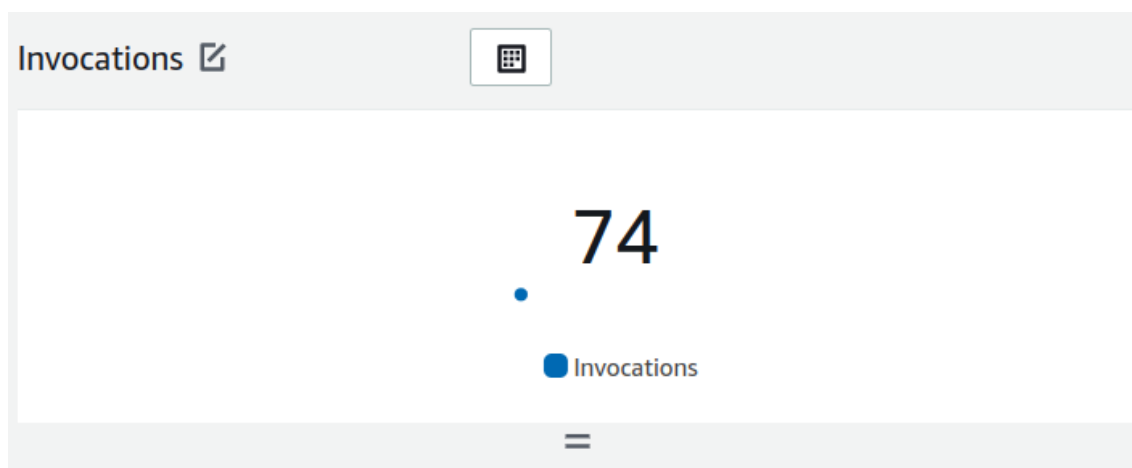


Figura 7. Painel do Cloudwatch, número de invocações do AWS Lambda de notificação. Fonte: [Elaborado pelo Autor](#).

4.2.2. Resultado escalabilidade ECS

O objetivo do teste foi sobrecarregar as APIs que estavam 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 ter sido provisionada mais uma tarefa (*task*), totalizando duas tarefas em execução (*running*). As figuras 8, 9 e 10 apresentam, respectivamente, o ECS inicialmente com uma tarefa em execução, o consumo de memória superior a 35% e o ECS com duas tarefas em execução.

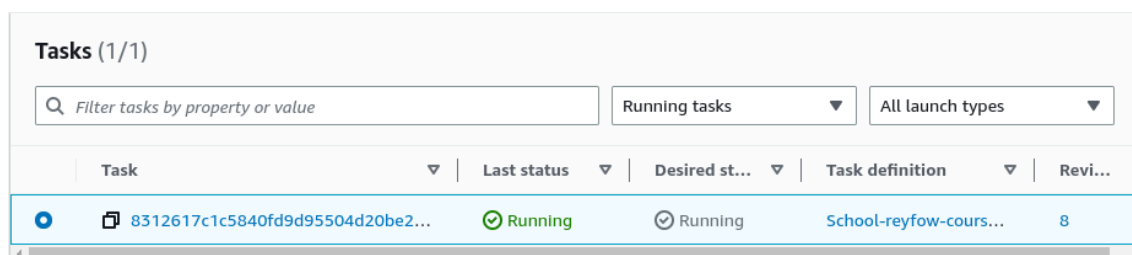
The image shows a screenshot of the AWS ECS console. At the top, it says 'Tasks (1/1)'. Below this is a search bar with the placeholder text 'Filter tasks by property or value'. To the right of the search bar are two dropdown menus: 'Running tasks' and 'All launch types'. Below these is a table with columns: 'Task', 'Last status', 'Desired st...', 'Task definition', and 'Revi...'. There is one row in the table with a blue circle icon, a task ID '8312617c1c5840fd9d95504d20be2...', a green checkmark icon, the status 'Running', a checkmark icon, the status 'Running', a task definition name 'School-reyfow-cours...', and the number '8'.

Figura 8. ECS com uma task em running. Fonte: [Elaborado pelo Autor](#).

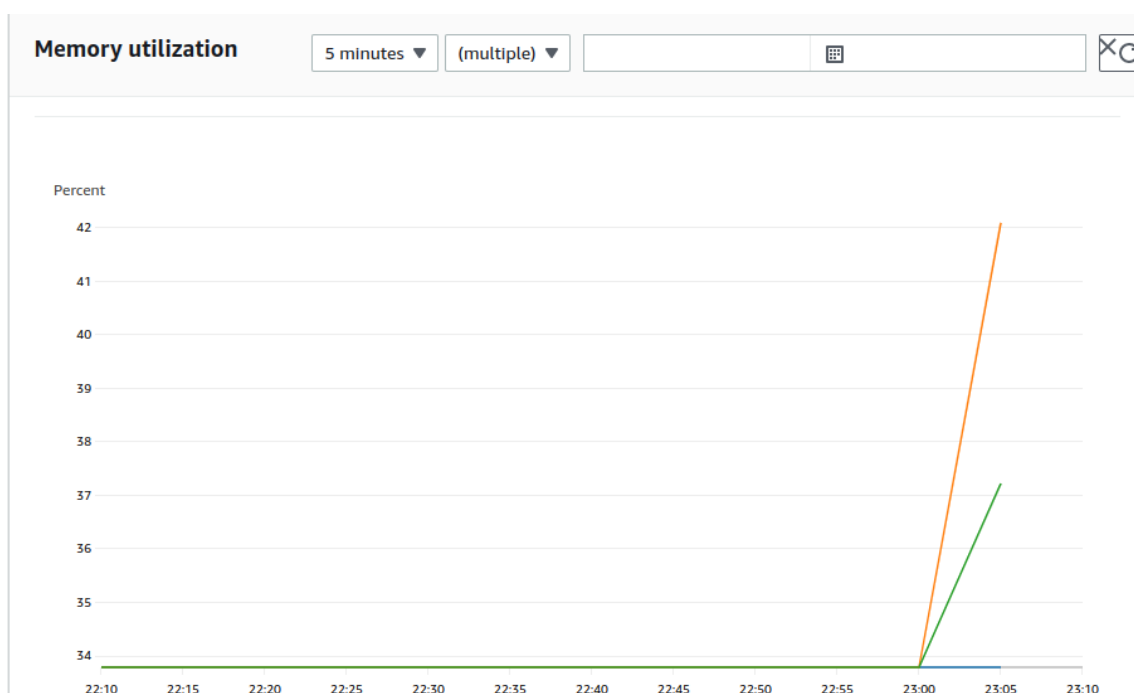


Figura 9. Metricas do ECS com consumo de memoria acima dos 35%. Fonte: Elaborado pelo Autor.

| Tasks (1/2) | | | | | |
|--|-------------|--|------------------------|---|--|
| <input type="text" value="Filter tasks by property or value"/> | | <input type="button" value="Running tasks"/> | | <input type="button" value="All launch types"/> | |
| Task | Last status | Desired st... | Task definition | Revi... | |
| <input type="radio"/> 6e7071467bb541859c031e3c7a8dd... | Running | Running | School-reyfow-cours... | 8 | |
| <input type="radio"/> b33780984a8a41059cad48efbe2d4... | Running | Running | School-reyfow-cours... | 8 | |

Figura 10. ECS com duas tasks em running. Fonte: Elaborado pelo Autor.

Por fim, quando o consumo de memória volta a ser inferior do que 35%, apenas uma tarefa em execução é mantida.

4.3. Resultados monitoramento

O teste consistiu em utilizar os recursos do sistema para monitorá-los por meio do *Cloudwatch*, que oferece um ecossistema de observabilidade que inclui métricas e *logs*.

A figura 11 mostra como visualizar os logs da aplicação de controle de estudantes no *Cloudwatch*.


```

2023-03-31T03:29:07.425Z INFO [school-reyfow-
students,642653825c8955383cb2e909c7a363c9,60683dbc6c9fda76] 1 --- [nio-5000-exec-6]
c.e.s.aws.PublishTopic : Send message to topic, topicName:school-reyfow
notification

2023-03-31T03:29:07.425Z INFO [school-reyfow-
students,642653825c8955383cb2e909c7a363c9,60683dbc6c9fda76] 1 --- [nio-5000-exec-6]
c.e.s.controller.StudentController : Student created with success,
studentId:ae99e36d-1125-439f-9da7-b9a053e19c66

2023-03-31T03:31:08.879Z INFO [school-reyfow-
students,642653fc11580119de00709eeeba4b60,a81b2b0754cabec5] 1 --- [nio-5000-exec-6]
c.e.s.controller.StudentController : Received request for created student,
bodyRequest:com.example.schoolreyfowstudents.dto.StudentFormDT0@6aea2341

2023-03-31T03:31:08.889Z INFO [school-reyfow-
students,642653fc11580119de00709eeeba4b60,a81b2b0754cabec5] 1 --- [nio-5000-exec-6]
c.e.s.service.CreateUserStudentService : Creating user for student,
username:ma.edsousa00@gmail.com

2023-03-31T03:31:09.187Z INFO [school-reyfow-
students,642653fc11580119de00709eeeba4b60,a81b2b0754cabec5] 1 --- [nio-5000-exec-6]
c.e.s.aws.PublishTopic : Send message to topic, topicName:school-reyfow
notification

2023-03-31T03:31:09.187Z INFO [school-reyfow-
students,642653fc11580119de00709eeeba4b60,a81b2b0754cabec5] 1 --- [nio-5000-exec-6]
c.e.s.controller.StudentController : Student created with success,
studentId:436cb225-9a9e-44f2-b0aa-6b2e67495620

```

[Back to top](#)

Figura 11. Logs da API de controle de estudantes no Cloudwatch. Fonte: Elaborado pelo Autor.

A figura 12 e 13 demonstram métricas de consumo de CPU e memória da aplicação de controle de estudantes.

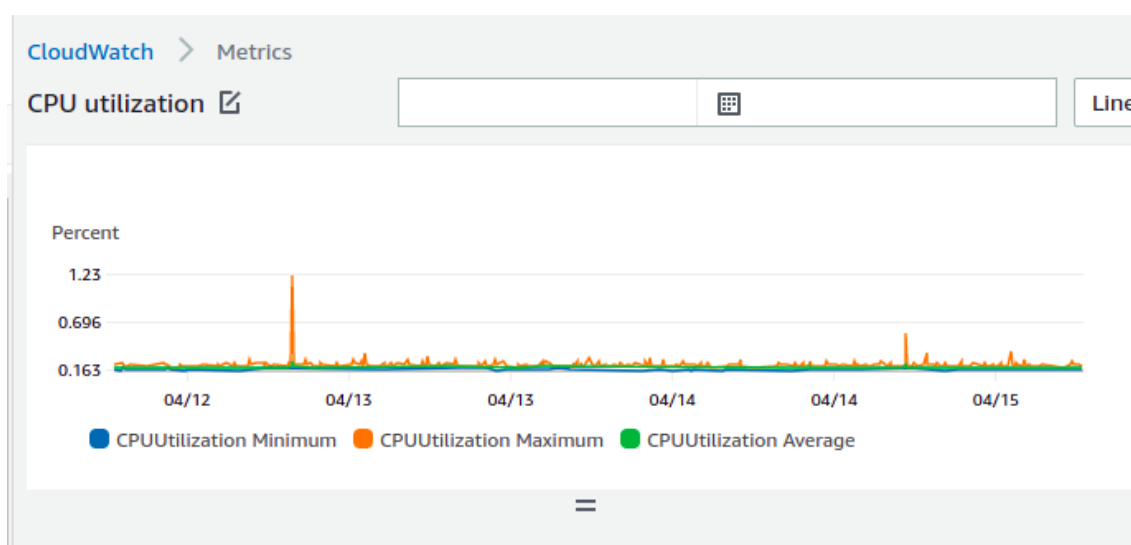


Figura 12. Consumo de CPU da API de controle de estudantes no Cloudwatch. Fonte: Elaborado pelo Autor.

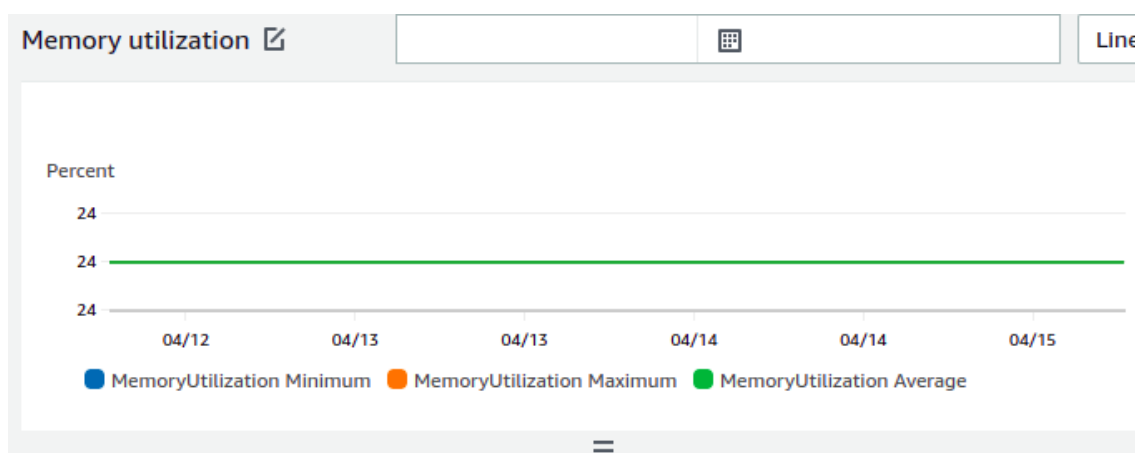


Figura 13. Consumo de memória da API de controle de estudantes no Cloudwatch. Fonte: Elaborado pelo Autor.

Ao ter uma aplicação implantada com ECS, a AWS já disponibiliza métricas sem a necessidade de realizar configurações adicionais. No entanto, ainda é possível personalizar vários recursos. Na figura 14, é possível visualizar um alarme personalizado, enquanto na figura 15, temos um exemplo de e-mail enviado pelo alarme após ser acionado. O objetivo do alarme é enviar um e-mail quando a fila de DLQ receber pelo menos 5 mensagens.

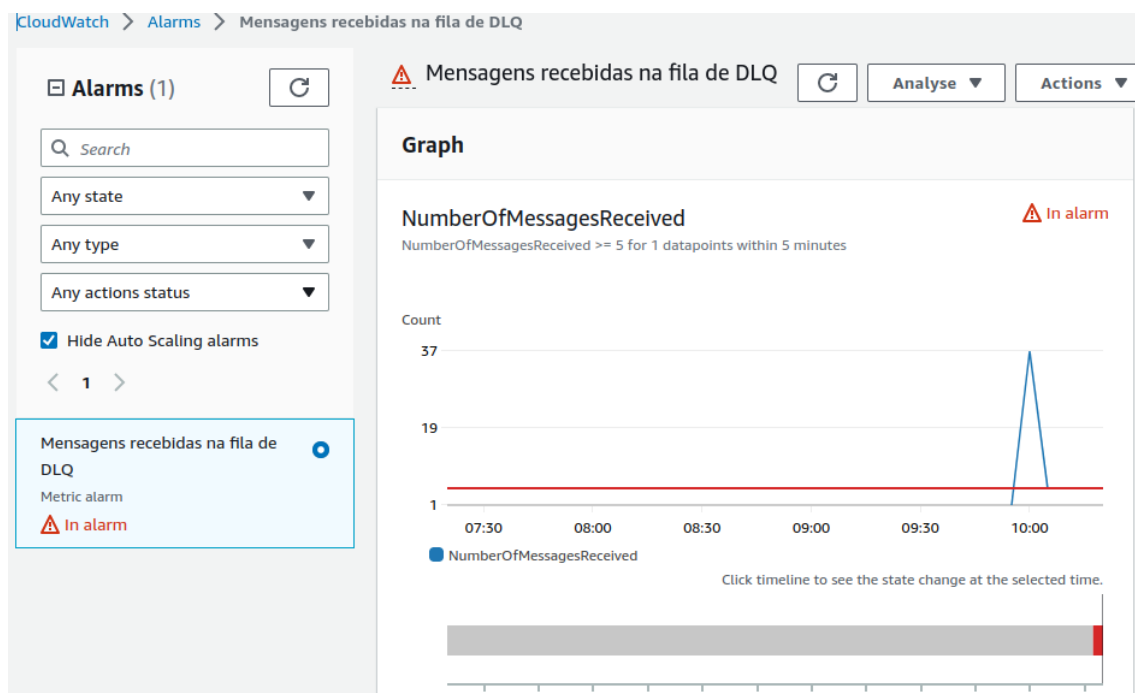


Figura 14. Alarme configurado. Fonte: Elaborado pelo Autor.

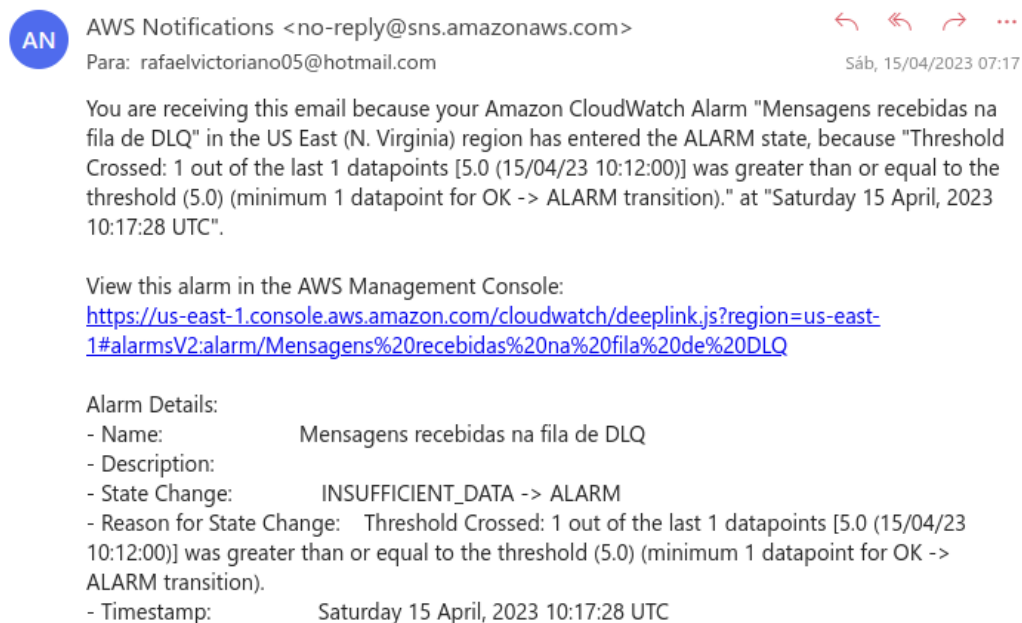


Figura 15. Email notificado após fila DLQ de eventos receber mais de 5 mensagens. Fonte: Elaborado pelo Autor.

Com todas essas ferramentas de monitoramento e alertas, ganha-se agilidade na detecção de erros nas aplicações.

4.4. Resultado comunicação entre serviços e clientes

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

As Figuras 16 e 17 apresentam os logs do envio de uma notificação, informando que um estudante foi registrado na plataforma e foi inscrito em um curso, que passa por um SQS e é consumido pelo lambda de notificações. Esse lambda dispara um e-mail notificando o cliente de que ele foi inscrito na plataforma de cursos, além de outro e-mail que o informa de que foi inscrito no curso API - Java Quarkus. As Figuras 18 e 19 mostram os e-mails enviados.

```
2023-04-15T10:51:13.190Z INFO [school-reyfow-students,643a81a1a75c9ed87a360f8302033b87,8f6534ca0f61467d] 1 --- [nio-5000-exec-2]
c.e.s.controller.StudentController : Received request for created student, bodyRequest:com.example.schoolreyfowstudents.dto.StudentFormDT0@469b11b2
2023-04-15T10:51:13.204Z INFO [school-reyfow-students,643a81a1a75c9ed87a360f8302033b87,8f6534ca0f61467d] 1 --- [nio-5000-exec-2]
c.e.s.service.CreateUserStudentService : Creating user for student, username:rafavicsk8l1fe@gmail.com
2023-04-15T10:51:13.506Z INFO [school-reyfow-students,643a81a1a75c9ed87a360f8302033b87,8f6534ca0f61467d] 1 --- [nio-5000-exec-2] c.e.s.aws.PublishTopic
: Send message to topic, topicName:school-reyfow-notification
2023-04-15T10:51:13.506Z INFO [school-reyfow-students,643a81a1a75c9ed87a360f8302033b87,8f6534ca0f61467d] 1 --- [nio-5000-exec-2]
c.e.s.controller.StudentController : Student created with success, studentId:676c9e07-9d50-4839-be24-90db093e258b
No newer events at this moment. Auto retry paused. Resume
```

Figura 16. Logs envio de evento para estudante registrado na plataforma. Fonte: Elaborado pelo Autor.

```
2023-04-15 10:59:20,501 INFO [com.sch.rey.ser.RegisterStudentService] (executor-thread-14) O estudante foi registrado no curso, curso:API - Java Quarkus, estudante:Rafael Damasceno
2023-04-15 10:59:20,501 INFO [com.sch.rey.ser.PublishNotificationService] (executor-thread-14) Publicando notificação para o tópico:arn:aws:sns:us-east-1:8500-12000002:school-reyfow-notification
```

Figura 17. Logs envio de evento para estudante registrado no curso API - Java Quarkus. Fonte: Elaborado pelo Autor.

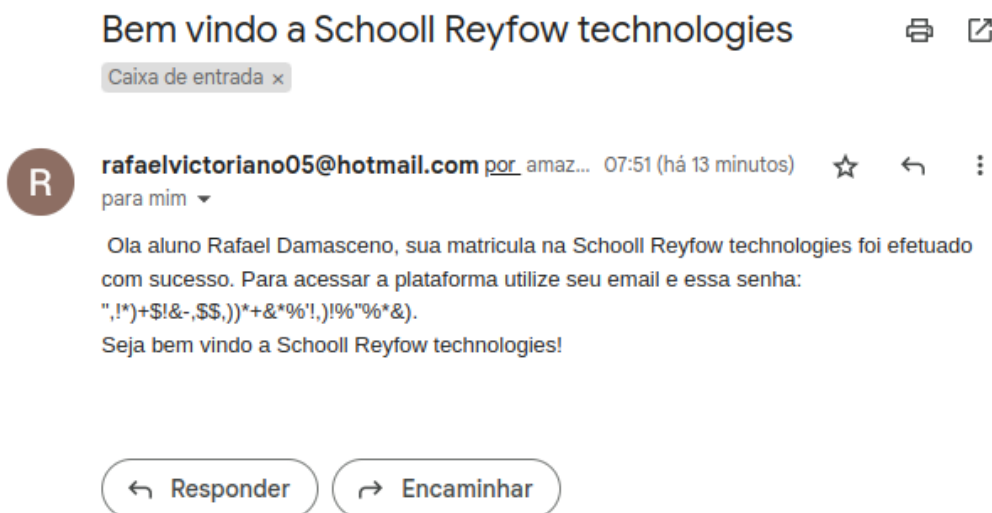


Figura 18. Email notificando que estudante foi registrado na plataforma. Fonte: Elaborado pelo Autor.

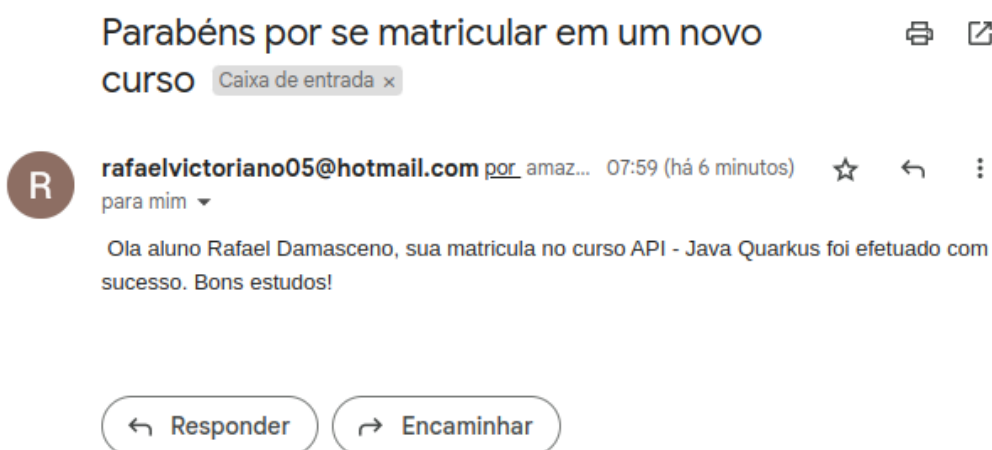


Figura 19. Email notificando que estudante se inscreveu no curso API - Java Quarkus. Fonte: Elaborado pelo Autor.

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

5. Conclusão

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* proporcionam 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 contêineres, 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. (2021). Java. 7
- AWS (2016). Autoescalabilidade do serviço. 4
- AWS (2020). Implantar uma aplicação web em contêiner no amazon ecs. 4
- AWS (2021). O que são microsserviços? 2
- AWS (2022a). Amazon dynamodb. 7
- AWS (2022b). Computação em nuvem com a aws. 4
- Azure (2022). O que é computação em nuvem? 3
- Camelo, R. (2020). Monólitos, serviços e microsserviços: impactos nos negócios. 4
- Coutinho, M. (2014). Saiba mais sobre streaming, a tecnologia que se popularizou na web 2.0. 5
- de Souza, A. V. F. (2022). O que é intellij idea? 6
- de Souza, Í. A. and Pelissari, W. R. (2017). Microserviços como alternativa de arquitetura monolítica. *DIVERSITÁ: Revista Multidisciplinar do Centro Universitário Cidade Verde*, 3(2). 2
- Felix, W. (2020). Escalabilidade vertical vs escalabilidade horizontal. 5
- Fowler, M. and Lewis, J. (2014). Microservices. 2, 3
- Geekhunter (2020). Spring framework: o que é, seus módulos e exemplos! 7
- IBM (2021). Docker. 8
- Linders, B. (2016). Microservices no spotify. 5
- LTS (2021). Arquitetura de microsserviços. 5, 6
- Monte, D. P. R. d. et al. (2020). Arquitetura de microsserviços: quando vale a pena migrar? 2
- Muller, R. H., Meinhardt, C., and Mendizabal, O. M. (2020). Microsserviços aplicados no gerenciamento de dados de vistorias imobiliárias: um estudo de caso. In *Anais do XVIII Workshop em Clouds e Aplicações*, pages 41–54. SBC. 2
- Pereira, C. R. (2014). *Aplicações web real-time com Node.js*. Editora Casa do Código. 7
- Pereira, M. M., Back, G., and Júnior, N. W. (2019). Arquitetura baseada em microserviço. 4
- RedHat (2023). O que é quakus? 8
- Ribeiro, L. (2020). Insomnia, um poderoso testador de rotas. 6
- Rosa, T. P. (2016). Um método para o desenvolvimento de software baseado em microsserviços. 3
- Roveda, U. (2021). O que é git e github, como usar e quais são as vantagens? 7
- Santos, L. (2021). Saiba mais sobre streaming, a tecnologia que se popularizou na web 2.0. 5
- Santos, W. R. M. (2020). Adaptação de aplicações baseadas em microsserviços usando aprendizagem de máquina. Master's thesis, Universidade Federal de Pernambuco. 2
- Sousa, F. R., Moreira, L. O., and Machado, J. C. (2009). Computação em nuvem: Conceitos, tecnologias, aplicações e desafios. *II Escola Regional de Computação Ceará, Maranhão e Piauí (ERCEMAPI)*, pages 150–175. 3
- Spotify (2021). <https://support.spotify.com/br/article/what-is-spotify/>. 5
- Terraform (2020). O que é terraform? 8
- Uber (2018). Descubra o que é o uber e saiba como ele funciona. 6