INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Automatização da Entrega do Produto ao Cliente

PETRA LOPES PISCO
Setembro de 2025

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Automatização da Entrega do Produto ao Cliente

Petra Lopes Pisco

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Engenharia de Software

Orientador: Paulo Sousa
Supervisor: Pedro Marquinhos

Porto, 28 de setembro de 2025

Declaração de Integridade

Declaro ter conduzido este trabalho académico com integridade.

Não plagiei ou apliquei qualquer forma de uso indevido de informações ou falsificação de resultados ao longo do processo que levou à sua elaboração.

Portanto, o trabalho apresentado neste documento é original e de minha autoria, não tendo sido utilizado anteriormente para nenhum outro fim. As exceções estão explicitamente reconhecidas na secção "Considerações éticas" do primeiro capítulo. Esta secção também declara como as ferramentas de IA foram utilizadas e para que finalidade.

Declaro ainda que tenho pleno conhecimento do Código de Conduta Ética do P.PORTO.

ISEP, Porto, 28 de setembro de 2025

Dedicatória

Aos meus pais e irmão, que sempre me incutiram o valor da perseverança e me apoiaram incondicionalmente ao longo de todo o percurso. Sem eles, não teria chegado até aqui.

À Erica Jesus, ao Luis Salomé, à Mónica Pereira, ao Rodrigo Tavares e ao Tiago Mairos, por estarem sempre presentes — mesmo nos momentos mais exigentes — com apoio, humor e uma assertividade que tantas vezes me ajudou a seguir em frente.

Resumo

A presente dissertação investiga o processo de *deployment* manual do PlexHub, um sistema modular e orientado a microsserviços desenvolvido pela PlexIT para o setor vinícola. O modelo atual, inteiramente manual, envolve tarefas repetitivas e configurações personalizadas, como modificações de configurações por cliente, migrações de bases de dados e instalação direta em máquinas virtuais, o que resulta em ineficiência, morosidade e alta vulnerabilidade a erros. Com isso, propõe-se uma solução baseada em práticas de DevOps, com foco na automação através de pipelines de CI/CD, utilizando ferramentas amplamente adotadas, como GitHub Actions, Docker e Docker Compose. O objetivo é aumentar a fiabilidade, reduzir o tempo de instalação e garantir maior consistência entre os ambientes.

Para validar a abordagem proposta, foi desenvolvido um ambiente de testes que simula as condições reais das máquinas virtuais dos clientes. Os resultados demonstram uma melhoria significativa no tempo de *deployment*. Este estudo visa não apenas otimizar os processos internos da PlexIT, mas também oferecer uma contribuição valiosa para a investigação e aplicação de práticas DevOps em contextos empresariais com recursos limitados.

Palavras-chave: *Deployment Automation, DevOps, CI/CD, GitHub Actions, Docker, Microservices*

Abstract

This dissertation investigates the manual deployment process of PlexHub, a modular and microservices-based system developed by PlexIT for the wine sector. The current model, fully manual, involves repetitive tasks and custom configurations, such as client-specific configuration changes, database migrations, and direct installation on virtual machines, which result in inefficiency, slowness, and high susceptibility to errors. Therefore, a solution based on DevOps practices is proposed, focusing on automation through CI/CD pipelines, using widely adopted tools such as GitHub Actions, Docker, and Docker Compose. The goal is to increase reliability, reduce installation time, and ensure greater consistency across environments.

To validate the proposed approach, a test environment was created to simulate the real conditions of client virtual machines. The results show a significant improvement in deployment time. This study aims not only to optimize PlexIT's internal processes but also to provide a valuable contribution to the research and practical application of DevOps strategies in business contexts with limited resources.

Agradecimentos

Ao Professor Paulo Gandra, meu orientador, expresso a mais sincera gratidão pela orientação dedicada, pela paciência inesgotável, pela compreensão e pelo incentivo constante que sempre me transmitiu. O seu apoio foi determinante não apenas para a concretização desta dissertação, mas também para o meu crescimento académico e pessoal.

Aos meus pais, agradeço de todo o coração. Foram o meu porto seguro, a minha inspiração e a minha maior força em cada etapa desta caminhada. Obrigada pelo amor incondicional, pelo apoio firme, pelos sacrifícios tantas vezes silenciosos e por acreditarem em mim mesmo quando eu própria duvidei. Sem vocês, nada disto teria sido possível.

Ao meu irmão, agradeço pela leveza e pelo humor com que sempre soube aliviar o peso deste percurso. Mesmo à distância, as tuas piadas e a tua forma simples de encarar a vida ajudaram-me a relativizar os momentos mais difíceis e a encontrar espaço para sorrir no meio da pressão.

Aos meus amigos de longa data — Erica Jesus, Luis Salomé, Mónica Pereira e Tiago Mairos — deixo um agradecimento profundo pela amizade genuína, pela alegria que trazem à minha vida, pela motivação constante e por nunca me deixarem percorrer este caminho sozinha. Partilhar convosco tantos momentos, desafios e conquistas transformou esta jornada numa verdadeira aventura. Crescer ao vosso lado tornou tudo mais leve e feliz.

Finalmente, aos meus colegas de curso — Beatriz Vaz, Tiago Pinto, Pedro Lemos, Rodrigo Tavares e José Oliveira — agradeço cada conversa, cada gesto de companheirismo e cada desafio ultrapassado em conjunto. Tornaram esta etapa única e inesquecível, e o vosso apoio foi essencial para que as dificuldades se transformassem em aprendizagens e os dias mais longos em memórias que levarei para sempre comigo.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Acrónimos

ACM/IEEE-CS	*Association for Computing Machinery / Institute of Electrical and Electronics Engineers - Computer Society*.
ACR	*Azure Container Registry*.
AMQP	*Advanced Message Queuing Protocol*.
CI/CD	*Continuous Integration / Continuous Delivery*.
ERP	*Enterprise Resource Planning*.
IaC	*Infrastructure as Code*.
IEEE	*Institute of Electrical and Eletronics Engineers*.
IIS	*Internet Information Services*.
IPP	Instituto Politécnico do Porto.
MR	*Merge Request(s)*.
PICOCS	*Population, Intervention, Comparison, Outcomes, Context and Study Design*.
PRISMA	*Preferred Reporting Items for Systematic reviews and Meta-Analyses*.
QA	*Quality Assurance*.
SCM	*Source Control Management System*.
SMS	*Systematic Mapping Study*.
SQL	*Standard Query Language*.
VPN	*Virtual Private Network*.

Capítulo 1

Introdução

O presente capítulo inclui a apresentação do contexto em que o problema se insere, a sua relevância e principais desafios, bem como uma descrição do problema e as motivações e benefícios da presente dissertação. Adicionalmente, apresentam-se os objetivos a atingir no decorrer da dissertação e os aspetos éticos que a dirigiram. Por fim, é detalhado o planeamento efetuado e apresentados os diferentes capítulos do documento, de forma a providenciar uma visão geral do seu fluxo.

1.1 Contexto e Problema

A *PlexIT* [1] é uma empresa especializada em soluções de infraestrutura tecnológica, *cibersegurança* e desenvolvimento de *software*. Foi fundada em 2018 com a missão de oferecer serviços personalizados e de alta qualidade, ajustados às necessidades específicas dos seus clientes.

Em 2023, a *PlexIT* criou uma equipa especializada no desenvolvimento de *software*, cujo trabalho culminou no *PlexHub*, um sistema de gestão concebido para otimizar operações, atualmente inserido no setor vinícola. O *PlexHub* é baseado numa arquitetura de microsserviços, integrando-se com *Enterprise Resource Planning* (ERP) preexistentes, como o Cegid [2], e oferece uma interface gráfica intuitiva que visa simplificar processos de negócios complexos, como a comercialização de produtos, suporte ao cliente e gestão de recursos humanos.

Atualmente, o processo de entrega e atualização do *PlexHub* aos clientes envolve uma sequência extensa de etapas realizadas manualmente. Estas incluem: a criação de *branches* de *release*, migração de bases de dados (utilizando *scripts Standard Query Language* ou *Entity Framework Migrations*), a inserção de dados padrão, a compilação e empacotamento (*zip*) dos serviços em artefactos, a configuração de conexões de rede (via *Virtual Private Network*) e adição dos serviços ao *Internet Information Services Manager* do ambiente remoto. Para cada novo cliente, é necessário ajustar configurações específicas, como conexões ao *provider* de autenticação e autorização e configuração personalizada deste, *message brokers* e domínios / portas nos quais os serviços estarão disponíveis. Após esse fluxo, a equipa realiza testes funcionais para verificar a integração adequada do sistema.

Ainda que efetuar o *deploy* manualmente ofereça maior flexibilidade, apresenta desafios operacionais, sobretudo pelo tempo necessário à sua execução e pela sobrecarga que impõe sobre uma equipa já reduzida, limitando o número de recursos disponíveis para a realização de outras atividades essenciais. O *deploy* é, normalmente, realizado por apenas um a dois elementos da equipa, o que o torna mais propenso a falhas como erros de configuração,

omissão de *scripts* ou incompatibilidades nas configurações dos servidores dos clientes. Isto não só aumenta o tempo necessário à instalação, como requer esforço adicional na resolução dos problemas que surgem durante esta. Consequentemente, este processo fomenta o risco de inconsistências nos *deploys* comprometendo tanto a escalabilidade quanto a qualidade das entregas finais. O estudo [3] reforça a relevância deste desafio, enfatizando que *deploys* manuais frequentes de múltiplos serviços em arquiteturas de microsserviços podem acarretar uma sobrecarga operacional significativa para as equipas de desenvolvimento e manutenção.

A presente dissertação constitui a oportunidade de otimizar e automatizar o processo de *deploy* do PlexHub, com o intuito de reduzir erros, diminuir o tempo necessário para instalação e assegurar uma integração mais consistente.

1.2 Motivações, Benefícios e Restrições

O processo de *deployment* é uma tarefa interna realizada com relativa frequência (aproximadamente a cada duas semanas). Atualmente, envolve um extenso conjunto de passos manuais que impactam significativamente a carga de trabalho da equipa e aumentam a propensão a erros humanos, como configurações incorretas das portas, conexões às bases de dados ou gestão do próprio *Internet Information Services (IIS) Manager*. Em casos extremos, esses erros ocasionam falhas na integração, que se tornam obstáculos a uma instalação confiável. Neste contexto, o presente estudo surge da necessidade de otimizar o tempo despendido na tarefa, bem como de aprimorar a infraestrutura do projeto, visando atender às necessidades dos clientes de forma mais eficiente. Este trabalho possui relevância académica pois conecta conceitos teóricos à sua aplicabilidade prática em cenários reais. Além disso, oferece *insights* sobre a adaptabilidade de metodologias modernas a organizações que enfrentam dependências *legacy* e configurações específicas por cliente. Este desafio apresenta benefícios significativos: constitui um primeiro passo para a adoção das práticas mais consolidadas no mercado, fornece documentação e uma solução que servirão como base de conhecimento para melhorias futuras, otimiza a alocação de recursos (permitindo que sejam direcionados para atividades de maior valor agregado) e, ao reduzir o tempo de *downtime*, minimiza os custos operacionais associados. No entanto, é crucial destacar um conjunto de restrições internas que orientaram tanto a pesquisa quanto o desenvolvimento da solução:

- **Base de Dados:** Pretende-se que a maioria dos serviços de *backend* tenha a sua base de dados migrada de *SQL Server* para *PostgreSQL*, com exceção de um serviço em específico (`PlexHub.Erp`) que, devido às suas múltiplas integrações com as bases de dados dos clientes, deve manter-se no mesmo ambiente de persistência, assegurando compatiblidade.
- **Proteção de dados / Segurança:** A segurança é um requisito crítico. Todas as credenciais e informações sensíveis devem ser geridas de forma a minimizar a sua exposição, privilegiando mecanismos que mantenham a responsabilidade e o controlo junto das entidades proprietárias dos dados.
- **Preferência por ferramentas open-source:** Sempre que possível, utilizar ferramentas de código aberto ou soluções baseadas em subscrições já existentes na organização.
- **Alteração de plataformas de desenvolvimento:** A organização encontra-se em processo de transição da plataforma *Azure DevOps* para *GitHub*, sendo imperativo fazer uso deste último durante o desenvolvimento.

Os desafios técnicos e as motivações foram cuidadosamente considerados ao longo da dissertação, de modo a garantir que a solução proposta é prática, eficiente e compatível com as melhores práticas da indústria.

1.3 Objetivos

O objetivo principal do trabalho é o **desenvolvimento de uma solução automatizada para o processo de entrega de software no âmbito de projetos de microsserviços em ambientes remotos, com ênfase na utilização de práticas de Continuous Integration / Continuous Delivery (CI/CD) e automação de setup de bases de dados**. De forma a atingir este propósito, foram definidos objetivos específicos:

Tabela 1.1: Objetivos secundários

Objetivo	Descrição
O1	Identificar as melhores práticas no *deployment* de microsserviços para ambientes remotos
O2	Desenvolver uma ferramenta que permita efetuar o *deployment* de microsserviços para ambientes remotos
O3	Criar um processo de automatização do versionamento e compilação de serviços
O4	Explorar técnicas de automatização de migrações de bases de dados e o seu *setup* garantindo a integridade dos dados

1.4 Considerações Éticas

A presente dissertação foi dirigida segundo princípios éticos rigorosos, de acordo com os Códigos de Conduta do Instituto Politécnico do Porto (IPP) [4], o Código de Ética *Institute of Electrical and Eletronics Engineers* (IEEE) [5] e o *Software Engineering Code of Ethics and Professional Practice da Association for Computing Machinery / Institute of Electrical and Electronics Engineers - Computer Society (ACM/IEEE-CS)* [6].

As integridades académicas e profissionais constituem um aspeto fundamental, assegurado pela citação apropriada de todas as fontes utilizadas, conforme o estabelecido no Artigo 6º do Código de Conduta do IPP e reforçado pela inclusão da Declaração de Integridade, que formaliza o compromisso com a honestidade e rigor científico no desenvolvimento da dissertação (Artigo 8º do Código de Conduta do IPP).

Considerando que o trabalho desenvolvido envolve o manuseio de informações sensíveis (como especificação de dados relevantes à conexão aos *providers* de autenticação e domínios de disponibilização destas aplicações), todas as configurações foram devidamente anonimizadas, em conformidade com o Princípio 2.05 do código ACM/IEEE-CS e o artigo I.1 do IEEE. Para fins de demonstração e validação optou-se, sempre que possível, pela utilização de configurações simuladas que, embora representativas, não comprometem a confidencialidade dos dados dos clientes.

Conforme se demonstrará nos capítulos subsequentes, uma fase fundamental do desenvolvimento envolve a seleção de ferramentas e tecnologias, esta seguiu critérios estritamente técnicos, conforme preconizado pelos princípios I.3 e I.5 do IEEE e pelo Princípio 3.04

do ACM/IEEE-CS. Para tal, foram somente consideradas a compatibilidade técnica com o projeto existente e a adequação ao ambiente de desenvolvimento, evitando influências pessoais ou institucionais. Cada decisão foi documentada e justificada de forma a garantir transparência no processo de decisão.

O compromisso com a qualidade do produto final manifesta-se através da inclusão dos mais elevados padrões profissionais, conforme estabelecido no Princípio 3 do código ACM/IEEE-CS. Este compromisso inclui a documentação meticulosa de desafios encontrados e soluções implementadas (Princípio 3.11), bem como a preservação da integridade dos dados durante todo o processo de desenvolvimento (Princípio 3.14).

Em conformidade com os princípios de transparência académica, é relevante mencionar que se fez uso de ferramentas de Inteligência Artificial para melhoramento da qualidade linguística e estrutural do texto produzido, sendo que toda a investigação, análise, desenvolvimento conceptual e conclusões são da autoria exclusiva da autora.

Por fim, em alinhamento com o Princípio 1.03 do ACM/IEEE-CS e o artigo I.1 do IEEE, o desenvolvimento deste trabalho visa contribuir para o bem público através da otimização de processos e redução de erros. A solução proposta não só corresponde aos requisitos técnicos do projeto, como também considera seu impacto na qualidade do trabalho da equipe de desenvolvimento, promovendo eficiência e confiabilidade nos processos automatizados.

1.5 Estrutura do Documento

A presente dissertação divide-se em seis capítulos. O primeiro (Introdução), no qual esta secção se insere, oferece uma visão geral do problema em estudo, bem como o que se pretende determinar uma vez que este esteja concluído. Adicionalmente, inclui uma reflexão crítica acerca das considerações éticas que impactam o trabalho desenvolvido e apresenta o planeamento efetuado para a conclusão das suas etapas.

Segue-se o capítulo de Estado da Arte, iniciado pela definição das questões e metodologia de pesquisa selecionadas e descrição do processo de *research mapping*, no qual são agregadas todas as informações derivadas da revisão sistemática da literatura. Este é finalizado com uma avaliação dos dados obtidos, servindo esta como ponte para o seguinte capítulo.

O terceiro capítulo engloba a análise do problema em maior detalhe, bem como o *design* da solução, incluindo representações gráficas quer do estado atual do sistema e do problema sob análise, quer da solução idealizada através de UML.

O quarto capítulo descreve detalhadamente todas as fases do desenvolvimento da solução, inclusivamente ajustes realizados no fluxo desenvolvido inicialmente devido ao *feedback* obtido no uso de cada protótipo.

O quinto efetua a avaliação do trabalho desenvolvido, aplicando a solução implementada a um contexto que emula os ambientes reais, bem como analisa o sucesso/insucesso desta e apresenta a resposta a todas as questões que dirigiram a presente dissertação. Este capítulo é encerrado com uma análise cuidadosa acerca dos resultados obtidos nos testes efetuados.

O sexto e último capítulo enquadra os objetivos estabelecidos e a solução desenvolvida, analisando o seu cumprimento. Inclui, ainda, uma reflexão crítica acerca da globalidade do trabalho, nomeadamente nas limitações do mesmo e trabalho futuro.

1.6 Planeamento e Metodologia de Trabalho

1.6.1 Identificação dos Stakeholders

Os *stakeholders* identificados para este projeto são:

- **Orientador:** Responsável pera orientação académica
- **Supervisor:** Representante da organização PlexIt, fornece o contexto técnico e requisitos
- **Equipa de desenvolvimento PlexIt:** São os principais interessados, dado que são eles que utilizarão a solução
- **Clientes:** Utilizadores finais que beneficiarão indiretamente da eficiência do processo de *deployment*

1.6.2 Cronograma e Metodologia

O desenvolvimento da dissertação seguiu uma abordagem incremental e iterativa, que se dividiu em 7 tarefas fundamentais:

Tarefa 1: Planeamento e Preparação

- Definição dos objetivos e âmbito do projeto
- Identificação das questões éticas e estabelecimento de diretrizes
- Análise preliminar dos requisitos técnicos e organizacionais

Tarefa 2: Estado da Arte

- Revisão sistemática da literatura utilizando metodologia SMS (Systematic Mapping Study)
- Definição das questões de investigação
- Análise comparativa de soluções existentes e identificação de lacunas

Tarefa 3: Análise do Problema

- Análise detalhada do problema atual
- Levantamento de requisitos funcionais e técnicos

Tarefa 4: Design da Solução

- Conceção da arquitetura da solução proposta

Tarefa 5: Implementação

- Desenvolvimento da solução
- Testes e refinamentos iterativos na própria máquina

Tarefa 6: Avaliação da Solução

- Aplicação da solução a um ambiente de testes controlado
- Análise da solução implementada e resultados obtidos
- Reflexão crítica sobre a globalidade do trabalho

Tarefa 7: Documentação

- Desenvolvimento do documento da dissertação
- Preparação da apresentação

1.6.3 Considerações Temporais

O desenvolvimento deste projeto decorreu num período de doze meses, sendo caracterizado por uma abordagem flexível na gestão temporal devido às limitações inerentes ao exercício simultâneo de funções profissionais. A execução do trabalho foi maioritariamente realizada em horário pós-laboral e durante os fins de semana, tendo sido, apenas numa fase mais avançada, negociada a atribuição de um dia semanal dedicado exclusivamente à investigação.

Esta distribuição temporal permitiu uma maturação gradual das ideias e uma reflexão aprofundada sobre os diversos aspetos do projeto, contribuindo para a solidez do trabalho desenvolvido.

1.6.4 Milestones Principais

Os principais marcos do projeto incluíram:

- **Milestone 1** - Aprovação do planeamento e definição do âmbito
- **Milestone 2** - Conclusão da revisão da literatura e identificação da solução conceptual
- **Milestone 3** - Implementação da solução finalizada
- **Milestone 4** - Validação da solução em ambiente controlado
- **Milestone 5**- Finalização da documentação e avaliação dos resultados

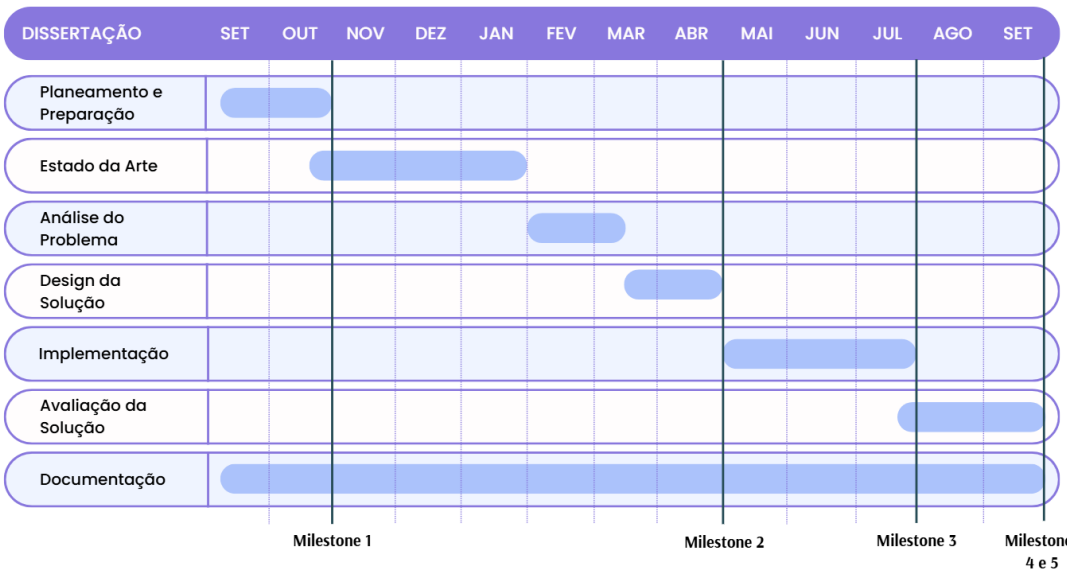


Figura 1.1: Cronograma de Tarefas

Capítulo 2

Estado da Arte

Este capítulo apresenta a metodologia da revisão sistemática da literatura, iniciando com a questão de pesquisa que norteou o estudo.

Em seguida, introduz os conceitos fundamentais necessários para a compreensão do trabalho desenvolvido, estabelecendo as bases teóricas para a discussão subsequente.

O capítulo prossegue com uma análise detalhada das práticas e ferramentas contemporâneas utilizadas na resolução do problema em questão, avaliando a sua eficácia e adequação ao contexto específico da pesquisa.

2.1 Metodologia de Pesquisa

De forma a efetuar uma revisão sistemática da literatura e garantir a compreensão das informações expostas neste capítulo, foi adotada uma metodologia de *Systematic Mapping Study* (SMS). Um SMS tem como objetivo principal efetuar um estudo imparcial acerca do estado da arte de um determinado tópico de pesquisa, identificando potenciais lacunas e tendências [7]. Este processo resulta numa agregação classificada de publicações que permite visualizar a estrutura da literatura existente e o estado atual do campo de pesquisa em relação às questões definidas. De acordo com [7], estabeleceu-se um SMS através das seguintes fases:

1. Planeamento
    - Definição das questões de pesquisa
    - Definição do âmbito
    - Estabelecimento da estratégia de pesquisa
    - Determinação dos critérios de seleção (inclusão/exclusão)
2. Execução da Pesquisa
    - Pesquisa nas fontes selecionadas utilizando as strings definidas
3. Seleção dos Estudos Primários
    - Aplicação dos critérios de seleção
    - Análise detalhada de títulos, resumos, introduções e conclusões para determinar relevância
4. Extração e Classificação

- Desenvolvimento do esquema de classificação
- Extração dos dados relevantes
- Categorização dos estudos

5. Análise e Mapeamento
    - Extração de estatísticas
    - Construção de mapas visuais
    - Documentação através do protocolo PRISMA

Este processo foi guiado pela *framework Population, Intervention, Comparison, Outcomes, Context and Study Design* (PICOCS), aplicada durante a fase de planeamento, e pelo protocolo *Preferred Reporting Items for Systematic reviews and Meta-Analyses* (PRISMA), utilizado na etapa de análise e mapeamento. Essas diretrizes, amplamente reconhecidas em revisões sistemáticas na área de engenharia de software, garantem rigor e transparência na identificação, seleção e análise de estudos empíricos relevantes para responder à questão de investigação.

2.1.1 Planeamento

O primeiro passo foi definir a questão de investigação para a revisão sistemática da literatura:

- (**QI**) Que estudos empíricos existem sobre práticas de *Continuous Integration / Continuous Delivery* (CI/CD) automatizadas para *deployment* de microsserviços em ambientes remotos?

Seguiu-se o enquadramento desta com a estrutura PICOCS:

QI - Que estudos empíricos existem sobre práticas de CI/CD automatizadas para deployment de microsserviços em ambientes remotos?

Tabela 2.1: PICOCS - Enquadramento da questão na *framework* e critérios de seleção

PICOCS	Excerto QI	Critérios Inclusão/Exclusão
Population (P)	Estudos sobre microsserviços	E: Antes de 2023 I: Estudos empíricos
Intervention (I)	Práticas de CI/CD automatizados	I: Uso de CI/CD em microsserviços I: Comparações entre abordagens automáticas e manuais
Comparison (C)	-	-
Outcomes (O)	Deployment em ambientes remotos	E: Deployments distintos que não forneçam informações sobre CI/CD
Context (C)	Ambiente profissional	I: Apenas profissionais
Study Design (S)	Estudos empíricos	E: Literatura cinzenta

Tabela 2.2: PICOCS - *Substrings* de Pesquisa

PICOCS	Substring
Population	"microservices*"
Intervention	"CI/CD", "continuous integration", "continuous delivery"
Comparison	-
Outcomes	"remote deployment", "automated deployment", "remote environment deployment"
Context	"industry", "enterprise"
Study Design	"empirical", "case study", "experience report"

Identificadas as *substrings* e após a construção de diversas *queries*, optou-se pelo uso da seguinte:

microservice*
AND
("continuous integration"**OR** "continuous delivery"**OR** "CI/CD")
AND
("remote deployment"**OR** "remote environment deployment"**OR** "automated deployment")

Figura 2.1: *Query* de Pesquisa

Bases de Dados

Procedeu-se, por fim, à escolha das bases de dados a utilizar:

1. ACM Digital Library
2. Science Direct
3. Web of Science

2.1.2 Execução da Pesquisa

Denote-se que no âmbito da *ACM Digital Library* não foram considerados os registos relativos a *Proceedings* pois estes agregavam em si um número avultado de estudos que, por estarem embebidos num único documento, não poderiam ser devidamente filtrados em fase posterior, nomeadamente no critério de duplicação. Na tabela abaixo apresenta-se o número de registos encontrados por base de dados:

Tabela 2.3: Resultados da execução da 2.1

Base de Dados	**Número de Resultados**
ACM Digital Library	55
Science Direct	61
Web of Science	2

2.1.3 Seleção dos Estudos Primários

O primeiro passo para a seleção dos estudos primários consistiu na aplicação dos critérios de exclusão previamente definidos sobre estes. Após limitação por filtros de datas, terá resultado que:

Tabela 2.4: Resultados de filtros por data 2.3

Base de Dados	**Número de Resultados**
ACM Digital Library	45
Science Direct	28
Web of Science	0

Procedeu-se depois à remoção dos duplicados entre as diferentes bases de dados e, por fim, removeram-se todos os estudos que, após leitura do título e resumo, não se enquadravam no contexto da pesquisa. Tendo-se que, dos **73 documentos identificados**, **apenas 25 permaneceram**.

2.1.4 Extração e Classificação

Da questão de investigação resultaram as seguintes categorias:

- C01 - Táticas de *Deployment* atuais: Engloba principalmente estudos de caso que demonstram detalhes na implementação de mecanismos de automatização de *deployments*
- C02 - Ferramentas de *Deployment*: Inclui as diversas tecnologias existentes como Docker, Kubernets, ferramentas de IaC, entre outros
- C03 - Melhores / Piores Práticas de DevOps e CI/CD - Práticas culturais e de colaboração que afetam o deployment automatizado, métricas de sucesso, falhas comuns, automatizações eficazes, etc.

2.1.5 Análise e Mapeamento

Abaixo apresenta-se o diagrama PRISMA, gerado através da ferramenta *online* oficial [8]:

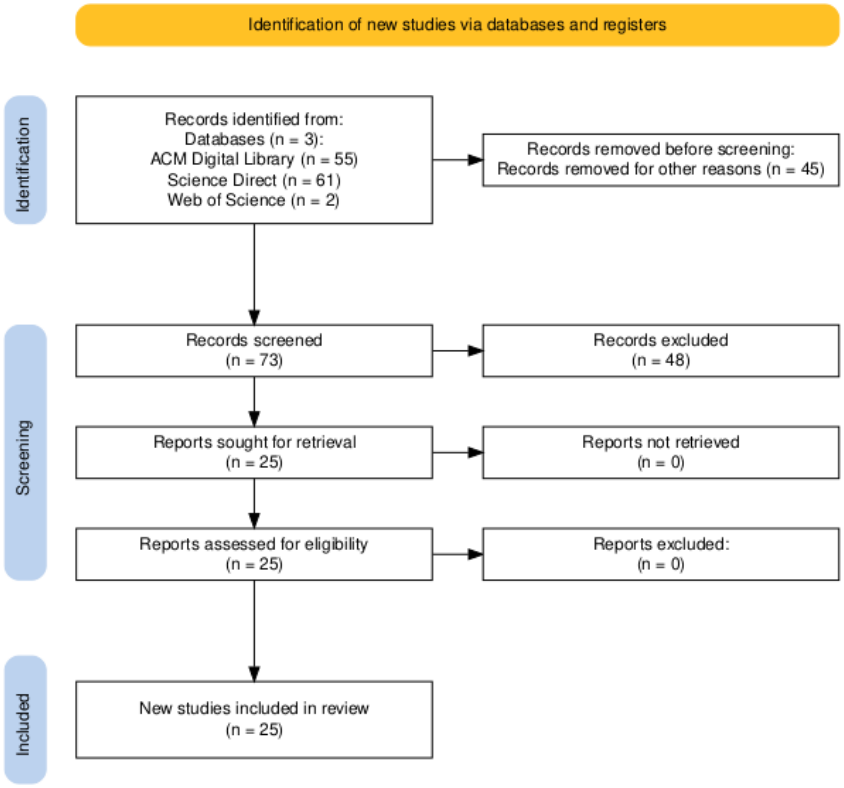


Figura 2.2: Diagrama PRISMA

Conforme se demonstra na figura acima, dos 176 artigos retornados pela execução da *query* de pesquisa, 45 foram removidos após a aplicação dos filtros definidos em 2.1.1. Dos 73 estudos que avançaram para a fase de *Screening* 48 foram removidos após a leitura do *abstract*. Deste processo resultaram os 25 estudos incluídos na presente revisão da literatura. [1]

2.2 DevOps e CI/CD

No mercado atual de software, cada vez mais competitivo e exigente, surgiu a necessidade de realizar entregas mais frequentes e desenvolver sistemas mais complexos. O DevOps surge como potencial solução para os desafios inerentes ao desenvolvimento moderno de software. Embora não exista uma definição definitiva para o conceito de DevOps, este pode ser entendido como uma abordagem que enfatiza comunicação e cooperação, eliminando barreiras entre equipas técnicas e operacionais. Tem por objetivo acelerar o processo de disponibilização de software através de automatizações na criação de novo software (sem comprometer a qualidade). Este pode ser, ainda, definido como uma *framework* concetual assente nos princípios CALMS (Cultura, Automatização, Princípios LEAN, Medição e Partilha). Incorpora também estratégias de Engenharia de Software que favorecem a adaptação

[1] Foram ainda consultados diversos websites oficiais e de referência que disponibilizam documentação especializada sobre cada ferramenta. Estas fontes, por se destinarem essencialmente ao suporte técnico da implementação, não foram incluídas na revisão sistemática da literatura, mas sim utilizadas como apoio prático durante o desenvolvimento da solução. Entre estas, destacam-se a documentação oficial do Docker[9], do Docker Compose[10] e do GitHub Actions[11].

rápida a mudanças nos requisitos e no design. O objetivo principal deste é proporcionar entregas de software mais rápidas, em ciclos de desenvolvimento mais curtos. Um dos pilares fundamentais do DevOps é precisamente o CI/CD, tornando a automatização de processos essencial para o sucesso das equipas. Esta automatização não só assegura a qualidade do software mais rapidamente, como também reduz os tempos de produção e garante que as necessidades dos *stakeholders* sejam atendidas oportunamente. [12–15]

Conforme supramencionado, estabelece-se uma estreita relação entre os conceitos de DevOps e CI/CD, ambos convergindo para o mesmo objetivo. É importante, contudo, detalhar o funcionamento deste último. No intuito de se alcançar código mais seguro e de maior qualidade, privilegiando a otimização do *time to market* do produto, prevê-se que o código esteja disponibilizado num ambiente de desenvolvimento partilhado: *Source Control Management System* (SCM). Sempre que é introduzida uma modificação no *software* é despoletado um processo de compilação. Este resulta, em última instância, num artefacto que é disponibilizado num repositório acessível. Neste ponto efetuam-se os testes necessários (CI) e a sua posterior publicação em ambiente produtivo (CD). [16] A integração frequente do código neste ambiente proporciona *feedback* mais rápido e facilita a deteção e resolução de erros, bem como reduz o tempo entre o desenvolvimento e produção, permitindo entregar novas funcionalidades com maior rapidez. [17]

As vantagens oferecidas pela adoção de uma abordagem DevOps, no entanto, podem variar significativamente. No estudo realizado por [18] foram analisadas duas equipas de desenvolvimento (compostas por 8 e 9 elementos) cujos produtos foram migrados para arquiteturas distintas. Uma migrou de uma arquitetura de microsserviços para monólito, enquanto a outra realizou o processo inverso. Tornou-se evidente que uma abordagem arquitetural modular tende a maximizar os benefícios do DevOps, promovendo sistemas com menor acoplamento, ciclos de deployment mais rápidos, testabilidade facilitada e maior modificabilidade. Esta modularidade decorre, em parte, da prática de manter cada serviço em repositórios próprios, permitindo o seu desenvolvimento e deployment de forma isolada. Observou-se também que a automatização das operações (ops) foi mais granular e intensiva na abordagem baseada em microsserviços, consequência direta da maior fragmentação dos componentes, o que, a longo prazo, favorece a escalabilidade do sistema. Contudo, esta abordagem resultou num *overhead* operacional superior quando comparada à simplicidade da estratégia monolítica. Foram, ainda, reportados desafios semelhantes aos enfrentados pela PlexIT, nomeadamente o facto de os próprios desenvolvedores acumularem responsabilidades relacionadas à componente operacional. À semelhança do reportado no artigo, também na PlexIT se denota uma tendência pela priorização de novas funcionalidades em detrimento da melhoria de processos operacionais, especialmente sob a pressão para cumprir prazos de entrega. No estudo, o ênfase na entrega rápida de novas capacidades de software levou a uma subvalorização das atividades de robustez operacional. Um exemplo concreto dessa situação foi a qualidade insatisfatória dos *playbooks Ansible* utilizados pelas equipas, cuja necessidade de aprimoramento não foi fortemente reivindicada internamente.

2.3 Ferramentas de Automatização e Frameworks para o Deployment

2.3.1 Containerization

O processo de *Containerization* consiste em encapsular uma aplicação, as suas dependências e ambiente de execução numa unidade auto-suficiente. [19]

No estudo [20] apresentam-se alguns dos padrões mais recorrentes no uso de múltiplos *containers*, voltado para o contexto de Docker Compose. Destes, destacam-se:

Tabela 2.5: Padrões de *multi-container composition*

Padrão	**Definição**
Single Service Per Container	Cada *container* executa um único processo
Sidecar	*Helper Container* que providencia mais funcionalidades como logging ou monitorização
Ambassador	*Container* que age como um *proxy* para os serviços externos, isolando configurações
Adapter	Um *container* que interpreta interfaces ou protocolos entre serviços
Initializer	Serviços utilizados temporariamente para efetuar tarefas de *setup* como migrações de bases de dados
Aggregator	Serviços que agregam os resultados ou informação de diversas fontes

Docker / Docker Compose

O Docker é um projeto *open-source* que permite virtualizar aplicações em *containers*, isto é, permite que aplicações e tudo o que estas necessitam para o seu funcionamento possa ser colocado numa caixa (*container*) facilmente criado, testado e executado em qualquer ambiente.

Todo este processo é configurado via *Dockerfile*; um ficheiro quee permite indicar o conjunto de ações necessárias para construir o *container*.

Um *container* é despoletado através de *Docker Images*. Estas, por sua vez, podem ser entendidas como um sistema de ficheiros pré-construído que inclui as bibliotecas, binários e pacotes necessários ao funcionamento de uma aplicação, podendo também conter o próprio código da aplicação e dependências adicionais. Cada imagem é construída a partir de uma imagem base e organizada em camadas de dados sobrepostas, onde cada camada representa uma modificação ou adição específica. Esta abordagem em camadas permite a reutilização eficiente de componentes e garante consistência entre diferentes ambientes de execução. Assim, sempre que uma alteração é introduzida, o *Docker* adiciona-a como uma nova camada, não refaz o *container*, o que impacta positivamente a sua eficiência. A facilidade em testar e atualizar rapidamente tornou o Docker uma das ferramentas de *containerization* mais aceites na indústria e muito ajustada a CI/CD. [19]

Em [19] estudaram-se os tempos de *build* dos *containers*, tendo sido reportado que 30% das *builds* analisadas eram lentas e que as suas durações aumentam ao longo do tempo. Esta variabilidade no tempo de execução pode resultar em custos elevados dado que os desenvolvedores têm de aguardar pela finalização da construção das imagens.

Em contrapartida, as causas da longa duração das *builds* reside quer no mau uso da ferramenta - má qualidade do *Dockerfile*, frequentes alterações neste, *Dockerfiles* com muitas instruções, não fazer uso de *.dockerignore* -, quer nas características intrínsecas dos projetos / desenvolvedores - inclusão de demasiados *third-party resources*, fraca conectividade à rede, entre outras. As consequências daí derivadas, segundo [19], são o *feedback* lento,

a interrupção do fluxo de desenvolvimento e subsequentemente a redução na eficiência do *deployment*. O estudo apresenta, ainda, um conjunto de ações que podem minimizar este desafio:

- Utilizar o .dockerignore para reduzir o contexto da *build*, excluindo ficheiros desnecessários;
- Desenvolver *containers* o mais atómicos possível, avaliando seriamente as camadas que devem conter;
- Desenvolver *Dockerfiles* com poucas instruções e organizá-las das menos frequentemente usadas para as mais, tomando partido da *build cache*.

Segundo [20] o *Docker Compose* oferece uma solução simplificada para gerir múltiplos *containers* e é vastamente utilizada para orquestração em ambientes de desenvolvimento / qualidade, contudo, gerir as dependências dos diferentes serviços e configurações específicas dos *containers* introduz complexidade. Não é muito típico o seu uso em ambientes de produção, pois não existe suporte específico para tal e não oferece potencialidades que supram as necessidades de uma orquestração mais complexa, o que muitas vezes se traduz na migração desta ferramenta para orquestradores como o Kubernetes.

2.4 Estratégias de Deployment de Microsserviços

O termo Microsserviço remete para um estilo arquitétonico que visa separar sistemas de grande escala em unidades funcionais de menor dimensão com o propósito de alcançar maior escalabilidade, disponibilidade, usabilidade, manutenabilidade e flexibilidade na automatização do *deployment*. [12, 17, 21]

Ainda que ofereça uma maior flexibilidade face à tradicional arquitetura de monólito, existem diversos desafios relativos à correta modularização dos serviços e ao próprio *deployment*, dado implicar a publicação de um número variável de serviços para ambientes cujos recursos são tipicamente limitados. [21].

De acordo com os estudos [21, 22] existem várias estratégias de *deployment* de microsserviços:

- **Serverless Deployment**
  - Utiliza *cloud service providers* como Amazon AWS Lambda, Azure, Google Cloud, etc
  - Executa os serviços sem configuração de servidor ou manutenção deste
  - Foca no desenvolvimento de *software*, sem responsabilidade sobre a infraestrutura
- **Service Instance per VM**
  - Cada serviço é tratado como uma imagem VM
  - Instâncias dos serviços são executadas independentemente e totalmente isoladas umas das outras
  - Limites de utilização de CPU e memória para não afetar outros serviços podem ser estabelecidos

  - Compatibilidade com *cloud infrastructure* que pode facilitar *load balancing* e *automated scaling*
  - Proteção contra a publicação indevida de dados sensíveis, *phishing*, etc
- **Service Instance per Container**
  - Cada serviço é uma *container image*, isto é um *container* contém apenas informações relativas a um serviço
  - Possibilidade de limitar CPU e memória
  - Gestão de *containers* via *cluster managers* como *Kubernetes* ou *Marathon*
  - Mesmas vantagens de segurança e monitoramento de recursos que VMs
  - Possibilidade de efetuar o *deployment* de múltiplos *containers* em simultâneo

2.4.1 Desafios

Segundo [21, 23] existem diversos desafios no *deployment* de soluções em microsserviços, abaixo enumerados:

- **Complexidade Arquitetural**
  - Muitos pacotes para implantar
  - Necessidade de ter em consideração as interdependências de serviços
  - Dificuldade em alocar os recursos da forma mais eficiente possível
  - Dificuldade em gerir manualmente este processo conforme o número de serviços a implantar aumenta
- **Tolerância a Falhas**
  - Ainda que quando ocorre uma falha num microsserviço tipicamente esta não se propaga para as restantes dado que cada serviço ser individualizado, podem existir dependências associadas ao serviço em falha que comprometam o normal funcionamento da aplicação
- **Coordenação de Deployment**
  - Facilidade em criar dependências cíclicas entre serviços que podem prejudicar o próprio *deployment* e o uso da aplicação
  - Em casos em que diferentes equipas se responsabilizam por diferentes serviços pode criar atrasos no *deployment*
- **Distributed Logs**
  - Aquando de uma falha pode ser difícil compilar e analisar todos os logs dos serviços de forma a detetar a causa desta
- **Vulnerabilidade da Imagem do Container**
  - Os mecanismos de *Black box reuse* utilizados por *containers* permitem *deployments* prontos a usar, contudo, são riscos de segurança pois não existe forma de assegurar que a informação transportada é autêntica

- **Configurações Específicas Necessárias**
  - Dependendo do ambiente para o qual se efetua o *deployment* podem ser necessárias configurações adicionais específicas
  - Sempre que o algum dos serviços é atualizado é muito fácil quebrar dependências externas, sendo necessário gerir cuidadosamente o versionamento
- **CI/CD**
  - Dificuldade em diminuir o *downtime* e esforço necessário à atualização ou implantação de uma solução baseada em microsserviços
  - Necessidade de desenvolver *pipelines* por serviço (o que pode ser custoso temporalmente)

2.5 Casos de Estudo

Nesta secção são avaliados alguns casos de estudo que descrevem a aplicação de uma abordagem *DevOps* focada na automatização de processos que envolvem o ciclo de vida de aplicações.

O estudo [17] apresenta uma pipeline CI/CD para uma aplicação desenvolvida numa arquitetura de microsserviços, aproveitando os recursos do *Microsoft Azure* e *DevOps*. A *pipeline* proposta divide-se em etapas sucessivas de testes até ao *deployment* para o ambiente produtivo. O processo é iniciado aquando da disponibilização do código no repositório *Azure*, a partir do qual segue para a fase de compilação e testagem (testes unitários e análise de *bugs* e vulnerabilidades). Posteriormente, procede-se à construção dos artefactos que, no contexto, consiste na criação de uma imagem e a sua publicação no *Azure Container Registries*. A imagem é, então, implantada no ambiente *Quality Assurance* (QA) onde é novamente testada. Finalizado este processo, é efetuada a implantação no ambiente *cloud*.

O trabalho apresentado em [24] foca-se na fase de *continuous integration* de uma aplicação de grande escala, cujo desenvolvimento foi organizado em três *pipelines*. Esta divisão justificou-se pelo facto de a equipa de desenvolvimento utilizar uma estratégia de *multi-branch*, que tornaria temporalmente custoso desencadear uma pipeline por cada *commit*. A primeira etapa consiste na validação da qualidade do código durante a criação de um *merge request*. Após aprovação, decorre a denominada *daily integration stage*, que consiste em efetuar o *build* e compilar o código, integrando-o no *branch* principal. Novos testes são então executados. O último nível é responsável por realizar testes de interação entre sistemas (como testes *End to End*).

No artigo [25], é apresentada a pipeline *MIRA* orientada a *deployments* para ambientes *cloud* utilizando *containers*. Segundo descrito, a solução *MIRA* é capaz de efetuar as fases de *build*, testagem, *release* e *deployment* para uma plataforma de orquestração de *containers*, abstraindo toda a lógica necessária para a construção manual de imagens *Docker*. Para tal, disponibiliza um portal onde o utilizador pode inserir os dados específicos do seu projeto para iniciar o *deployment*. Esta solução reduz significativamente a complexidade técnica e o tempo de *deployment*, bem como minimiza os erros humanos deste processo, sendo especialmente eficaz para equipas com recursos limitados.

No seguimento das soluções expostas destaca-se o artigo [26]. O projeto descrito depende de *Merge Request(s)* (MR) para manter a qualidade do código e o *branch main* pronto para

produção. Assim, sempre que um desenvolvedor disponibiliza código aprovado pela *pipeline* de desenvolvimento, um MR é criado para avaliação pelos colegas. Adicionalmente, utilizam *git workflows* (modelos que definem como se introduzem, avaliam e integram mudanças num repositório Git) para gerir as *pipelines* de CI/CD dos microsserviços. Desta forma, dividem a sua pipeline de CI em duas fases; primeiro, avaliam a qualidade do código previamente à revisão (efetuando a compilação, análise e os testes unitários sempre que o desenvolvedor realiza um *commit* ou *push*); a segunda fase (*build phase*) corresponde à criação da *docker image* e sua publicação no registo, sendo despoletada pela aprovação do MR. Para a pipeline de CD, estabelecem dois tipos: *Functional System Deployment Pipeline*, que efetua o deployment integral do sistema de microsserviços (ainda não implementada na *framework* DevOps) e *Microservice Deployment Pipeline* responsável pelo deployment individualizado dos serviços. Para esta última, é apenas necessário aceder ao artefacto armazenado no registo e disponibilizá-lo nos ambientes necessários.

Em contraste com as abordagens anteriores, o trabalho descrito em [27] apresenta uma metodologia diferenciada baseada em *Infrastructure as Code* (IaC) (toma partido de código *machine readable* para gerir sistemas de forma autónoma, aplicando práticas consolidadas de desenvolvimento como controlo de versões, análises estáticas, revisão de código e, por fim, o próprio *deployment*). Os autores propõem o uso de *muse*, uma ferramenta de IaC descentralizado, em oposição a diversas soluções existentes no mercado (como Terraform, AWS CloudFormation, Azure Resource Manager, etc.). O estudo argumenta que as soluções IaC convencionais contradizem o princípio DevOps de autonomia das equipas, pois frequentemente as aplicações apresentam interdependências que exigem coordenação de *timing* e sequência de *deployment*. Em contrapartida, o *muse* concebe os *deployments* como um processo contínuo e reativo: uma vez identificadas as dependências, estabelece-se um sistema onde, se A depende de B e A é alterado, automaticamente se desencadeia a atualização de B. Os resultados demonstraram que a ferramenta permite o *deployment* independente entre equipas diferentes e garante que o *deploy* dos recursos só é efetuado se as suas dependências estiverem configuradas e vice-versa.

Numa perspectiva orientada à unificação das ferramentas de *deployment*, o estudo [22] começa por destacar o risco elevado resultante de um *deployment* manual de uma arquitetura de microsserviços, quer pelo tempo necessário, quer pela quantidade de erros que pode gerar. Prossegue para o uso de imagens Docker como forma de reduzir o tempo necessário na execução deste processo e a posterior adição de Kubernetes para garantir a escalabilidade e automatização. Contudo, face à dificuldade de gerir todas as ferramentas distintas necessárias, propõem o uso de uma ferramenta unificadora: *OpenShift*. Esta é configurada para executar o *build*, a testagem, o *deployment* e a monitorização da aplicação. Por fim, efetuam uma comparação com as ferramentas *Kubernetes* e *Docker Swarm*, concluindo que, à medida que a complexidade arquitetural aumenta, também o tempo de *deployment* aumenta, exceto quando se utiliza a ferramenta.

O estudo [28] relativo ao software desenvolvido no CERN descreve uma migração na metodologia de deployment dos serviços. A primeira abordagem baseava-se na implantação de *virtual machines* na infraestrutura *cloud* (CMSWEB *cluster*). Esta metodologia, contudo, dependia fortemente da interação entre as diferentes equipas de desenvolvimento e os operadores do *cluster*, sendo que as incompatibilidades horárias, bem como a compartimentalização da implementação dos serviços, dificultavam o processo, o que resultava em implantações com periodicidade apenas mensal. Para reduzir a latência produzida pelas inúmeras interações e entregar os serviços atualizados mais rapidamente aos desenvolvedores,

bem como reduzir os ciclos de *deployment*, os autores optaram por migrar para *Docker* e *Kubernetes*. Nesta nova abordagem, dividiram o *cluster* existente em dois: um responsável pelo frontend e outro pelo backend, mantendo em ambos a utilização do *Nginx ingress controller*. Adicionalmente, implementaram mecanismos de replicação e *horizontal pod autoscaling* para métricas específicas, e, dado que o Kubernetes nativamente suporta apenas métricas baseadas em RAM e CPU, utilizaram o *Prometheus* para expandir essas capacidades. Inicialmente, o processo de CI/CD definido incluía diversos passos manuais, pelo que optaram por automatizar todo o processo utilizando o *GitHub Action workflow*. No entanto, fizeram-no apenas para o ambiente de desenvolvimento nesta fase. De acordo com o reportado, estas mudanças proporcionaram uma melhor performance quando comparada ao uso de *virtual machines* e simplificaram o dia a dia dos desenvolvedores, os quais se tornaram capazes de efetuar o *deployment* sem a necessária intervenção dos operadores do *cluster*.

O estudo [23] assume especial preponderância pois, não focando nas tecnicidades de *frameworks*, ferramentas e tecnologias que suportam o *deployment* de soluções, enquadra o presente estudo num âmbito muito mais profundo. Estuda três empresas que procuraram transitar de uma metodologia de desenvolvimento e implantação manual ou semi-automática para uma totalmente automatizada e expõe a complexidade de uma transição que envolve muito mais do que apenas novas ferramentas, requer uma mudança organizacional e cultural e é por isso um processo que se prolonga no tempo. Uma característica de relevo incide também nas arquiteturas das soluções sob análise destas empresas que ou eram já baseadas em microsserviços ou estavam em transição para e por isso enfatizaram os desafios de inclusão desta arquitetura numa metodologia de CI/CD como a necessidade de descrever múltiplas *pipelines*, a coordenação de equipas necessária e a dificuldade de manutenção de consistência entre os serviços.

2.5.1 Comparação e Análise dos Casos de Estudo

Tabela 2.6: Tabela Comparativa dos Casos de Estudo

Estudo	Contexto & Metodologia	Ambiente	Aspetos Relevantes
[17]	**Contexto:** App demonstrativa (NodeJS, Angular) **Metodologia:** CI/CD com Azure DevOps, IaC (Terraform), Docker/Kubernetes	• Ambiente Cloud • IaC com Terraform • Docker e Kubernetes	• Forte integração com CI • Testes e validações automáticas • Auto-*scaling* e monitorização contínua • **Aplicabilidade:** Complexidade elevada para o contexto sobre estudo

Continua na página seguinte...

Tabela 2.6 – continuação da página anterior

Estudo	Contexto & Metodologia	Ambiente	Aspetos Relevantes
[24]	**Contexto:** Projeto de grande escala **Metodologia:** Não especificada (foco em CI e deteção precoce)	Não especificado	• Ênfase na deteção de falhas iniciais • Integração com múltiplas pipelines • Validações regulares (diárias/semanais) • **Aplicabilidade:** Inclui insights aplicáveis ao processo atual
[25]	**Contexto:** Equipas pequenas **Metodologia:** Pipeline completa (build a deployment) com Docker / Kubernetes via Mira	• Cloud (Crane Cloud Platform)	• Foco em ambientes similares ao estudo • Solução robusta para *Cloud* • **Aplicabilidade:** Incompatibilidade com linguagens do presente problema
[26]	**Contexto:** Ferramenta JuNo-Ops **Metodologia:** CI/CD com Docker e Docker Hub	• Ambiente Cloud	• Explora Pipeline orquestradora do deployment dos microsserviços • Automatização da build • Containerization e Orchestration • **Aplicabilidade:** Semelhante ao pretendido na resolução do problema atual
[27]	**Contexto:** Equipa DevOps independente **Metodologia:** IaC descentralizado (Muse)	Não especificado	• *Deployment* independente reativo • Deteção de dependências entre serviços • **Aplicabilidade:** Complexidade excessiva para uma única equipa

Continua na página seguinte...

Tabela 2.6 – continuação da página anterior

Estudo	Contexto & Metodologia	Ambiente	Aspetos Relevantes
[22]	**Contexto:** Ferramenta unificadora de deployment **Metodologia:** OpenShift baseado em containerization	• Servidor de Produção	• Benefícios do containerization confirmados • Gestão de ferramentas complexa • **Aplicabilidade:** Foco excessivo em cloud e requer subscrição (não aplicável)
[28]	**Contexto:** Migração de infraestrutura (CERN) **Metodologia:** Docker, Kubernetes, GitHub Actions e Prometheus	• Cluster interno	• Redução de dependência humana • Automatização com ferramentas modernas • **Aplicabilidade:** Desafio similar ao do estudo
[23]	**Contexto:** Transição de deployment manual para automático **Metodologia:** Não especificada	Não especificado	• Contexto técnico e cultural semelhante • Reforça o caráter de longo prazo da adoção de CI/CD

Da análise das diversas abordagens descritas nos artigos incluídos na revisão sistemática da literatura, observa-se uma preferência pelo *deployment* para ambientes *cloud*, em detrimento de *ambientes* remotos, como é o caso do problema do PlexHub.

A construção das *pipelines* segue uma estrutura relativamente padronizada entre os diversos métodos de *deployment* e arquiteturas: inicia-se tipicamente com a introdução de alterações no repositório, seguida de uma série de validações automatizadas baseadas nos testes configurados, culminando numa fase de *deployment* para ambiente de qualidade ou diretamente para produção.

Na situação específica da PlexIT, o início da pipeline partilharia o mesmo *trigger*, porém filtrado para o *branch main*. Esta abordagem alinha-se com a estratégia de desenvolvimento baseada em múltiplos *branches* que, após aprovação, são integrados no *branch develop*. As integrações no *main* são mais pontuais, sempre associadas à efetuação de uma *release* ou *hotfix*.

O estudo de [26], embora não diretamente relacionado com os objetivos desta dissertação, apresenta uma abordagem relevante para trabalho futuro. Os autores propõem a implementação de um fluxo de *Continuous Integration* que, adaptado ao contexto da PlexIt, poderia reduzir significativamente a carga de trabalho associada à validação de *pull requests* e eliminar as validações inadequadas que atualmente ocorrem durante o processo de instalação.

Destaca-se, igualmente, uma clara preferência pelo uso de *containers*, especialmente quando integrados em arquiteturas de microsserviços [24, 25]. Esta tendência justifica-se pelos benefícios em termos de eficiência de recursos de *hardware*, portabilidade e pelo controlo rigoroso das dependências externas dos diferentes serviços. No contexto específico do PlexHub, as configurações personalizadas por cliente representam um desafio particular para a implementação de *pipelines* CI/CD. Uma abordagem híbrida que combine a padronização do ambiente através de *containers* com mecanismos de parametrização dinâmica para personalização surge como a solução mais adequada às necessidades específicas da PlexIT.

A adoção de uma abordagem automatizada e baseada em *containers* apresenta-se como particularmente relevante para resolver os desafios identificados na presente dissertação:

- **Inconsistência nas Configurações:** Resolvida através da padronização por *templates*
- **Recursos Humanos Limitados:** Otimizado pela introdução de tarefas automatizadas
- **Complexidade arquitetural:** Abstração da complexidade operacional através da automatização

Soluções como o MIRA, que abstraem a complexidade da criação de *containers*, poderiam ser adaptadas para automatizar as configurações específicas de cada cliente. Esta abordagem minimizaria os erros manuais e reduziria significativamente o tempo necessário para a instalação, aspeto crítico quando se trata de configurações específicas por cliente, como enfatizado em [22].

Considerando as restrições de segurança dos dados privados dos clientes mencionadas no Capítulo 1, a solução ideal deve contemplar a automatização de configurações através de templates parametrizáveis ajustáveis para cada cliente. Adicionalmente, a integração de ferramentas de monitorização *pós-deployment*, conforme sugerido em [22], permitiria identificar precocemente potenciais problemas de integração, possibilitando uma resposta rápida pela equipa de desenvolvimento já reduzida da PlexIT.

Por fim, destaca-se o estudo de [23] que demonstrou a natureza iterativa e prolongada da adoção de metodologias DevOps, especificamente dos mecanismos de CI/CD. Esta investigação enquadra o presente trabalho como parte de um processo de transformação organizacional de longo prazo, em oposição a uma implementação pontual, uma vez que envolve o planeamento estratégico de ferramentas, a mudança de hábitos dos desenvolvedores, a reestruturação de rotinas organizacionais e o estabelecimento de mecanismos de *feedback* eficazes. Esta abordagem gradual e sustentada permitirá desenvolver uma solução mais robusta e adaptável, capaz de evoluir em sintonia com as necessidades da aplicação que suporta.

Capítulo 3

Análise e Design

3.1 Análise do Problema

Conforme explicitado nas secções anteriores, o PlexHub é um projeto modular baseado numa arquitetura de microsserviços. Existem, até ao momento, dez serviços e uma aplicação monólito que atua como interface visual de interação com o sistema. Este modelo arquitetural, embora fomente maior flexibilidade e escalabilidade, introduz também complexidade na entrega e manutenção da aplicação em ambientes remotos, particularmente quando os clientes possuem requisitos e configurações específicas. A entrega da solução aos clientes é, atualmente, executada manualmente, o que compromete não só o tempo de atualização/instalação, como a eficiência operacional da equipa de desenvolvimento. Para uma mais simplificada visualização do processo, desenvolveram-se 3 fluxogramas. O primeiro, abaixo apresentado, indica os passos iniciais e comuns tanto ao fluxo de atualização (mais frequente) quanto ao de instalação:

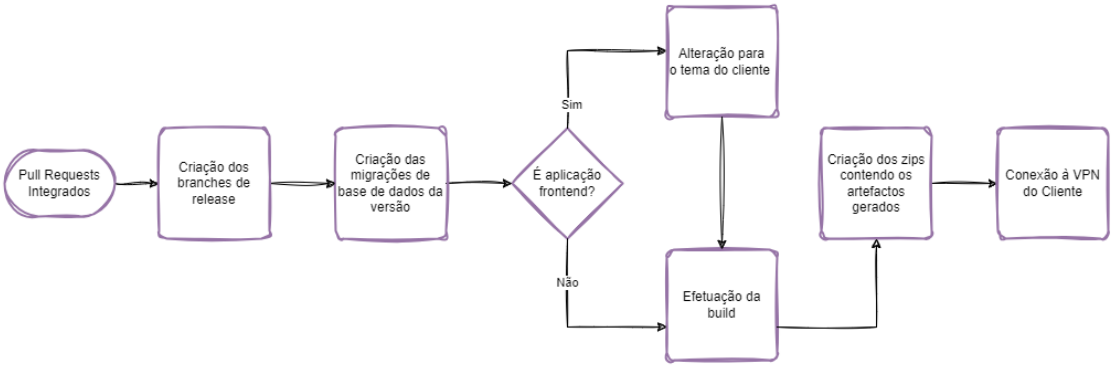


Figura 3.1: Passos iniciais à Instalação / Atualização do Serviço no Ambiente do Cliente

Criados os artefactos necessários, apresenta-se de seguida o fluxo de atualização:

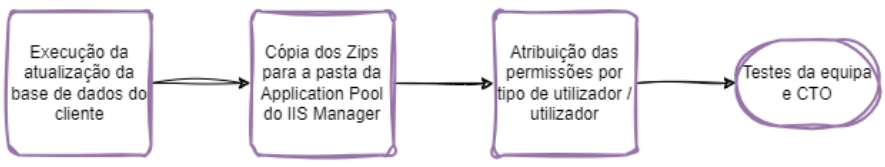


Figura 3.2: Atualização do Serviço no Ambiente do Cliente

De enfatizar que, no caso da aplicação de *frontend* (`PlexHub`) é necessário recorrer a um passo adicional. Esta aplicação foi desenvolvida a partir de um dos temas do *Metronic UI* [29] e ainda que o design da aplicação seja comum a todos os clientes, cada qual configura o seu próprio esquema de cores. Previamente à efetuação da build deste projeto recorre-se à substituição manual dos valores das variáveis de CSS no ficheiro styles.scss do tema (cada conjunto de valores encontra-se já armazenado por cliente).

Em casos de primeira instalação do serviço são necessários passos de configuração adicionais, conforme o fluxograma:

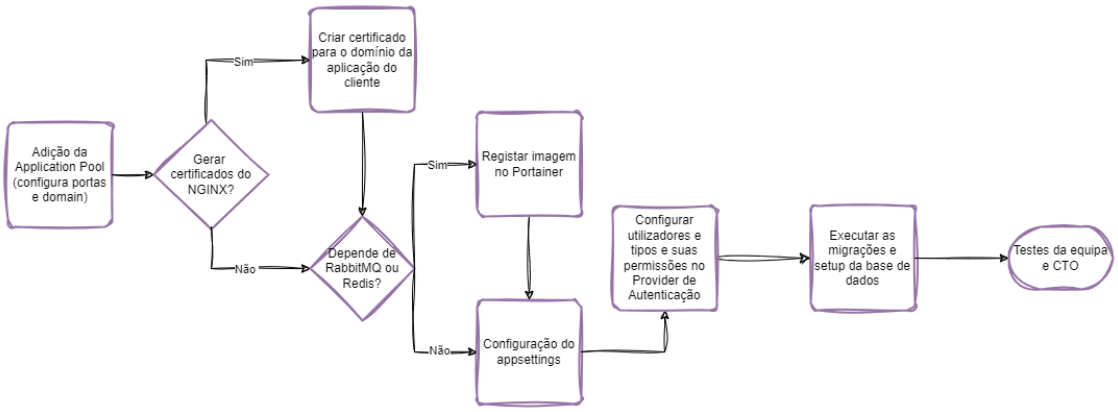


Figura 3.3: Instalação de cada Serviço no Ambiente do Cliente

O conjunto de passos a executar é significativo e trabalhoso, tornando-se suscetível a falhas humanas como omissões de *scripts* ou configurações incorretas, especialmente considerando que este é normalmente conduzido por apenas um ou dois membros da equipa, o que gera sobrecarga de trabalho e reduz a capacidade da equipa de se focar em atividades de maior valor, como o desenvolvimento de novas funcionalidades ou resolução de *bugs*. De destacar ainda a necessidade de configurações específicas por cliente como:

- Tema
- Conexão à base de dados (*SQL Server*, *PostgreSQL* e nalguns casos - `PlexHub.Erp` e `PlexHub.Reporting` - uma base de dados de *caching Redis* [30])
- *RabbitMQ*
- *Provider* de Auntenticação
- VPN

Face a este cenário, torna-se evidente a necessidade urgente de implementar uma solução automatizada que padronize o processo de *deployment*, minimize o risco de erro humano e liberte recursos da equipa para tarefas de desenvolvimento. A implementação de uma pipeline CI/CD representa, assim, não apenas uma otimização técnica, mas uma necessidade estratégica para a sustentabilidade operacional da *PlexIT*.

3.1.1 Complexidade e Crescimento da Solução

A complexidade do sistema está em crescimento constante. A comunicação entre serviços é recorrente, envolvendo mecanismos como *Advanced Message Queuing Protocol* (AMQP)

(via *RabbitMQ*), autenticação centralizada, gestão de permissões e a utilização de um *API Gateway* (`PlexHub.Gateway`) com configuração via *Ocelot*. Adicionalmente, versões específicas de serviços por cliente foram já desenvolvidas, o que aumenta ainda mais a dificuldade em garantir consistência e padronização nos *deployments*.

Estes fatores realçam a necessidade de uma solução automatizada, modular e segura, capaz de mitigar o risco de erros, reduzir o tempo de instalação e permitir uma manutenção evolutiva mais eficiente.

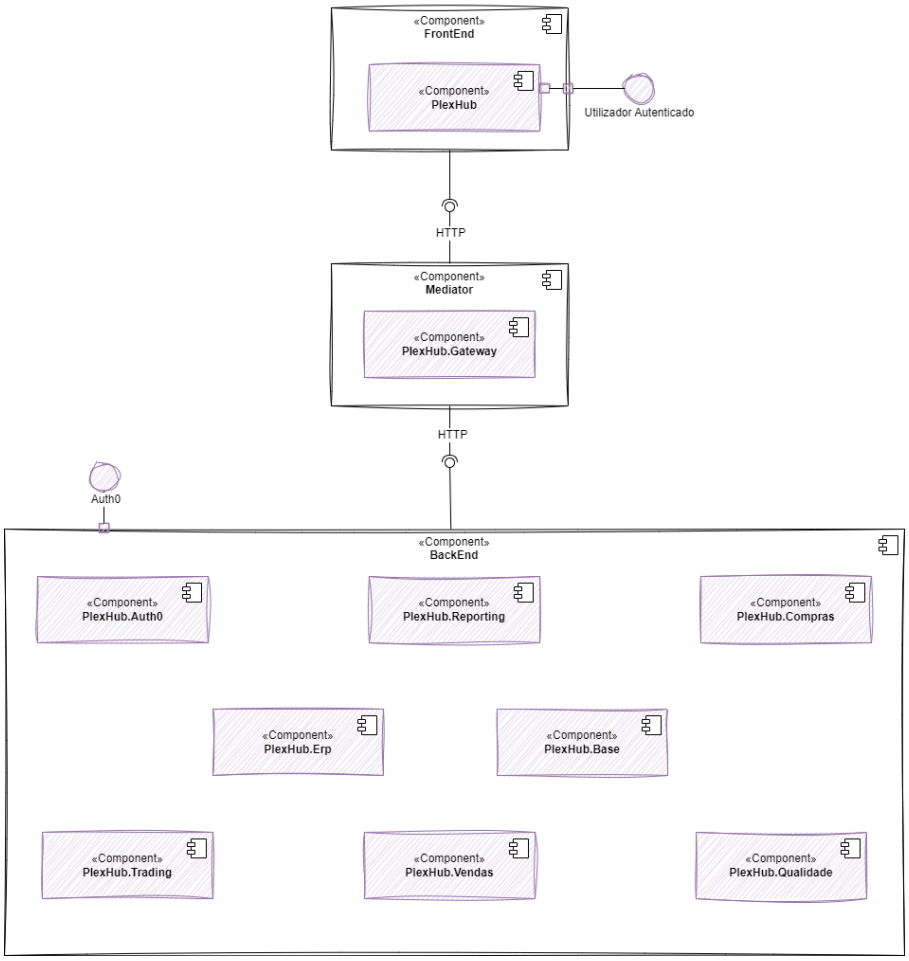


Figura 3.4: Vista Lógica de Nível 2 do Sistema

Algumas comunicações foram omitidas por via a simplificar a visualização:

- Todos os serviços contêm referência ao *provider* de autenticação para garantir segurança nos acessos às rotas e gerir as permissões associadas às diferentes funcionalidades
- Existem diversas comunicações entre microsserviços estabelecidas via AMQP. Todos os eventos estão definidos num *package* específico desenvolvido pela equipa e incluído em todos os projetos (`PlexHub.Contracts`)
- Alguns dos serviços representados têm diferentes versões consoante o cliente, utilizadas para adicionar funcionalidades específicas em conjunto com as *standard*

- Foi omitido um serviço que funciona somente como *Script Runner* para atualizar as nossas bases de dados no ambiente SQL dos clientes e/ou as dos clientes

3.1.2 Requisitos de Implementação

A definição dos requisitos é preponderante para garantir que a solução automatizada esteja alinhada com os objetivos do projeto e as restrições impostas pelo ambiente organizacional da PlexIT. A seguir são apresentados os principais requisitos identificados:

Requisitos Funcionais

Tabela 3.1: Requisitos Funcionais

Código	Descrição
RF1	O sistema deve automatizar o processo de construção, versionamento e distribuição dos componentes de software, garantindo o controlo de versões.
RF2	O sistema deve disponibilizar um mecanismo de orquestração personalizado por cliente
RF3	O sistema deve permitir a instalação e execução da solução PlexHub diretamente nos ambientes dos clientes, utilizando os componentes pré-construídos e distribuídos.
R4	O sistema deve reduzir o tempo de *downtime* nas atualização do produto.

Restrições Técnicas

Tabela 3.2: Restrições Técnicas

Código	Descrição
RT1	Compatibilidade obrigatória com ambientes dos clientes.
RT2	Preferência por ferramentas *open-source* ou já licenciadas pela PlexIT.
RT3	Uso obrigatório de *GitHub Actions* como ferramenta de integração contínua.

3.2 Design da Solução

3.2.1 Visão Geral

A solução proposta incorpora um conjunto robusto de mecanismos de automatização, mas distingue-se, sobretudo, por introduzir uma mudança de paradigma alinhada com princípios DevOps. Esta abordagem promove uma integração mais fluida entre desenvolvimento e operações, privilegiando práticas de entrega contínua. Importa, ainda, salientar que a instalação nos clientes é realizada de forma semi-automática por motivos de segurança, isolamento das infraestruturas e necessidade de controlo sobre o momento da atualização. Esta característica, embora não totalmente automatizada, foi considerada uma salvaguarda arquitetural essencial.

Em primeiro lugar, procedeu-se a uma clarificação estrutural entre as fases de *Continuous Integration* (CI) e *Continuous Delivery* (CD). A ausência de mecanismos de teste suficientemente abrangentes na solução conduzem, frequentemente, à realização de testes críticos

diretamente em produção. Tal prática resulta na manutenção prolongada de *branches* de *release* e *hotfix*, comprometendo a clareza do *scope* das atualizações — uma vez que correções e novas funcionalidades acabavam por se misturar no mesmo ciclo.

Com a introdução da automatização do *deployment*, estabeleceu-se que o ponto de entrada do processo ocorre na integração no *branch main*, momento em que os *branches* de *release* ou *hotfix* são aprovados. Esta decisão não só separa de forma explícita os fluxos de CI e CD, como também abre espaço para a introdução de mecanismos de qualidade, nomeadamente a execução de testes em ambientes dedicados.

Portanto, uma vez integrada a mudança no *branch main* o fluxo idealizado bifurca-se. De denotar que a partir deste ponto o processo decorre integralmente no *Portainer*, por forma a facilitar a instalação com uma interface gráfica mais intuitiva:

- **Atualizações:** Efetua-se a obtenção das imagens já disponibilizadas no *Azure Container Registry* e subsequente orquestração dos serviços em execução;
- **Instalações:** Implicam a especificação de variáveis de ambiente utilizando *templates* previamente definidos e alojados num repositório orquestrador, ajustando-os aos dados privados e credenciais específicas de cada cliente. A variabilidade desta configuração é significativamente inferior à do cenário pré-existente, em que, além de erros de parametrização, eram recorrentes falhas de configuração no *IIS* e nos *Application Pools*.

Concluídas estas etapas, a atualização das bases de dados é realizada por intermédio da ferramenta interna desenvolvida para esse efeito, assegurando consistência entre versões e integridade transacional.

Por fim, procede-se à configuração de utilizadores no *provider* de autenticação. Esta atividade não foi alvo de esforços de automatização nesta fase, dado que a gestão de utilizadores será incorporada como responsabilidade funcional da plataforma, passando futuramente para o domínio dos administradores do sistema.

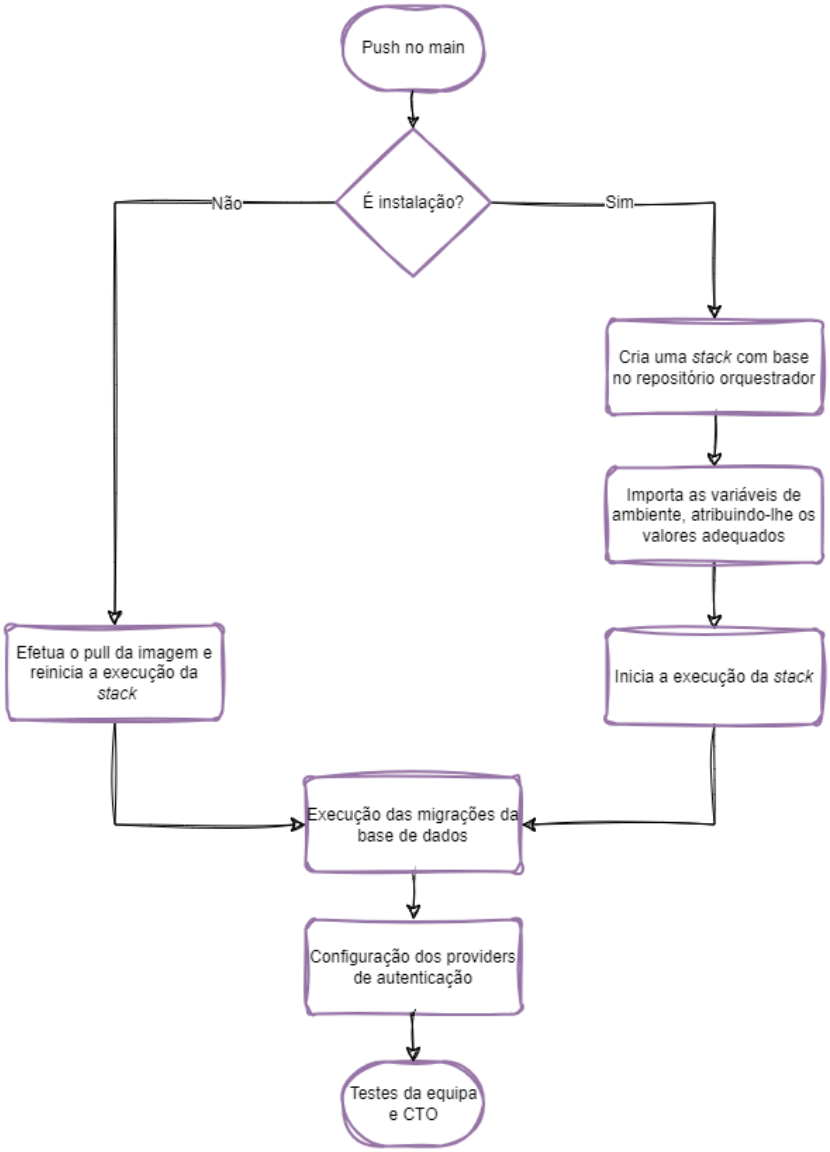


Figura 3.5: Diagrama de Processo da Solução Idealizada

3.2.2 Arquitetura e Componentes

A filosofia *build once, deploy everywhere* foi adotada como base, assegurando que cada serviço é construído e versionado uma única vez, sendo posteriormente disponibilizado em diferentes ambientes mediante parametrização. Esta abordagem garante consistência dos artefactos, reduz redundâncias e cria as bases para a automatização do *deployment*.

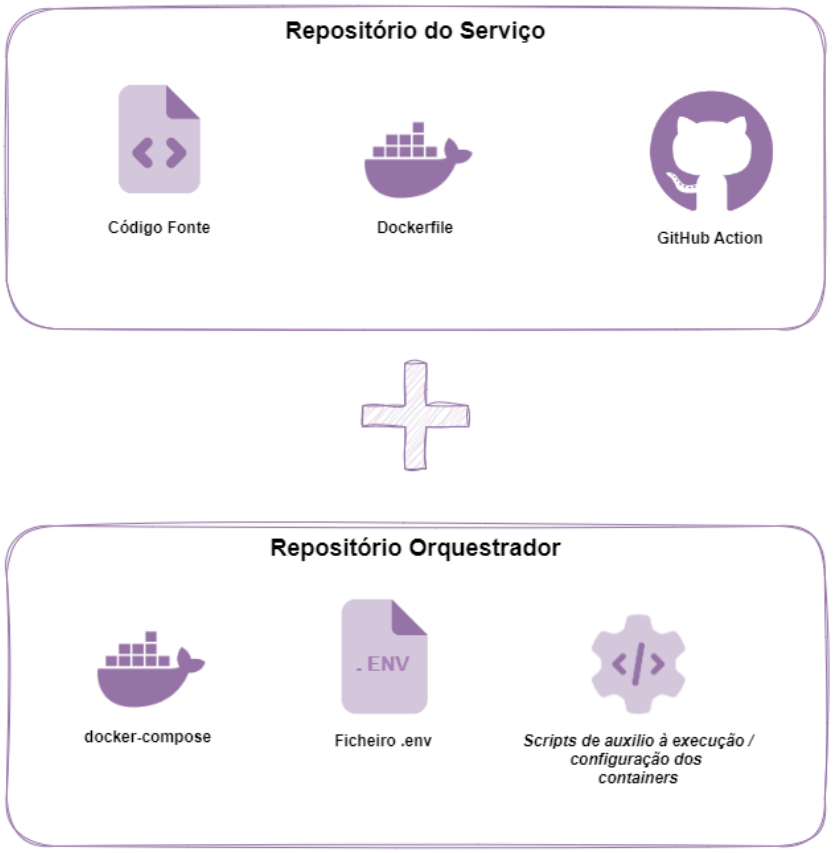


Figura 3.6: Diagrama ilustrativo dos Componentes principais da Solução

Cada repositório contém um serviço e inclui um *Dockerfile* responsável pela construção da sua imagem *Docker*. Estas imagens são independentes de configurações específicas (Auth0, Temas, Bases de Dados, etc.), sendo tais parâmetros injetados via variáveis de ambiente e ficheiros de configuração já nos *docker composes*. Nos repositórios definem-se ainda *GitHub Workflows* que permitem versionamento automático, construção e publicação das imagens para o *Azure Container Registry* (ACR).

Complementarmente, existe um repositório orquestrador (`PlexHub.Orquestrador`), concebido como *single source of truth*. Neste armazenam-se os *templates* necessários à instalação:

- **Ficheiro de Orquestração Global (`compose.yaml`)** - Contém a lista de serviços a implantar no ambiente remoto, neste apenas se configuram por predefinição os serviços que são obrigatoriamente incluídos em todas as instalações dos clientes. É possível, consoante a necessidade, vir a definir diferentes *compose.yaml* com configurações não *standard* caso, por exemplo, surjam instalações que recorram ao mesmo conjunto de serviços
- **Ficheiros de Orquestração por Serviço** - *Composes* orientados ao ambiente de produção para cada serviço
- **Ficheiros Environment** - Utilizados para armazenar dados privados como credenciais, conexões à base de dados (estes valores não estarão presentes no repositório) e dados que não acarretam qualquer questão de segurança (como os temas)
- **Scripts (`entrypoint.sh`)** - Usados como apoio para configurações adicionais.

Componentes Principais

Dockerfile

Conforme evidenciado, cada serviço contém o seu próprio Dockerfile com a responsabilidade única da construção de uma imagem *Docker* reutilizável entre diferentes ambientes de instalação e parametrizável. Esta decisão visa mitigar a situação atual em que o processo manual implica a repetição de compilações consoante o ambiente destino e depende de configurações locais.

Na conceção dos *Dockerfiles* teve-se em consideração as limitações identificadas na literatura (Capítulo 2) - nomeadamente as *builds* lentas e imagens pesadas - e as melhores práticas descritas na documentação oficial [31]. Pelo que se optou por uma estruturação em camadas lógicas:

1. Imagem Base Oficial - Garantindo segurança e atualizações, sempre que possível optando por imagens de menor dimensão (alpine ou distroless).
2. Instalação das dependências de sistema - Agrupadas para redução do número de camadas.
3. Cópia dos ficheiros de configuração - Nos serviços *backend*, a cópia inicial refere-se aos ficheiros `.csproj` necessários para restaurar dependências. Já no *frontend*, esta camada corresponde aos ficheiros `package.json/package-lock.json` que permitem instalar dependências Node.
4. Restauro das Dependências - No caso dos serviços *.NET*, a etapa de restauração de pacotes inclui também a configuração para acesso ao *feed* privado de *NuGets*, garantindo que todas as dependências internas ficam disponíveis de forma segura e consistente durante a construção da imagem.
5. Cópia do Código Restante
6. Build e Publish da Aplicação
7. Imagem final mínima, apenas com os binários necessários

Para reforçar a eficiência, prevê-se a introdução de ficheiros `.dockerignore` para exclusão de artefactos locais (`bin`, `obj`, `node_modules`) e ficheiros do IDE, evitando que a *build* seja inflacionada.

Docker Compose

Um dos grandes obstáculos identificados nos fluxos atuais foi a dificuldade em gerir as múltiplas dependências entre serviços e respetivas configurações específicas por cliente. Após rever as abordagens existentes, considerou-se que *Kubernetes* ou outras soluções de orquestração seriam demasiado complexas face ao contexto e recursos da equipa. Por isso, definiu-se a utilização de *Docker Compose*, que oferece simplicidade e flexibilidade adequadas a ambientes controlados como os dos clientes.

O design do *Compose* assenta em dois princípios: isolamento de redes e padronização através de padrões multi-*container*. O isolamento em duas redes distintas — uma interna partilhada e outra restrita à comunicação com a base de dados — reduz o risco de exposição indevida e ajuda a controlar a superfície de ataque.

Isolamento das redes:

O isolamento em duas redes distintas — uma interna partilhada e outra restrita à comunicação com a base de dados — reduz o risco de exposição indevida e ajuda a controlar a superfície de ataque. Assim, cada aplicação conterá até duas redes:

- Rede interna partilhada que permite comunicação entre os microsserviços ou entre o monólito de *frontend* e o *gateway*;
- Rede restrita usada exclusivamente entre cada serviço e a sua base de dados, evitando exposição indevida.

Isto reduz a superfície de ataque e simplifica o rastreio de comunicações entre *containers*.

Padrões aplicados:

Dos padrões expostos em 2.3.1 aplicou-se o *Single Service per Container* em que cada *container* executa apenas um processo, garantindo isolamento. No caso, a aplicação, bases de dados e serviços externos relacionados executam cada qual no seu *container*.

Configuração:

As variáveis sensíveis (palavras-passe, *connection strings*) não serão definidas no `compose.yaml`, mas sim em ficheiros .env, seguindo a recomendação oficial do *Docker*.

Serviços auxiliares como backups automáticos de bases de dados poderão ser incluídos no *compose*, de forma transparente para a equipa de desenvolvimento.

GitHub Workflow

Por fim, para superar a morosidade e vulnerabilidade a erros do fluxo manual (criação de *branches*, *builds* locais, publicação manual no IIS), desenhou-se uma estratégia baseada em **GitHub Workflows**, que automatiza desde a integração até à publicação no *Azure Container Registry* (ACR).

A decisão de isolar *workflows* por serviço surge da constatação de que a arquitetura de microsserviços do PlexHub exige independência: cada serviço poderá ser construído, testado e publicado sem impactar os restantes, reduzindo falhas propagadas.

O fluxo definido é:

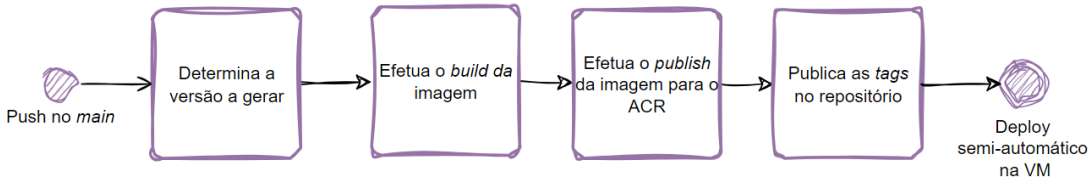


Figura 3.7: Fluxograma do *GitHub Workflow*

O processo é desencadeado automaticamente pela integração de alterações no *branch main*, iniciando-se imediatamente o cálculo da versão seguinte com base em *Semantic Versioning* [32]. Esta automatização do versionamento representa uma das transformações mais significativas do fluxo, eliminando por completo a necessidade de verificação manual das *tags* existentes no repositório, a definição manual da versão na aplicação e o *push* manual das *tags* correspondentes. Subsequentemente, procede-se à construção da imagem *Docker* utilizando

o *Dockerfile* previamente definido (3.2.2), seguida da sua publicação no *Container Registry*. O processo culmina com a criação automática das *tags* no repositório, estabelecendo um ponto de rastreabilidade claro. A conclusão bem-sucedida desta pipeline marca a transição entre o contexto automatizado de *deployment* e as configurações manuais necessárias no ambiente produtivo do cliente.

É importante destacar que, contrariamente ao habitual em *pipelines* de CI/CD, este fluxo não inclui um passo dedicado à execução de testes. Esta omissão justifica-se pela ausência de testes unitários ou de integração no projeto atual - os únicos testes existentes limitam-se a validações arquiteturais executadas numa fase anterior do desenvolvimento. Consequentemente, o código que atinge o pipeline de *deployment* já foi previamente validado.

A estratégia definida nestas subsecções reflete não apenas boas práticas de CI/CD, mas responde diretamente ao problema identificado: reduzir o esforço humano e assegurar consistência entre clientes, ao mesmo tempo que preserva simplicidade de operação para a equipa da PlexIT.

3.2.3 Distribuição e Instalação nos Clientes

Embora a construção e publicação dos artefactos seja inteiramente automatizada, a instalação nas infraestruturas dos clientes não é realizada diretamente pelas *pipelines* CI/CD. O *pull* e execução dos serviços é efetuado manualmente pelo elemento responsável pela atualização / instalação no cliente, através do *Portainer*. A solução idealizada configura, portanto, uma distribuição do produto parcialmente descentralizada.

Isto deve-se principalmente a três fatores:

- **Segurança** - Os ambientes dos clientes são isolados e protegidos por VPN, sendo o acesso externo restrito, pelo que se pretende, assim, evitar violar requisitos de segurança dos próprios clientes e evitar a potencial rejeição das comunicações por estas serem interpretadas como intrusão.
- **Descentralização das Infraestruturas dos Clientes** - Efetuar o *deployment* completo requer que o ambiente do cliente esteja disponível e acessível à pipeline e as credenciais de acesso às VPNs estejam armazenadas de forma segura remotamente, esta abordagem introduz um acoplamento significativo com o ambiente produtivo do cliente, tornando a pipeline um *single-point of failure* e condicionando o seu normal funcionamento à disponibilidade do ambiente do cliente.
- **Maior controlo sobre o momento de instalação / atualização** - A descentralização das atualizações garante uma maior autonomia relativamente à janela temporal em que estas são efetuadas e inclusivamente à própria necessidade de se efetuar a atualização para o cliente em específico. Esta abordagem permite que estas sejam incorporadas de forma planeada, controlada e integrada nos procedimentos de gestão de mudanças do cliente.

Esta solução alinha-se com os princípios de:

- **Modularidade** - cada serviço tem o seu ciclo de vida e pipeline, a *build* e publicação das imagens é automática e centralizada e devidamente separada das ações de *pull* e execução das imagens
- **Padronização** - construção unificada das imagens *docker* (uma imagem *docker* publicada é usada por todos os clientes)

- **Parametrização** - configurações específicas são injetadas por cliente via variáveis de ambiente e / ou ficheiros auxiliares e semi-automatização - o processo de *build*, *publish* e versionamento serão automatizados, contudo o *pull* e *deploy* serão manuais, suportando o crescimento do projeto e as suas características intrínsecas

Para finalizar o presente capítulo, enquadram-se os requisitos funcionais e restrições definidos em 3.1.2 no *design* da solução:

Tabela 3.3: Mapeamentos das Requisitos e Soluções Planeadas

Requisitos	**Enquadramento no Design**
RF1	Pipelines com versionamento automático e build / publish
RF2	Docker Compose
RF3	Publicação de uma única imagem no ACR, ficheiros `.env` e parametrização
RF4	Automatização de várias tarefas com o objetivo de minimizar esse tempo
RT1	Garantido através do uso de Dockerfiles e composes genéricos e parametrizáveis
RT2	Todas as ferramentas escolhidas aderem a esta restrição
RT3	Todas as *pipelines* foram desenvolvidas no ambiente *GitHub*

Capítulo 4

Implementação da Solução

4.1 Âmbito do Trabalho

Dada a complexidade do PlexHub e o facto deste se encontrar em constante evolução - tanto pela sua natureza como projeto de longo prazo em desenvolvimento, como pela necessidade de suporte contínuo em múltiplas instalações de clientes -, tornou-se inviável aplicar a solução proposta de forma integral a todos os seus componentes. Ademais, qualquer alteração na infraestrutura ou no processo de *deployment* poderia influenciar negativamente o funcionamento normal dos serviços, colocando em risco a estabilidade das soluções já em produção. Para mitigar este risco e assegurar a viabilidade da implementação sem comprometer o sistema existente, optou-se por uma abordagem seletiva. Foram escolhidos três serviços representativos do ecossistema do PlexHub, tendo sido extraída a versão mais atualizada:

- Um serviço backend em .NET com Entity Framework Core;
- Um serviço backend em .NET com Dapper;
- Um serviço frontend (monolítico).

Esta seleção permite abranger diferentes cenários e desafios técnicos que ocorrem no deployment do sistema, nomeadamente:

- A gestão de migrações de base de dados com EF Core;
- A execução de scripts SQL manuais com Dapper;
- A compilação e personalização de estilos no frontend.

Através desta amostra cuidadosamente escolhida, foi possível implementar a solução de automatização proposta, demonstrar a sua aplicabilidade prática e identificar os ajustes necessários à sua futura generalização para toda a solução PlexHub. Dada a arquitetura de microsserviços, sem um mecanismo de *service discovery*, tornou-se necessário extrair o projeto `PlexHub.Gateway` baseado em ficheiros `ocelot.json` e proceder à automatização do seu *deployment*, procedimento igualmente aplicado ao serviço responsável pela comunicação com o *provider* de autenticação (`PlexHub.Auth0`). Estes passos não serão descritos em detalhe, uma vez que o *GitHub Workflow* e os *Dockerfiles* gerados seguem de forma muito próxima os modelos aplicados aos restantes serviços de *backend*. Contudo, na subsecção 4.2.3 será apresentada uma breve descrição dos mecanismos de construção dinâmica dos ficheiros `ocelot.json` específicos do `PlexHub.Gateway`.

Para maior isolamento dos projetos selecionados, foi criada uma equipa específica no *GitHub* da Organização, contendo apenas como membros o supervisor da *PlexIT*, a autora do presente documento e os dois colegas alocados ao projeto *PlexHub*. Nesta, criaram-se os repositórios representativos dos serviços supramencionados, contendo a versão mais recente destes aquando da implementação, e um repositório orquestrador. Em adição, foi necessário configurar repositórios para os *packages* internos, conforme brevemente se explicitará.

Todos os serviços usados sofreram alterações de nomenclatura:

Tabela 4.1: Mapeamento dos nomes dos Serviços

Nome Original	Nome no Âmbito da Dissertação
PlexHub.Base	Tese.Base
PlexHub.Erp	Tese.Erp
PlexHub	Tese.Frontend
PlexHub.Gateway	Tese.Gateway
PlexHub.Auth0	Tese.Auth0
PlexHub.Servicos.Contracts	Tese.Shared.Contracts
PlexHub.Servicos.Core	Tese.Shared.Core
PlexHub.Servicos.Infrastructure	Tese.Shared.Infrastructure

4.2 Descrição da Implementação

4.2.1 Gestão do NuGet Interno

Paralelamente ao desafio principal de automatizar o *deployment* do projeto PlexHub, a equipa iniciou uma migração estratégica para o ambiente GitHub. Esta transição exigiu a implementação de passos adicionais não contemplados no planeamento inicial, os quais são detalhados de seguida.

Contexto e Estrutura

Conforme explicitado no capítulo anterior, todos os projetos de *backend* dependem de uma solução interna denominada `PlexHub.Servicos`, que se organiza em três projetos com versionamento independente:

- `PlexHub.Contracts` - Centraliza todos os modelos de dados representativos dos eventos e mensagens Request/Response do RabbitMQ
- `PlexHub.Core` - Agrupa exceções personalizadas, enumeráveis, modelos de domínio e abstrações comuns ao ecossistema
- `PlexHub.Infrastructure` - Contém implementações partilhadas, incluindo serviços de envio de email e outras funcionalidades transversais

Migração dos Artefactos

Anteriormente, estes projetos encontravam-se alojados nos Artefactos do *Azure DevOps* sobre um *Feed* da organização. O objetivo estratégico consiste em migrar esta infraestrutura para os *GitHub Packages*, otimizando simultaneamente os processos de gestão e distribuição.

Em adição, pretende-se a migração deste repositório hospedado no *Azure DevOps* para o *GitHub*. Este processo de migração levantou uma questão arquitetural fundamental: manter os três projetos num repositório único com versionamento independente ou separá-los em repositórios distintos. Após análise das implicações de cada abordagem, verificou-se que a manutenção de projetos independentemente versionados num único repositório introduziria complexidades significativas na gestão de *tags*, nos mecanismos de *trigger* dos *workflows*, e na organização dos *packages* resultantes. Consequentemente, em conjunto com a equipa de desenvolvimento do PlexHub, optou-se pela separação destes projetos em repositórios individuais. Esta decisão alinha-se com as melhores práticas para microsserviços verdadeiramente independentes, eliminando a complexidade desnecessária no desenvolvimento de mecanismos de automatização, simplificando substancialmente a gestão dos *packages*, e garantindo que cada projeto mantém total autonomia no seu ciclo de desenvolvimento e publicação.

Assim, extraiu-se a versão mais recente do repositório e foram criados repositórios privados na organização contendo o código base de cada projeto. Finalmente, ajustaram-se as nomenclaturas do projeto para que mais claramente se distinguissem do original.

Implementação da Automação

Em adição à migração e tendo em vista a maximização da eficiência operacional, foram desenvolvidos mecanismos automáticos para o versionamento e publicação dos *packages*. Esta automatização materializou-se na criação de três *GitHub Workflows* com estrutura similar, abaixo detalhada:

Trigger e Inicialização

O *Workflow* é acionado automaticamente mediante qualquer operação de push no repositório. Dado que estes repositórios não seguem a estratégia de *branching* convencional adotada pelos restantes projetos da organização (que utilizam *branches* específicos para desenvolvimento e produção), optou-se por não implementar filtros baseados em *branches*. Esta abordagem significa que qualquer alteração integrada no repositório resulta automaticamente na geração de uma nova versão do *package*.

Para otimizar a execução, foram configurados *paths* de exclusão que ignoram diretórios contendo ficheiros do *Git* e *GitHub*, evitando *builds* desnecessários para alterações que não afetam o código-fonte propriamente dito.

Adicionalmente, foi implementada uma cláusula de *concurrency* [33] que previne a execução simultânea de múltiplos *workflows*. Em cenários de *pushes* consecutivos, o sistema interrompe automaticamente a execução do *workflow* anterior, garantindo que apenas a versão mais recente é processada. Este mecanismo evita conflitos de versionamento e otimiza a utilização de recursos computacionais, assegurando que os *packages* publicados refletem sempre o estado mais atual do código.

```yaml
name: "Deploy Tese.Shared.Contracts to GitHub Packages"

on:
  push:
    paths-ignore:
      - '.github/**'
      - '.gitignore'

env:
  CONTRACTS_PATH: 'src/Tese.Shared.Contracts/Tese.Shared.Contracts.csproj'
  CONTRACTS_DIR: 'src/Tese.Shared.Contracts/'

concurrency:
  group: ${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: true
```

Figura 4.1: *Triggers* do *Workflow*

Job de Publicação do Package

Toda a lógica do *workflow* foi desenvolvida num único *job*, pois permite um melhor controlo do estado entre *steps* e evita a complexidade de partilhar artefactos entre *jobs* diferentes, garantindo que a versão calculada seja consistentemente aplicada tanto na geração do *package* como na criação das *tags*.

A primeira fase do *Job* corresponde à preparação do ambiente, para tal são executados os seguintes passos:

- Atribuição das permissões necessárias; para todos os *workflows* foi necessário incluir a permissão de escrita - obrigatória para que possam fazer o *push* das *tags*. No caso do `Tese.Shared.Contracts` dado que depende do *package* `Tese.Shared.Core`, também foi necessário atribuir a permissão de leitura de *packages*
- *Checkout* do Código com o histórico completo, necessário ao cálculo de versões
- Configuração do *.NET SDK*, no caso são introduzidas duas versões (`6.0.x` e `8.0.x`) devido às dependências dos projetos
- Adição do *feed* do *GitHub Packages* (apenas ocorre no caso do `Tese.Shared.Contracts`)

```yaml
jobs:
  build-publish:
    runs-on: ubuntu-latest
    environment: Production
    permissions:
      contents: write
      packages: read
    steps:
      - name: Checkout code
        uses: actions/checkout@v4
        with:
          fetch-depth: 0

      - name: Set up .NET SDK
        uses: actions/setup-dotnet@v4
        with:
          dotnet-version: |
            6.0.x
            8.0.x

      - name: Configure NuGet sources
        run: |
          dotnet nuget add source \
           --username [redacted] \
           --password ${{ secrets.GITHUB_TOKEN }} \
           --store-password-in-clear-text \
           --name github \
           "[redacted]/index.json"
```

Figura 4.2: Preparação do Ambiente

Seguem-se os *steps* de restauro das dependências do projeto e da *build* do mesmo em modo *release*.

```yaml
      - name: Restore dependencies
        run: dotnet restore ${{ env.CONTRACTS_PATH }}

      - name: Build project
        run: dotnet build ${{ env.CONTRACTS_PATH }} --configuration Release --no-restore
```

Figura 4.3: Restauro de Dependências e build do Projeto

Utiliza-se o `github-tag-action` [34] para calcular automaticamente a próxima versão baseada nas convenções de versionamento semântico, sem ainda criar as *tags* no repositório (propriedade `DRY_RUN`).

```yaml
52          - name: Check Version
53            id: bump_version
54            uses: anothrNick/github-tag-action@v1
55            env:
56              GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
57              PRERELEASE: false
58              DRY_RUN: true
```

Figura 4.4: Verificação da nova versão

O *package* é gerado com a versão determinada no *step* anterior e publicado no *feed* privado, ignorando duplicados para evitar erros.

```yaml
60        - name: Generate NuGet Package
61          run: |
62            dotnet pack ${{ env.CONTRACTS_DIR }} \
63            --configuration Release \
64            -p:PackageVersion=${{ steps.bump_version.outputs.new_tag }} \
65            --output packages --no-restore --no-build
66
67        - name: Publish NuGet Packages
68          run: dotnet nuget push packages/*.nupkg \
69               --api-key ${{ secrets.PUSH_NUGET }} \
70               --source "[redacted]" \
71               --skip-duplicate
```

Figura 4.5: Geração e Publicação do *Package*

Após a publicação bem-sucedida do *package*, as *tags* são criadas no repositório para marcar a versão publicada.

```yaml
73          - name: Push version tag
74            uses: anothrNick/github-tag-action@v1
75            env:
76              GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
77              PRERELEASE: false
```

Figura 4.6: Publicação das *tags*

Quando os packages foram publicados pela primeira vez, foi necessário aceder à aba de *Packages* da organização:

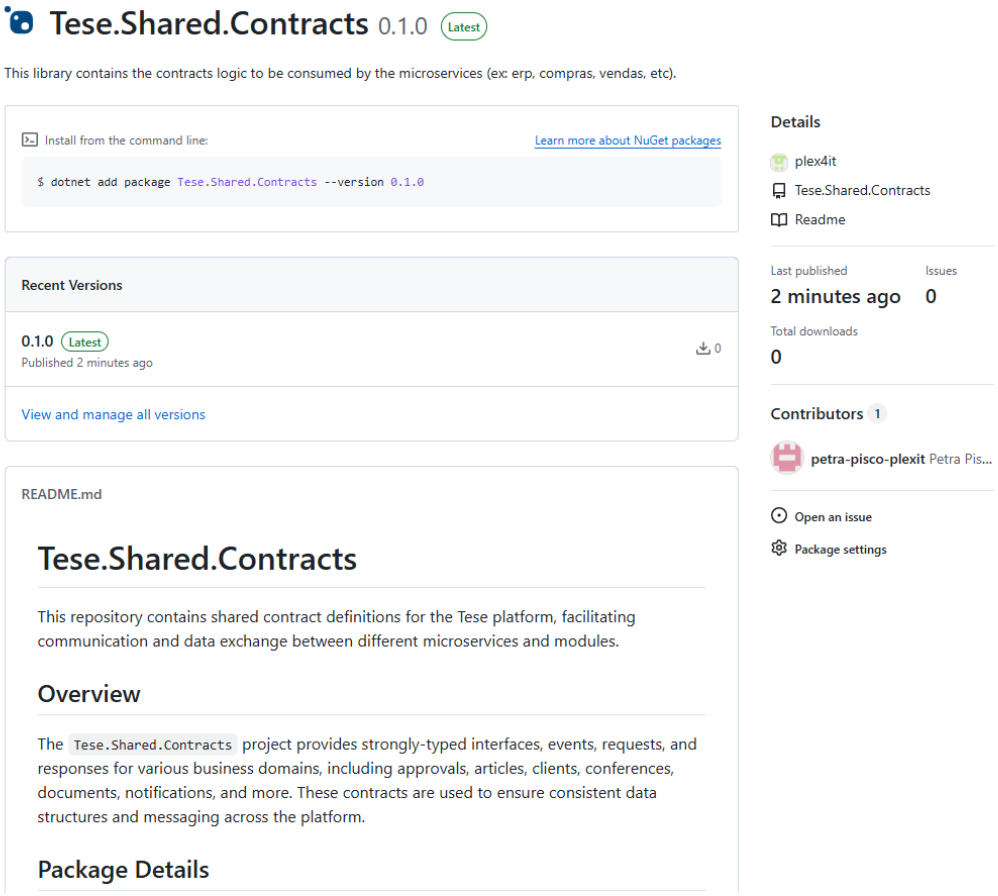


Figura 4.7: Package `Tese.Shared.Contracts` publicado

E configurar acessos de leitura para as *GitHub Actions* dos repositórios `Tese.Erp` e `Tese.Base`:

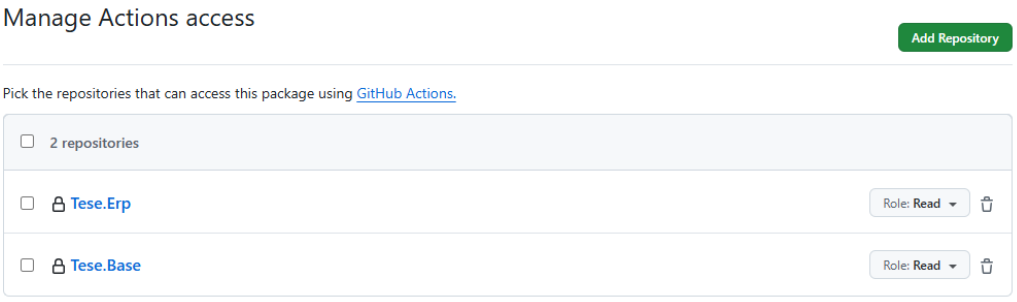


Figura 4.8: Atribuição de permissões de leitura do *Package*

4.2.2 Pipeline de Continuous Delivery com GitHub Actions

Foi implementado um *workflow* de *Continuous Delivery* (CD) automatizado em cada repositório que garante a entrega contínua e confiável dos serviços. Este processo é ativado automaticamente sempre que ocorre um *push* no *branch main*, assegurando que apenas as

alterações validadas sejam incluídas no *deployment*. Abaixo apresenta-se a *pipeline* implementada para o serviço `Tese.Base`, contudo esta teve a mesma estrutura para todos os repositórios, variando apenas os valores de `IMAGE_NAME` e `DOCKERFILE_PATH`.

```yaml
name: Bump version And Push Docker Image to ACR
on:
  push:
    branches:
      - main
    paths-ignore:
      - '.github/**'
      - '.gitignore'
      - 'src/Tese.Base.WebApi/appsettings.json'
      - 'src/Tese.Base.WebApi/appsettings.Development.json'
      - 'src/Tese.Base.WebApi/Properties/launchSettings.json'
      - 'README.md'
env:
  IMAGE_NAME: "tese_base"
  DOCKERFILE_PATH: "src/Tese.Base.WebApi/Dockerfile"
  DKR_GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}

concurrency:
  group: ${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: true
```

Figura 4.9: Triggers do Workflow e Configurações Iniciais

Foram excluídos alguns *paths* cujas modificações não devem despoletar a geração de uma nova versão, sendo estes: o diretório `.github`, o `README.md`, o `.gitignore` e os ficheiros de configuração que não têm influência sobre a imagem *docker* gerada.

Em adição, utilizou-se uma cláusula `concurrency` que garante que em situações em que são realizados múltiplos `push` no *branch* `main`, quaisquer execuções prévias ao mais recente destes são paradas, desta forma evitando que se gerem múltiplas imagens.

A *pipeline* foi estruturada em três *jobs* interdependentes que executam tarefas específicas:

1. **Verificação da Versão**: Verifica a versão mais recente no repositório e calcula a seguinte, sem efetuar quaisquer alterações;
2. **Build e Publicação**: Constrói a imagem *Docker* e publica-a no *Azure Container Registry (ACR)*;
3. **Versionamento Automático**: Responsável pelo incremento automático da versão do projeto e publicação de *tags* no repositório.

O primeiro *job* implementa um sistema de versionamento automático baseado no *Semantic Versioning* [32], já adotado como padrão na empresa. Utiliza a *GitHub Action* especializada [34] que analisa as *tags* existentes no repositório e gera uma nova versão com base na mensagem do *commit*. O sistema reconhece palavras-chave específicas nas mensagens de *commit*:

- **#major**: para alterações incompatíveis com versões anteriores

- **#minor**: para novas funcionalidades retro-compatíveis
- **#patch**: para correções de bugs

Por predefinição, quando nenhuma palavra-chave é especificada, o sistema incrementa automaticamente a versão *minor*. Foi, ainda, incluído um passo de *debug* que facilita a identificação e resolução de eventuais anomalias no processo de versionamento. Denote-se a propriedade `DRY_RUN: true` que assegura que a *tag* não é adicionada ao repositório.

```yaml
23      check-version:
24        environment: Production
25        runs-on: ubuntu-latest
26        outputs:
27          new_tag: ${{ steps.check_version.outputs.new_tag }}
28          old_tag: ${{ steps.check_version.outputs.old_tag }}
29        permissions:
30          contents: read
31        steps:
32          - name: Checkout
33            uses: actions/checkout@v4
34            with:
35              #Histórico completo (necessário para o step abaixo)
36              fetch-depth: '0'
37
38            # https://github.com/marketplace/actions/github-tag-bump
39          - name: Check latest tag
40            id: check_version
41            uses: anothrNick/github-tag-action@v1
42            env:
43              GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
44              DRY_RUN: true
45              PRERELEASE: false
46
47          - name: Debug
48            run: |
49              echo "new_tag: ${{ steps.check_version.outputs.new_tag }}"
50              echo "old_tag: ${{ steps.check_version.outputs.old_tag }}"
51              echo "part: ${{ steps.check_version.outputs.part }}"
52              echo "latest_commit_message: $(git log -1 --pretty=%B)"
```

Figura 4.10: *Job* de Verificação da Versão

O segundo *job* executa condicionalmente, apenas quando a *tag* gerada difere da última versão publicada. Esta validação previne tentativas desnecessárias de criação de imagens Docker já existentes no registry, otimizando o tempo de execução da *pipeline*. O processo de *build* e publicação segue uma sequência estruturada:

1. **Autenticação**: Realiza o *login* no Azure Container Registry utilizando a ação oficial [35]
2. **Construção da Imagem**: Executa o comando `docker build` com as seguintes configurações:
   - Utilização de variáveis de ambiente definidas no início do *workflow*
   - Especificação do caminho para o `Dockerfile` através da *flag* `-f`
   - Configuração segura do *GitHub Token* via *Docker Secrets* para acesso aos *packages* privados da empresa
   - Definição da *tag* da imagem através da *flag* `-t`
3. **Publicação**: Executa o `docker push` para enviar a imagem para o ACR
4. **Validação e Limpeza**: Inclui um passo de *debug* para verificar a correção da *tag* gerada, seguido do *logout* do *registry* para garantir a segurança do acesso. Note-se que o passo de *logout* também apresenta um comportamento condicional. A expressão `always()` é utilizada para garantir que mesmo que algum dos passos anteriores falhe, o *logout* é realizado. A expressão `steps.azure-login.outcome == 'success'` é utilizada para assegurar que o *logout* só ocorre se o *login* tiver sido bem sucedido.

```yaml
  build-publish:
    if: ${{ needs.check-version.outputs.new_tag != needs.check-version.outputs.old_tag }}
    environment: Production
    runs-on: ubuntu-latest
    needs: check-version
    steps:
    - uses: actions/checkout@v4
      # https://github.com/Azure/docker-login
    - name: Azure Container Registry Login
      id: azure-login
      uses: Azure/docker-login@v1
      with:
        # Container registry server url
        login-server: ${{secrets.ACR_LOGIN_SERVER}}
        # Container registry username
        username: ${{secrets.ACR_USERNAME}}
        # Container registry password
        password: ${{secrets.ACR_PASSWORD}}
    - name: Build and Push the Docker image
      run: |
        DOCKER_BUILDKIT=1 docker build -f ${{ env.DOCKERFILE_PATH }} \
        --secret id=github_token,env=DKR_GITHUB_TOKEN \
        -t ${{ secrets.ACR_LOGIN_SERVER }}/${{ env.IMAGE_NAME }}:${{ needs.check-version.outputs.new_tag }} \
        .
        docker push ${{ secrets.ACR_LOGIN_SERVER }}/${{ env.IMAGE_NAME }}:${{ needs.check-version.outputs.new_tag }}
    - name: Debug
      run: |
        echo "generated docker image: ${{ env.IMAGE_NAME }}:${{ needs.check-version.outputs.new_tag }}"
    - name: Docker logout
      if: always() && steps.azure-login.outcome == 'success'
      run: docker logout ${{ secrets.ACR_LOGIN_SERVER }}
```

Figura 4.11: Job para construir e publicar a imagem *Docker* no ACR

O terceiro e último *job* recorre à mesma *action* do primeiro, contudo, sem a definição da propriedade `DRY_RUN`. Neste caso, a permissão atribuída é de escrita (`write`) pois são

efetivamente criadas as *tags* no repositório. Denote-se que este *job* depende dos anteriores, sendo apenas executado se ambos forem concluídos com sucesso.

```
86      bump-version:
87        environment: Production
88        runs-on: ubuntu-latest
89        needs: [check-version, build-publish]
90        permissions:
91          contents: write
92        steps:
93          - name: Checkout
94            uses: actions/checkout@v4
95            with:
96              #Histórico completo (necessário para o step abaixo)
97              fetch-depth: '0'
98
99            # https://github.com/marketplace/actions/github-tag-bump
100         - name: Bump version and push tag
101           uses: anothrNick/github-tag-action@v1
102           env:
103             GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
104             PRERELEASE: false
105
```

Figura 4.12: *Job* de Auto versionamento e publicação de *Tags*

4.2.3 Gestão dos Serviços de Backend

Containerization

O *dockerfile* apresentado foi desenvolvido tendo por base a documentação oficial disponível em [36] e orientado segundo as camadas lógicas identificadas em 3.2.2, conforme a imagem abaixo:

Figura 4.13: *Dockerfile*

Na camada inicial, recorreu-se à imagem oficial **.NET 6.0 ASP.NET**, otimizada para Alpine Linux, de forma a garantir um menor consumo de recursos e maior segurança. As portas de rede foram mantidas expostas por predefinição (80 e 443), ainda que, na prática, estas sejam substituídas pelas definições presentes nos ficheiros `compose.yaml`.

Na fase seguinte, é utilizada a imagem **.NET 6.0 SDK**, que contém todas as ferramentas necessárias para a compilação do código e o restauro das dependências. É nesta camada que se encontra a principal diferenciação face ao modelo proposto pela documentação oficial da Microsoft: além da cópia do ficheiro de configuração do projeto e da execução do comando `dotnet restore`, procede-se também ao registo do repositório interno de *packages*. Para tal, recorreu-se a **Docker Secrets**, permitindo o acesso seguro ao *GitHub Token*, indispensável à autenticação no *feed* privado de pacotes da organização.

Concluída esta configuração, procede-se à cópia integral do código-fonte para o *container* e à respetiva compilação em modo `Release` através do comando `dotnet build`.

A etapa subsequente corresponde à fase de **publicação** (`dotnet publish`), na qual os binários são otimizados e preparados para distribuição. Nesta fase foi incluída a opção `/p:UseAppHost=false`, que assegura a exclusão do *host* específico da plataforma, favorecendo a portabilidade da aplicação.

Por fim, na construção da **imagem final**, retoma-se a imagem mínima de execução (correspondente a `aspnet: 6.0-alpine`). Apenas os ficheiros resultantes da publicação são copiados, excluindo *toolchains* e dependências transitórias utilizadas no processo de *build*. A definição do `ENTRYPOINT` garante a execução direta da API através do comando `dotnet Tese.Base.WebApi.dll`. Esta estratégia, característica de um *multi-stage build*,

permite reduzir substancialmente o tamanho da imagem final, aumentando a eficiência e simplificando a distribuição em diferentes ambientes.

Migrações das Bases de Dados

Conforme detalhado em secções anteriores, os serviços de *backend* recorrem tanto a *Entity Framework Migrations* como à execução de *scripts* SQL para gerir a evolução das bases de dados. A execução destas migrações é assegurada por uma ferramenta interna, capaz de direcionar automaticamente as instruções para a instância de base de dados pretendida. Embora tenha sido avaliada a possibilidade de automatizar integralmente este processo, a sua concretização exigiria a publicação e orquestração contínua de ferramentas auxiliares sempre que ocorressem alterações aos modelos de dados. Tal abordagem, ainda que tecnicamente viável, introduziria complexidade adicional e potenciais riscos em torno da integridade das migrações. Por esse motivo, optou-se por preservar o fluxo já estabelecido e validado pela equipa, garantindo estabilidade, consistência com os procedimentos anteriormente adotados e compatibilidade com os diferentes ambientes dos clientes.

Construção dinâmica dos ficheiros `ocelot.json`

Conforme evidenciado no início do presente capítulo, o serviço `Tese.Gateway` é responsável pelo redirecionamento das comunicações para os microsserviços adequados via ficheiros *Ocelot*.

Diferentes clientes requerem serviços distintos, disponibilizados em *hosts*, portas e métodos `HTTP` também variáveis. Por essa razão, tornou-se imperativo desenvolver um mecanismo que conferisse maior flexibilidade na construção dos ficheiros `ocelot.json`.

Para tal, foi introduzida a classe estática `OcelotConfigGenerator`, cuja única responsabilidade é gerar dinamicamente o ficheiro `ocelot.json`, recorrendo às variáveis de ambiente disponíveis.

A classe contém apenas o método `GenerateOcelotConfigIfNeeded`, que recebe como parâmetro o caminho completo onde o ficheiro deverá ser escrito no contexto da aplicação. O primeiro passo consiste em garantir que a escrita ocorre no diretório correto, seguindo-se a obtenção do conjunto de variáveis de ambiente necessárias:

```
1 reference | AzureAD\PetraPisco, 12 hours ago | 1 author, 2 changes
internal static void GenerateOcelotConfigIfNeeded(string configPath)
{
    // Path validation: ensure configPath is inside the app base directory
    string baseDir = AppContext.BaseDirectory;
    string safePath = Path.GetFullPath(configPath, baseDir);

    if (!safePath.StartsWith(baseDir, StringComparison.OrdinalIgnoreCase))
    {
        throw new InvalidOperationException("Attempted to write outside the application directory.");
    }

    var servicesConfig = Environment.GetEnvironmentVariable("SERVICES") ?? string.Empty;
    var baseUrl = Environment.GetEnvironmentVariable("OCELOT_BASE_URL") ?? string.Empty;
    var extraMethodsConfig = Environment.GetEnvironmentVariable("OCELOT_EXTRA_METHODS") ?? string.Empty;
```

Figura 4.14: Excerto de código do método `GenerateOcelotConfigIfNeeded` (Processamento inicial)

As variáveis de ambiente `SERVICES` e `OCELOT_EXTRA_METHODS`, como os nomes indicam, podem conter múltiplos valores. Definiu-se, assim, que seguiriam os formatos `name:host:port`,

`name:host:port` e `name:X,Y;name:X`, respetivamente. A título de exemplo, considerem-se os seguintes valores:

- `OCELOT_BASE_URL` - https://gateway.example.com
- `SERVICES` - serviceA:serviceA.example.com:5001,serviceB:serviceB.example.com:5002
- `OCELOT_EXTRA_METHODS` - serviceA:Options,Connect;serviceB:Options

Após a execução do método, seria gerado um ficheiro semelhante ao seguinte:

```
{
  "GlobalConfiguration": {
    "BaseUrl": "https://gateway.example.com",
    "RequestIdKey": "OcRequestId"
  },
  "Routes": [
    {
      "DownstreamPathTemplate": "/api/{everything}",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "serviceA.example.com",
          "Port": 5001
        }
      ],
      "UpstreamPathTemplate": "/api/serviceA/{everything}",
      "UpstreamHttpMethod": [ "Get", "Post", "Put", "Patch", "Delete", "Options", "Connect" ]
    },
    {
      "DownstreamPathTemplate": "/api/{everything}",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "serviceB.example.com",
          "Port": 5002
        }
      ],
      "UpstreamPathTemplate": "/api/serviceB/{everything}",
      "UpstreamHttpMethod": [ "Get", "Post", "Put", "Patch", "Delete", "Options" ]
    }
  ]
}
```

Figura 4.15: Exemplificação do `ocelot.json`

A fase seguinte corresponde à lógica de substituição da variável de ambiente `OCELOT_EXTRA_METHODS`. Sempre que esta variável é preenchida, o método percorre os diferentes elementos (separados por `;`) e, para cada um, obtém os métodos HTTP a adicionar. Para evitar adições inválidas, todos os *inputs* são validados com base na lista `AllowedMethods`.

```
var extraMethodsMap = new Dictionary<string, string[]>(StringComparer.OrdinalIgnoreCase);
if (!string.IsNullOrWhiteSpace(extraMethodsConfig))
{
    var entries = extraMethodsConfig.Split(';', StringSplitOptions.RemoveEmptyEntries | StringSplitOptions.TrimEntries);
    foreach (var entry in entries)
    {
        var parts = entry.Split(':', 2, StringSplitOptions.RemoveEmptyEntries | StringSplitOptions.TrimEntries);
        if (parts.Length == 2)
        {
            var serviceName = parts[0];
            var methods = parts[1]
                .Split(',', StringSplitOptions.RemoveEmptyEntries | StringSplitOptions.TrimEntries)
                .Where(m => AllowedMethods.Contains(m))
                .Distinct(StringComparer.OrdinalIgnoreCase)
                .ToArray();
            extraMethodsMap[serviceName] = methods;
        }
    }
}
```

Figura 4.16: Processamento da variável de ambiente `OCELOT_EXTRA_METHODS`

```
private static readonly HashSet<string> AllowedMethods = new(StringComparer.OrdinalIgnoreCase)
{
    "Get", "Post", "Put", "Patch", "Delete", "Options", "Connect"
};
```

Figura 4.17: Variável estática `AllowedMethods`

Segue-se o processamento da variável `SERVICES`. Tal como anteriormente, o método percorre cada elemento (neste caso separados por `,`) e atribui-os aos diferentes componentes de uma *route*. Neste ponto, é também injetado o conjunto de métodos HTTP comuns a todos os serviços, definido por predefinição. Assim, a variável `OCELOT_EXTRA_METHODS` só será utilizada quando for necessário adicionar métodos como `"Options"` ou `"Connect"`. Embora nenhum dos serviços da amostra requeira esta configuração, ela é utilizada em determinados casos na solução PlexHub.

```
var defaultMethods = new[] { "Get", "Post", "Put", "Patch", "Delete" };
var routes = new List<object>();

if (!string.IsNullOrEmpty(servicesConfig))
{
    var services = servicesConfig.Split(',', StringSplitOptions.RemoveEmptyEntries | StringSplitOptions.TrimEntries);
    foreach (var service in services)
    {
        var parts = service.Split(':', StringSplitOptions.RemoveEmptyEntries | StringSplitOptions.TrimEntries);
        if (parts.Length == 3 && int.TryParse(parts[2], out int port))
        {
            var name = parts[0];
            var host = parts[1];

            // Merge default and extra methods for this service, only whitelisted
            var methods = defaultMethods;
            if (extraMethodsMap.TryGetValue(name, out var extra))
            {
                methods = [.. defaultMethods.Concat(extra)
                    .Where(m => AllowedMethods.Contains(m))
                    .Distinct(StringComparer.OrdinalIgnoreCase)];
            }

            routes.Add(new
            {
                DownstreamPathTemplate = "/api/{everything}",
                DownstreamScheme = "http",
                DownstreamHostAndPorts = new[] { new { Host = host, Port = port } },
                UpstreamPathTemplate = $"/api/{name}/{{everything}}",
                UpstreamHttpMethod = methods
            });
        }
    }
}
```

Figura 4.18: Processamento da variável de ambiente `SERVICES`

O método é finalizado com a criação do objeto de configuração final e pela efetiva escrita no ficheiro:

```
var ocelotConfig = new
{
    GlobalConfiguration = new
    {
        BaseUrl = baseUrl,
        RequestIdKey = "OcRequestId"
    },
    Routes = routes
};

var configJson = JsonSerializer.Serialize(ocelotConfig, s_writeOptions);

if (routes.Count > 0 && !File.Exists(safePath))
{
    File.WriteAllText(safePath, configJson);
}
```

Figura 4.19: Processamento final do objeto de configuração e Escrita para o ficheiro

A classe é, posteriormente, injetada no `OcelotExtension`, que, por sua vez é invocado no `Program`. Desta forma, a lógica é despoletada sempre que a aplicação inicia:

```
internal static class OcelotExtension
{
    1 reference | AzureAD\PetraPisco, 15 hours ago | 1 author, 2 changes
    internal static IServiceCollection AddOcelot(this IServiceCollection services, ConfigurationManager configuration, WebApplicationBuilder builder)
    {
        string path = $"ocelot.{builder.Environment.EnvironmentName}.json";
        OcelotConfigGenerator.GenerateOcelotConfigIfNeeded(Path.Combine(AppContext.BaseDirectory, path));

        configuration.AddJsonFile(path, false, true);
        services.AddOcelot(builder.Configuration);

        return services;
    }
}
```

Figura 4.20: Classe `OcelotExtension`

4.2.4 Gestão do Monólito de Frontend

Mecanismos de Configuração Dinâmica

No caso do monólito de *Frontend* (`Tese.Base`), para além da execução dos procedimentos já enunciados para os restantes serviços — os quais serão detalhados posteriormente nesta subsecção — foi necessário implementar um conjunto adicional de adaptações específicas.

A versão pré-existente do código utilizava os *Build Environments* do Angular para a definição de parâmetros de configuração, nomeadamente o *provider* de autenticação e o URL do *Gateway*. Relativamente aos temas, estes eram aplicados de forma manual, através da substituição explícita de variáveis CSS. Contudo, esta abordagem revelou-se incompatível com o princípio *build once, deploy everywhere*, sobre o qual assenta a solução proposta. Tal incompatibilidade decorre do facto de os *Build Environments* serem resolvidos em tempo de compilação, impossibilitando a sua alteração após a criação da imagem *Docker*. Face a esta limitação, tornou-se imperativo conceber um mecanismo alternativo que permitisse a parametrização dinâmica da aplicação após a construção da imagem.

Numa primeira fase, foi elaborado um modelo representativo dos dados anteriormente contidos nos ficheiros de *Build Environment*, ao qual se acrescentou uma entrada destinada à definição das cores a aplicar:

```
1  export interface Environment {
2      production: boolean;
3      gatewayUrl: string;
4      appVersion: string;
5      auth0: {
6          domain: string;
7          clientId: string;
8          audience: string;
9      };
10     portal: string;
11     colors: Record<string, string>;
12 }
```

Figura 4.21: Modelo *Environment*

Com base neste modelo, foi produzido um ficheiro de configuração *template* (`config.json`), incluído na pasta `assets`. A decisão de recorrer a esta pasta justifica-se pelo facto de, no processo de compilação de projetos Angular, todos os ficheiros nela presentes serem preservados na sua forma original. Embora fosse possível configurar no `angular.json` uma

pasta de configurações dedicada, optou-se por recorrer à estrutura existente, de modo a reduzir o impacto das modificações numa aplicação de grande dimensão e complexidade.

```
{
    "production": "${PRODUCTION}",
    "gatewayUrl": "${GATEWAY_URL}",
    "appVersion": "${APP_VERSION}",
    "auth0": {
        "domain": "${AUTH0_DOMAIN}",
        "clientId": "${AUTH0_CLIENT_ID}",
        "audience": "${AUTH0_AUDIENCE}"
    },
    "portal": "${PORTAL}",
    "colors": {
        "--kendo-color-primary": "${COLOR_PRIMARY}",
        "--kendo-color-primary-subtle": "${COLOR_PRIMARY_SUBTLE}",
        "--kendo-color-primary-subtle-hover": "${COLOR_PRIMARY_SUBTLE_HOVER}",
        "--kendo-color-primary-subtle-active": "${COLOR_PRIMARY_SUBTLE_ACTIVE}",
        "--kendo-color-primary-rgb": "${COLOR_PRIMARY_RGB}",
        "--kendo-color-primary-hover": "${COLOR_PRIMARY_HOVER}",
        "--kendo-color-primary-active": "${COLOR_PRIMARY_ACTIVE}",
        "--kendo-color-primary-emphasis": "${COLOR_PRIMARY_EMPHASIS}",
        "--kendo-color-primary-on-subtle": "${COLOR_PRIMARY_ON_SUBTLE}",
        "--kendo-color-primary-on-surface": "${COLOR_PRIMARY_ON_SURFACE}"
    }
}
```

Figura 4.22: config.json

Subsequentemente, foi desenvolvido um serviço dedicado ao mapeamento dos dados provenientes do ficheiro `config.json` para o modelo *Environment*:

```typescript
@Injectable({
  providedIn: 'root'
})
export class ConfigurationService {
  private httpClient = inject(HttpClient);
  private readonly httpBackend = inject(HttpBackend);
  private readonly authConfig = inject(AuthClientConfig);
  private readonly CONFIG_URL = 'assets/config/config.json';
  private readonly RETRY_COUNT = 3;

  public async initialize(): Promise<Environment> {
    return await firstValueFrom(this.load());
  }

  private load(): Observable<Environment> {
    this.httpClient = new HttpClient(this.httpBackend);
    return this.httpClient.get<Environment>(this.CONFIG_URL)
      .pipe(
        retry(this.RETRY_COUNT),
        tap(config => {
          RuntimeConfiguration.initialize(config);
          this.setAuth0ConfigurationParameters(config);
          if (config.colors) {
            this.applyTheme(config.colors);
          }
        })
      );
  }

  private applyTheme(colors: Record<string, string>): void {
    for (const [key, value] of Object.entries(colors)) {
      document.documentElement.style.setProperty(key, value);
    }
  }

  private setAuth0ConfigurationParameters(config: Environment): void {
    this.authConfig.set({
      domain: config.auth0.domain,
      clientId: config.auth0.clientId,
      authorizationParams: {
        redirect_uri: window.location.origin,
        audience: `${config.auth0.audience}`,
        scope: 'openid email profile'
      },
      httpInterceptor: {
        allowedList: [
          {
            uri: `${config.gatewayUrl}/*`,
            tokenOptions: {
              authorizationParams: {
                audience: `${config.auth0.audience}`,
                scope: ''
              }
            }
          }
        ]
      }
    });
  }
}
```

Figura 4.23: *ConfigurationService*

O método `load()` devolve um *cold observable*, isto é, cada nova *subscription* origina um pedido independente do tipo `HTTP GET`, incorporando, adicionalmente, mecanismos de *retry*

em caso de falha. A execução dos *side effects* encontra-se encapsulada no operador `tap`, o qual assegura a inicialização do *provider* de autenticação, bem como a aplicação das definições de cores do tema.

Este serviço é utilizado exclusivamente no contexto do `APP_INITIALIZER`, de modo a garantir que todas as configurações necessárias são carregadas previamente ao arranque da aplicação Angular. Esta integração no ciclo de bootstrap é determinante, pois assegura que os parâmetros de autenticação e de estilo estão corretamente aplicados antes de qualquer outro componente ou serviço ser instanciado.

Na prática, esta decisão implica que qualquer alteração posterior às configurações — por exemplo, a redefinição de temas ou a modificação dos parâmetros de autenticação — requer a interrupção da execução do *container* e a sua reinicialização. Ainda que esta limitação possa parecer restritiva, não se revelou problemática no contexto do projeto. Em primeiro lugar, porque tais alterações constituem uma situação altamente pontual: desde a definição inicial dos temas e da configuração do Auth0, nunca se verificou a necessidade de os alterar em tempo de execução. Em segundo lugar, porque o processo de reinicialização é relativamente célere: a execução de um ciclo `docker compose down` seguido de `docker compose up` traduz-se, na prática, numa operação rápida, dado que as imagens se encontram previamente construídas e apenas é necessário restabelecer os *containers* e relançar a aplicação. Assim, não foi considerado determinante introduzir mecanismos adicionais que viabilizassem a reconfiguração dinâmica durante o ciclo de vida da aplicação.

Na linha 24, o `HttpClient` é injetado manualmente a partir do `HttpBackend`, o que permite contornar todos os *interceptors* HTTP registados. Esta opção é fundamental neste contexto, uma vez que evita dependências cíclicas que impediriam o arranque da aplicação: caso contrário, o `HTTP Client` tentaria recorrer ao *interceptor* `authHttpInterceptorFn` para injetar *headers* de segurança que, naquele momento, ainda não se encontram configurados.

O método `applyTheme` é responsável pela substituição dinâmica das variáveis CSS, assegurando a atualização imediata da aparência da interface.

Por fim, o método `setAuth0ConfigurationParameters` tem como finalidade inicializar o *provider* de autenticação a partir dos parâmetros definidos no ficheiro de configuração, garantindo a correta integração dos serviços de segurança.

Com o intuito de assegurar que as alterações introduzidas tivessem um impacto mínimo no quotidiano da equipa de desenvolvimento, foi concebida a classe `RuntimeConfiguration`. Esta solução materializa-se como uma camada de abstração para o acesso a campos definidos dinamicamente no ficheiro `config.json`. A disponibilização de métodos estáticos nesta classe não apenas mantém a consistência com o padrão de invocação anteriormente utilizado (`environment.var`), como também proporciona uma transição gradual e intuitiva, reduzindo a necessidade de *refactoring* extensivo. Deste modo, a `RuntimeConfiguration` contribui para a preservação da compreensibilidade e da manutenibilidade do código.

```typescript
export class RuntimeConfiguration {
    private static _environment: Environment | null = null;

    static initialize(environment: Environment) {
        this._environment = environment;
    }

    static get getConfig(): Environment {
        return this._environment!;
    }

    static get gatewayUrl(): string {
        return this._environment?.gatewayUrl || '';
    }

    static get appVersion(): string {
        return this._environment?.appVersion || '0.0.0';
    }

    static get production(): boolean {
        return this._environment?.production || false;
    }

    static get auth0(): { domain: string; clientId: string; audience: string } {
        return this._environment?.auth0 || { domain: '', clientId: '', audience: '' };
    }

    static get portal(): string {
        return this._environment?.portal || '';
    }
}
```

Figura 4.24: Classe *Runtime Configuration*

Numa primeira iteração, procedeu-se à integração da lógica de carregamento no `app.component.ts`, assegurando que a aplicação apenas expunha as suas funcionalidades após o carregamento das configurações.

```typescript
  async ngOnInit() {
    await this.carregaConfiguracoesRuntime();

    this.translationService.loadTranslations(enLang, ptLang);

    this.carregaTitulos();

    await this.carregaPermissoes();

    await this.carregaAparencia();

    this.configuracoesCarregadas = true;

    this.changeDetectorRef.detectChanges();

    this.notificacoesHubService.novaConexaoHub();
  }

  private carregaConfiguracoesRuntime() {
    return this.configService.initialize();
  }

```

Figura 4.25: Lógica inicial do `app-component.ts`

Todavia, esta solução demonstrou-se inadequada, dado que o *provider* de autenticação era

injetado no `app.module.ts` e dependia da classe `RuntimeConfiguration`, cujos campos não detinham ainda valores.

De forma a superar a limitação identificada, o serviço de configuração passou a ser injetado aquando da inicialização da aplicação, por meio do *provideAppInitializer*. Este mecanismo possibilitou a execução antecipada do método `initialize`, responsável por obter os dados do `config.json` e preparar o *provider* de autenticação antes da fase de arranque completo da aplicação.

```
101        provideAppInitializer(async () => {
102            await inject(ConfigurationService).initialize();
103        }),
```

Figura 4.26: Injeção do serviço de configuração

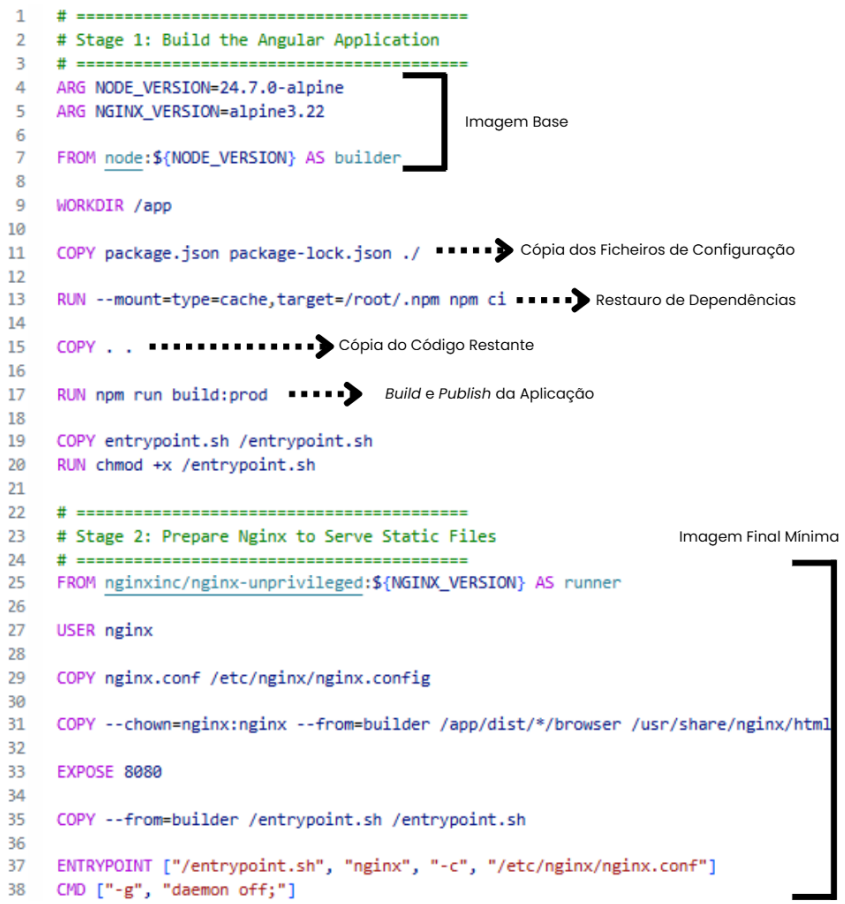
Adicionalmente, considerando que os pedidos HTTP requerem os *headers* com dados de autenticação e autorização, e que o *interceptor* responsável por esta tarefa também é injetado no `app.module.ts`, tornou-se necessário criar uma configuração inicial vazia para o *provider* de autenticação. Esta abordagem garantiu que o pedido inicial ao ficheiro `config.json` pudesse ser executado sem dependências circulares.

```
109        provideHttpClient(withInterceptors([
110            authHttpInterceptorFn,
111            httpRequestInterceptor
112        ])),
113        //Esta configuração será substituída pela configuração dinâmica no ConfigurationService,
114        //contudo é necessário fornecer valores padrão para evitar erros na inicialização do authHttpInterceptorFn
115        provideAuth0({
116            domain: '',
117            clientId: '',
118            authorizationParams: {
119                redirect_uri: window.location.origin,
120                audience: '',
121                scope: 'openid email profile'
122            },
123            httpInterceptor: {
124                allowedList: []
125            }
126        })
```

Figura 4.27: *Providers* para o `HttpClient`

Containerization

O *Dockerfile* associado a esta solução foi desenvolvido em conformidade com as recomendações constantes na documentação oficial, de forma a assegurar a sua manutenção e compatibilidade com as práticas consolidadas [37]. Na figura abaixo, uma vez mais, identificam-se as camadas idealizadas em 3.2.2 e a sua aplicabilidade prática.




```
# ========================================
# Stage 1: Build the Angular Application
# ========================================
ARG NODE_VERSION=24.7.0-alpine
ARG NGINX_VERSION=alpine3.22

FROM node:${NODE_VERSION} AS builder

WORKDIR /app

COPY package.json package-lock.json ./

RUN --mount=type=cache,target=/root/.npm npm ci

COPY . .

RUN npm run build:prod

COPY entrypoint.sh /entrypoint.sh
RUN chmod +x /entrypoint.sh

# ========================================
# Stage 2: Prepare Nginx to Serve Static Files
# ========================================
FROM nginxinc/nginx-unprivileged:${NGINX_VERSION} AS runner

USER nginx

COPY nginx.conf /etc/nginx/nginx.config

COPY --chown=nginx:nginx --from=builder /app/dist/*/browser /usr/share/nginx/html

EXPOSE 8080

COPY --from=builder /entrypoint.sh /entrypoint.sh

ENTRYPOINT ["/entrypoint.sh", "nginx", "-c", "/etc/nginx/nginx.conf"]
CMD ["-g", "daemon off;"]
```

Figura 4.28: *Dockerfile Frontend*

Na camada da Imagem Base define-se a versão específica de *Node.js* (`24.7.0-alpine`) e cria-se um *alias* para que possa ser referenciada mais tarde.

Na camada de Restauro de Dependências começa por copiar somente os ficheiros de dependências (`package.json` ou `package-lock.json`). Caso não existam alterações nestes ficheiros, a camada do `npm ci` (*clean install*) será reutilizada. Os argumentos definidos nesse comando permitem utilizar a *cache* entre *builds*.

Segue-se a cópia do restante código e a execução do `build:prod` definido no `package.json` (`ng build -configuration production`).

Por fim, efetua-se a cópia do `entrypoint.sh` para que seja incluído na imagem final. Este *script* é utilizado para efetuar as substituições dos valores no ficheiro `config.json`.

```
#!/bin/sh

CONFIG_PATH="/usr/share/nginx/html/assets/config/config.json"
TMP_CONFIG_PATH="/usr/share/nginx/html/assets/config/config.tmp.json"

if [ -f "$CONFIG_PATH" ]; then
  envsubst < "$CONFIG_PATH" > "$TMP_CONFIG_PATH"
  mv "$TMP_CONFIG_PATH" "$CONFIG_PATH"
fi

exec "$@"
```

Figura 4.29: *Script entrypoint.sh*

De forma sucinta o *script* obtém o diretório do ficheiro `config.json` e um diretório para um ficheiro temporário utilizado para efetuar as substituições. Caso encontre o `config.json` utiliza o comando `envsubst` para substituir todos os *placeholders* pelos valores das variáveis de ambiente. O resultado é escrito para o ficheiro temporário e depois movido para o ficheiro original.

4.2.5 Repositório Orquestrador

A última componente da solução corresponde ao repositório orquestrador, cuja principal função é centralizar e normalizar a gestão de todos os serviços que compõem a solução. Este repositório funciona como *single source of truth*, reunindo num único ponto todos os ficheiros de orquestração e configuração necessários ao *deployment* do sistema em ambientes de cliente. Denote-se que este será utilizado apenas na primeira instalação e, eventualmente, na adição de novos serviços à solução do cliente.

Figura 4.30: Estrutura de Pastas do Repositório

Estrutura e Organização

Na raiz do repositório encontram-se definidos o ficheiro `compose.yaml`, de caráter global, e o ficheiro `README.md`, que contém as instruções de utilização. Estão igualmente presentes um ficheiro de configuração auxiliar (`servers.json`) e dois ficheiros de variáveis de ambiente (`.env`): um associado especificamente ao `compose.yaml` e outro, designado `portainer.env`, que agrega todas as variáveis de ambiente do repositório e se destina à sua importação direta no *Portainer*. As mesmas variáveis encontram-se também replicadas nos diretórios correspondentes a cada serviço, possibilitando a inicialização isolada de serviços. Embora esta abordagem não esteja prevista para ambientes de produção, pode ser utilizada em contextos de desenvolvimento. Para além destes elementos, o repositório inclui um conjunto de diretórios, cada qual representando individualmente um dos serviços disponibilizados.

O ficheiro global de orquestração contém por predefinição os serviços comuns e obrigatórios (no caso da amostra selecionada, todos os serviços o são) e pode ser estendido através do `include`.

```yaml
include:
  - Tese.Auth0/compose.yaml
  - Tese.Base/compose.yaml
  - Tese.Erp/compose.yaml
  - Tese.Gateway/compose.yaml
  - Tese.Frontend/compose.yaml

services:
  pgadmin_tese:
    image: dpage/pgadmin4:latest
    container_name: pgadmin_tese
    environment:
      PGADMIN_DEFAULT_EMAIL: ${PGADMIN_EMAIL}
      PGADMIN_DEFAULT_PASSWORD: ${PGADMIN_PASSWORD}
      PGADMIN_REPLACE_SERVERS_ON_STARTUP: true
    ports:
      - ${PGADMIN_PORT}
    networks:
      - pg-network
    depends_on:
      db_tese_base:
        condition: service_healthy
    user: root
    configs:
      - source: servers.json
        target: /pgadmin4/servers.json

networks:
  tese-backend:
    driver: bridge
  pg-network:
    driver: bridge
  frontend-gateway:
    driver: bridge

configs:
  servers.json:
    file: ./servers.json
```

Figura 4.31: `compose.yaml` Global

De enfatizar a inclusão do serviço auxiliar `pgAdmin` especificamente no *compose* global. Este é disponibilizado como *container* com configuração automática através do ficheiro `servers.json` que inclui os *servers* configurados. Este mecanismo permite interagir graficamente com as bases de dados `PostgreSQL`, simplificando a análise e diagnóstico durante o *deployment*.

```
{
  "Servers": {
    "1": {
      "Name": "Base de dados do serviço Tese.Base",
      "Group": "Servers",
      "Host": "db_tese_base",
      "Port": 5432,
      "MaintenanceDB": "postgres",
      "Username": "mydbuser",
      "SSLMode": "prefer"
    }
  }
}
```

Figura 4.32: Ficheiro `servers.json`

É também neste *compose* que se encontram declaradas as redes utilizadas:

- **tese-backend**: Utilizada para todos os serviços *backend*;
- **pg-network**: Utilizada pelos serviços cuja base de dados se encontra num *container*. Esta é uma rede privada exclusiva para comunicação com a base de dados;
- **frontend-gatewa**y: Rede exclusiva de comunicação entre o monólito de *frontend* e o *gateway*

Cada pasta representa um microsserviço e é, na generalidade, composta por um ficheiro `.env` onde se armazenam as credenciais e parâmetros customizados e por um `compose.yaml`. Abaixo detalham-se alguns exemplos:

Tese.Erp

Por depender de bases de dados externas dos clientes (mantendo o uso de SQL Server), este serviço não foi containerizado ao nível da persistência. O seu compose dedica-se essencialmente à configuração de variáveis e parâmetros de ligação.

```
services:
  tese_erp:
    container_name: tese_erp
    image: plexhub.azurecr.io/tese_erp:${ERP_TAG}
    ports:
      - ${ERP_PORT}
    restart: unless-stopped
    environment:
      - ConnectionStrings__PlexHub=server=${SS_HOST};database=${SS_DB};uid=${SS_UID};password=${SS_PASSWORD};MultipleActiveResultSets=true;TrustServerCertificate=True;
      - ConnectionStrings__Cache=${REDIS_HOST}
      - Messaging__Provider=${MB_PROVIDER}
      - Messaging__Host=${MB_HOST}
      - Messaging__Port=${MB_PORT}
      - Messaging__VirtualHost=${MB_VIRTUAL_HOST}
      - Messaging__Username=${MB_USERNAME}
      - Messaging__Password=${MB_PASSWORD}
      - Auth__Provider=${AP_PROVIDER}
      - Auth__Authority=${AP_AUTHORITY}
      - Auth__Audience=${AP_AUDIENCE}
      - Auth__MetadataAddress=${AP_METADATA_ADDRESS}
      - Auth__RequireHttpsMetadata=${AP_REQUIRE_HTTPS_METADATA}
      - Integration__UserAgent=${INTEGRATION_USER_AGENT}
      - Integration__BaseAddress=${INTEGRATION_BASE_ADDRESS}
    networks:
      - tese-backend
```

Figura 4.33: `compose.yaml` do `Tese.Erp`

Tese.Base

No caso do Tese.Base foi necessário desenvolver um *compose* mais completo que inclui três *containers*:

- Base de dados PostgreSQL

```yaml
services:
  db_tese_base:
    image: postgres:18-alpine
    container_name: db_tese_base
    environment:
      POSTGRES_USER: ${PG_USER}
      POSTGRES_PASSWORD: ${PG_PASSWORD}
      POSTGRES_DB: ${PG_DB}
    ports:
      - ${DB_PORT}
    volumes:
      - postgres_data:/var/lib/postgresql/data
      - ./config:/etc/postgresql
      - ./backup:/backup
    healthcheck:
      test:
        - CMD-SHELL
        - pg_isready -U ${PG_USER} -d ${PG_DB}
      interval: 10s
      timeout: 5s
      retries: 5
    restart: unless-stopped
    networks:
      - pg-network
```

Figura 4.34: Extrato do `compose.yaml` correspondente à base de dados PostgreSQL

O serviço `db_tese_base` corresponde à instância principal da base de dados *PostgreSQL* utilizada pelo microsserviço. É definido a partir da imagem oficial do *Postgres* e parametrizado através de variáveis de ambiente, que permitem configurar o utilizador, palavra-passe e base de dados inicial. Inclui ainda a declaração de volumes, que garantem a persistência dos dados, bem como diretórios locais dedicados à configuração e a cópias de segurança. De forma a assegurar que os restantes serviços apenas iniciam após a base estar operacional, foi configurado um mecanismo de *healthcheck* baseado no comando `pg_isready`.

- *Backup* da Base de dados

```yaml
backup_db_service_tese_base:
  image: postgres:18-alpine
  container_name: backup_db_service_tese_base
  environment:
    PGHOST: db_tese_base
    PGUSER: ${PG_USER}
    PGPASSWORD: ${PG_PASSWORD}
    PGDATABASE: ${PG_DB}
    BACKUP_SCHEDULE: 0 0 * * *
  volumes:
    - ./backup:/backup
    - ./backup-script.bash:/backup-script.bash
  entrypoint:
    - /bin/bash
    - /backup-script.bash
  depends_on:
    db_tese_base:
      condition: service_healthy
  networks:
    - pg-network
```

Figura 4.35: Extrato do `compose.yaml` correspondente ao *backup* da base de dados PostgreSQL

O serviço `backup_db_service_tese_base` foi concebido para executar automaticamente cópias de segurança da base de dados. Utiliza igualmente a imagem oficial do *Postgres*, mas é configurado para se conectar ao serviço `db_tese_base` através de variáveis de ambiente que definem *host*, utilizador, palavra-passe e base de dados. O agendamento do backup é parametrizado através de uma expressão cron, interpretada pelo *script* de inicialização fornecido via *bind mount*. Este *script* é definido como *entrypoint* do *container*, garantindo a execução imediata do processo de cópia aquando do arranque do serviço. O *container* está condicionado à disponibilidade da base de dados, sendo declarado em `depends_on` com base no estado de saúde do serviço `db_tese_base`. Finalmente, partilha o diretório de backups com o *container* da base de dados, assegurando que os ficheiros gerados permanecem acessíveis no sistema anfitrião.

```bash
#!/bin/bash

calculate_sleep_time() {
    local schedule="$1"
    local hour=$(echo "$schedule" | cut -d' ' -f2)
    local minute=$(echo "$schedule" | cut -d' ' -f1)

    local now=$(date +%s)
    local today_target=$(date -d "today ${hour}:${minute}" +%s)
    local tomorrow_target=$(date -d "tomorrow ${hour}:${minute}" +%s)

    if [ $now -lt $today_target ]; then
        echo $((today_target - now))
    else
        echo $((tomorrow_target - now))
    fi
}

echo "Starting backup service..."
echo "Schedule: $BACKUP_SCHEDULE"

BACKUP_FILE="/backup/backup_$(date +%Y%m%d_%H%M%S).dump"

echo "Performing initial backup..."
pg_dump -h "$POSTGRES_HOST" -U "$POSTGRES_USER" -d "$POSTGRES_DB" -F c -f "$BACKUP_FILE"

if [ $? -eq 0 ]; then
    echo "Backup completed: $BACKUP_FILE"
else
    echo "Backup failed!"
fi

find /backup -name "backup_*.dump" -mtime +7 -delete

while true; do
    sleep_seconds=$(calculate_sleep_time "$BACKUP_SCHEDULE")
    echo "Next backup in $sleep_seconds seconds ($(date -d "+${sleep_seconds} seconds"))"

    sleep "$sleep_seconds"

    BACKUP_FILE="/backup/backup_$(date +%Y%m%d_%H%M%S).dump"

    echo "Performing scheduled backup..."
    pg_dump -h "$POSTGRES_HOST" -U "$POSTGRES_USER" -d "$POSTGRES_DB" -F c -f "$BACKUP_FILE"

    if [ $? -eq 0 ]; then
        echo "Backup completed: $BACKUP_FILE"
    else
        echo "Backup failed!"
    fi

    find /backup -name "backup_*.dump" -mtime +7 -delete
done
```

Figura 4.36: *Backup Script*

O *scrip* de *backup* foi desenvolvido com o objetivo de automatizar a criação periódica de cópias de segurança da base de dados. Quando o *container* é iniciado é efetuado um backup inicial imediato, garantindo desde logo um ponto de recuperação válido.

Para tal, é utilizado o comando `pg_dump`, que exporta a base de dados num formato comprimido (`-F c`) e grava o ficheiro no diretório partilhado `/backup`, acrescentando ao nome um carimbo temporal.

Após a execução inicial, o serviço entra num ciclo contínuo. Através da função `calculate_sleep_time` determina o número de segundos restante até à próxima execução (tendo por base a expressão definida na variável de ambiente `BACKUP_SCHEDULE`, mantendo-se inativo até atingir o intervalo de tempo definido (o *script* é executado diariamente à meia noite, conforme estabelecido no`compose.yaml`).

Quando chega a hora agendada, é gerado um novo ficheiro de *backup*, novamente com um identificador temporal único, e é executado pg_dump para capturar o estado atual da base de dados. A cada execução, é também realizada a rotação de *backups*, eliminando automaticamente todos os ficheiros mais antigos que sete dias, de forma a otimizar o espaço ocupado em disco.

Este mecanismo garante que existem sempre cópias de segurança recentes disponíveis, com um equilíbrio entre fiabilidade (através da redundância temporal dos ficheiros) e sustentabilidade do armazenamento.

- Aplicação

```yaml
tese_base:
  container_name: tese_base
  image: plexhub.azurecr.io/tese_base:${BASE_TAG}
  depends_on:
    db_tese_base:
      condition: service_healthy
  ports:
    - ${BASE_PORT}
  restart: unless-stopped
  environment:
    - ConnectionStrings__PlexHub=server=localhost;Host=db_tese_base;Port=5432;Database=${PG_DB};Username=${PG_USER};Password=${PG_PASSWORD};
    - Messaging__Provider=${MB_PROVIDER}
    - Messaging__Host=${MB_HOST}
    - Messaging__VirtualHost=${MB_VIRTUAL_HOST}
    - Messaging__Username=${MB_USERNAME}
    - Messaging__Password=${MB_PASSWORD}
    - Auth__Authority=${AP_AUTHORITY}
    - Auth__Audience=${AP_AUDIENCE}
    - Auth__MetadataAddress=${AP_METADATA_ADDRESS}
    - Auth__RequireHttpsMetadata=${AP_REQUIRE_HTTPS_METADATA}
  networks:
    - tese-backend
    - pg-network
```

Figura 4.37: Extrato do `compose.yaml` correspondente à aplicação

O serviço `tese_base` representa a aplicação em si, construída e publicada previamente no *Azure Container Registry*. O seu arranque depende diretamente da disponibilidade da base de dados. A configuração do *container* é feita através de um conjunto alargado de variáveis de ambiente, que definem a *connection* string da base de dados, parâmetros de *RabbitMQ* e credenciais de autenticação. Este modelo segue a convenção do .NET, que mapeia variáveis de ambiente com duplo *underscore* para as secções equivalentes no `appsettings.json`. O serviço expõe a sua porta de comunicação através de uma variável definida no ficheiro `.env` e integra-se em duas redes distintas: uma dedicada ao tráfego da base de dados e outra orientada à comunicação com os restantes serviços do *backend*.

4.3 Configuração dos Ambientes de Cliente

Evidenciados todos os componentes desenvolvidos para a presente solução, importa agora demonstrar de que forma estes são utilizados no processo de instalação. O procedimento de *deployment* foi significativamente simplificado através da injeção direta do repositório orquestrador no *Portainer*, recorrendo à criação de uma *stack* baseada no *GitHub Repository*.

A opção por este mecanismo não é meramente técnica, mas estratégica: elimina a necessidade de copiar manualmente os ficheiros `compose.yaml` e garante que as configurações disponibilizadas aos clientes estão sempre em conformidade com a versão mais recente validada no repositório. Ou seja, reduz-se o risco de discrepâncias entre o ambiente de desenvolvimento e o ambiente de produção, assegurando consistência e reprodutibilidade.

O primeiro passo consiste, assim, na criação da *stack*, especificando o repositório e o caminho para o ficheiro *compose.yaml* principal. Dado que todos os repositórios da organização PlexIT são privados, é necessário gerar credenciais de acesso adequadas, nomeadamente um *Token* exclusivo com permissões restritas de leitura sobre o repositório orquestrador:

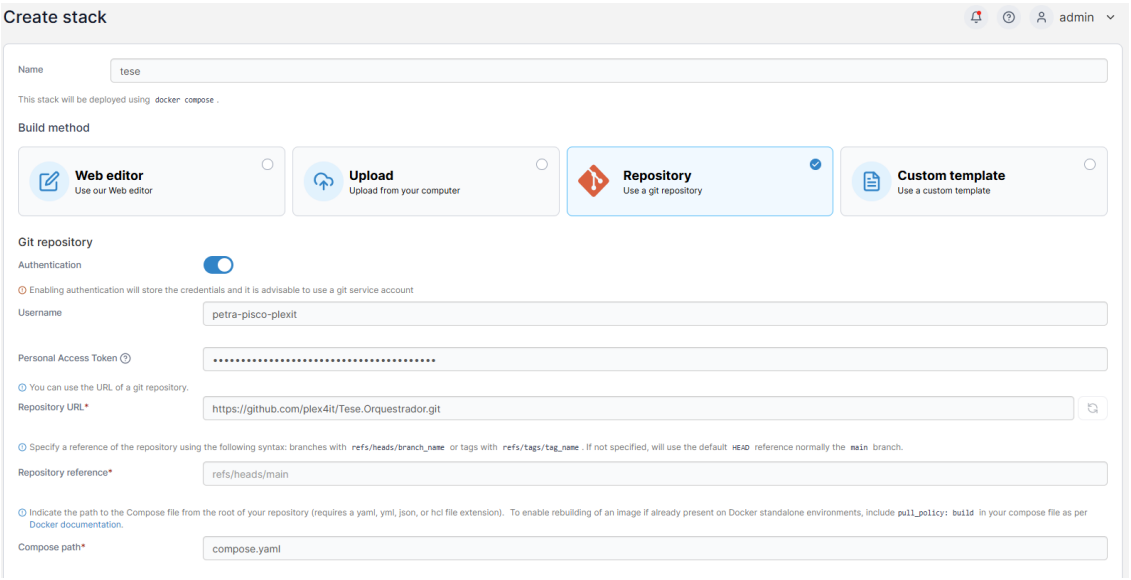


Figura 4.38: Criação da *stack* no *Portainer*

Na configuração das variáveis de ambiente, o Portainer permite importar diretamente a partir de um ficheiro. Neste caso, recorreu-se ao ficheiro `portainer.env`, disponibilizado no próprio repositório orquestrador e já preparado com todas as variáveis necessárias:

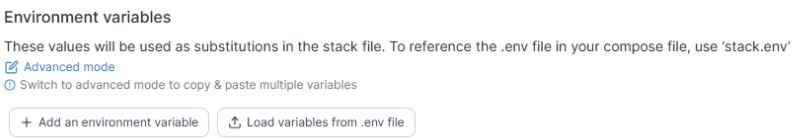


Figura 4.39: Função de importação de variáveis de ambiente para a *stack*

O resultado desta importação é a disponibilização automática das variáveis na stack:

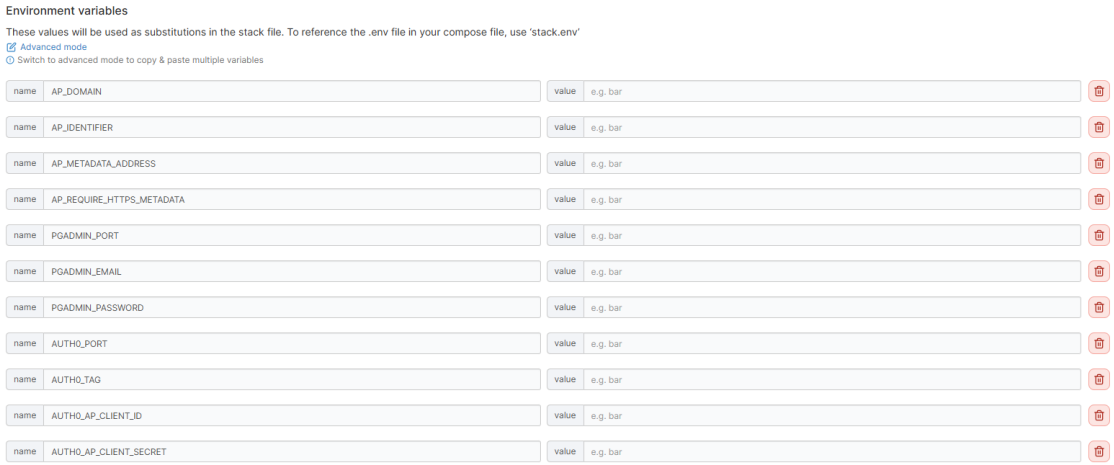


Figura 4.40: Variáveis de Ambiente importadas para a *stack*

Após a substituição de valores específicos, basta acionar o botão *Deploy Stack* para concluir o processo.

Capítulo 5

Avaliação da Solução

O presente capítulo dedica-se à avaliação da solução implementada, procurando verificar de forma sistemática e imparcial se os artefactos desenvolvidos correspondem aos requisitos previamente definidos e se cumprem os objetivos do projeto. Para tal, a solução foi aplicada num ambiente de testes controlado que reproduz, com elevada fidelidade, as condições reais de produção, permitindo analisar a sua robustez, portabilidade e capacidade de automatização.

5.1 Casos de Teste

Foram definidos um conjunto de casos de teste que visam validar a solução proposta e comprovar a exequibilidade técnica da abordagem.

Cada caso de teste foi delineado para reproduzir situações concretas do processo de *deployment* do *PlexHub*, permitindo observar a robustez dos mecanismos implementados, a consistência do versionamento e a fiabilidade do sistema perante falhas ou cenários de parametrização dinâmica.

No decorrer da presente secção apresentam-se diversas figuras demonstrativas do processo de testes efetuado, de modo a garantir maior transparência e compreensão dos procedimentos adotados.[1].

[1]Denote-se que, em algumas capturas de ecrã, foram ocultados valores sensíveis (como credenciais de autenticação, dados de clientes ou endereços IP de máquinas privadas - mesmo o acesso a estes estar protegido por VPN e *username / password*). Em certos casos optou-se mesmo por não apresentar a captura, uma vez que a ocultação total inviabilizaria a sua utilidade demonstrativa. Adicionalmente, nas imagens da aplicação instalada surge a indicação de ligação não segura, situação decorrente da pendência de emissão de certificados pela equipa de infraestruturas, não comprometendo a validade da demonstração.

Tabela 5.1: Cenários de Teste

Casos de Teste	Objetivo	Procedimento	Métricas	Resultados Esperados
CT1	Validar que a imagem é gerada automaticamente	Efetuar um *push* no `main`, aguardar conclusão da execução da pipeline e efetuar *pull* na VM	Tempo Total de Execução, erros nos *logs*	Pipeline concluída em menos de 5 minutos, sem falhas. Inlcusão da imagem *docker* na instalação não afeta o funcionamento
CT2	Validar consistência do versionamento entre ambientes	Aguardar a execução da pipeline e verificar o nome da imagem gerada, aceder ao ACR e garantir que coincidem	Os nomes coincidem	Coerência de versionamento
CT3	Validar falhas na Pipeline	Forçar um erro na pipeline, verificar que se falhar na *build* da imagem, a *tag* não é criada	*Logs* da pipeline, notificações de falha de execução	Falha detetada, sem impacto no ambiente do cliente
CT4	Validar parametrização	Alterar porta / tema nos ficheiros respetivos (`.env`), relançar os `containers`	Serviço inclui as modificações	Alterações refletidas sem ter que se gerar nova versão da imagem *docker*
CT5	Validar base de dados *containerizada*	Iniciar o `container`, verificar que a base de dados é criada, efetuar as migrações utilizando a ferramenta interna	Base de dados é criada, as migrações são aplicadas sem erros, a aplicação estabelece comunicação com a base de dados	Aplicação funciona corretamente
CT6	Validar base de dados externa	Definir a *connection* string no ficheiro `.env`	Conexão estabelecida	Aplicação funciona corretamente
CT7	Validar recuperação automática	Parar os *containers* e reiniciá-los	Serviços reiniciam sem inconsistências	Estado Recuperado

5.1.1 Caso de Teste 1 - Geração automática da Imagem

Para a realização deste teste optou-se por utilizar o repositório `Tese.Base`.

O primeiro passo consistiu em introduzir uma mudança no código de forma a despoletar o *GitHub Workflow*. No caso, adicionou-se um *Controller* vazio apenas para o efeito:

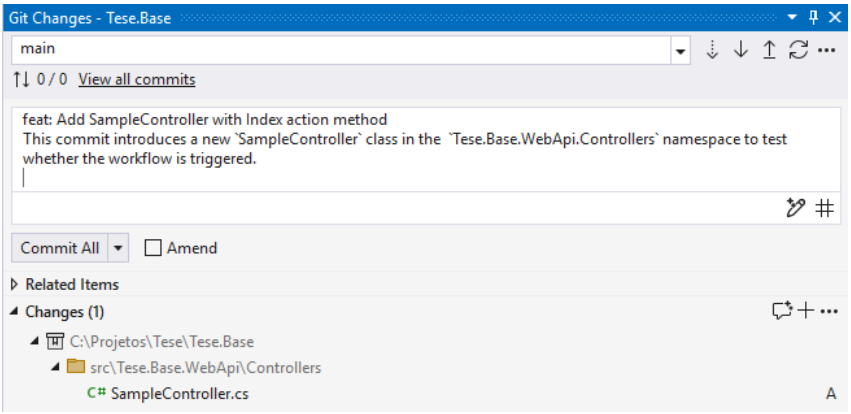


Figura 5.1: *Commit* que introduz a mudança

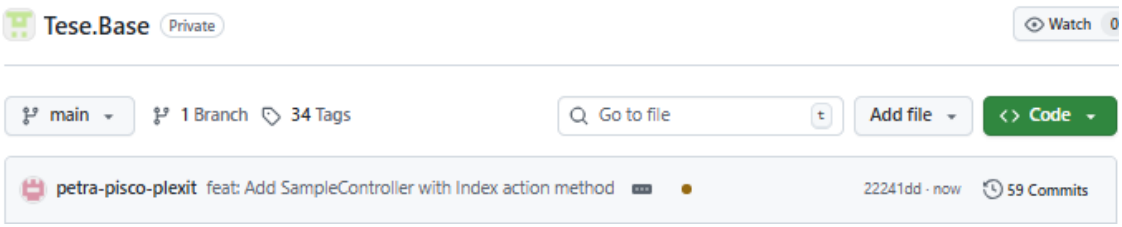


Figura 5.2: Demonstração de que o *push* despoletou o workflow

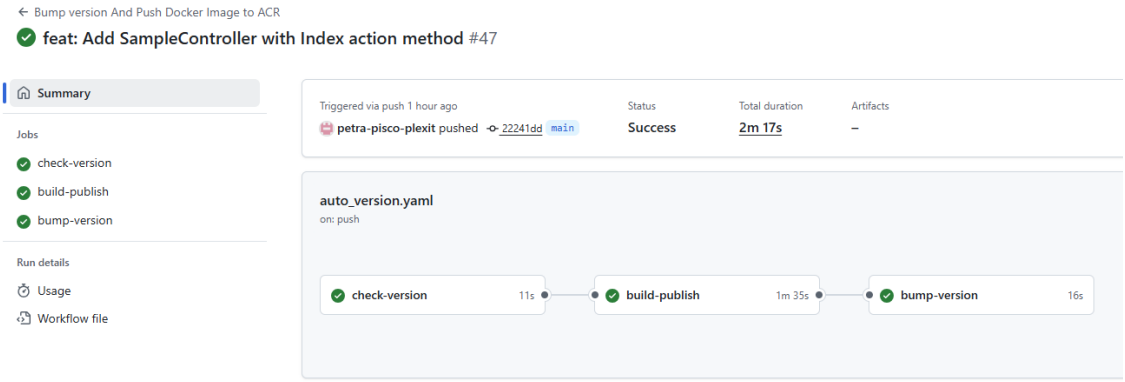


Figura 5.3: Demonstração da execução bem sucedida do *workflow*

A figura acima demonstra já a conformidade com o primeiro *outcome* definido: a execução do *workflow* decorreu em apenas dois minutos e dezassete segundos, consideravelmente abaixo do tempo limite estabelecido de cinco minutos.

Por fim, acedeu-se ao *Portainer* no ambiente de instalação e alterou-se a variável de ambiente que contém a versão da aplicação:

Figura 5.4: *Tag* do Projeto inicialmente aplicada

Figura 5.5: *Tag* do Projeto após mudança

Efetuou-se o *pull* e o *redeploy*, com a opção de obter novamente as *imagens*:

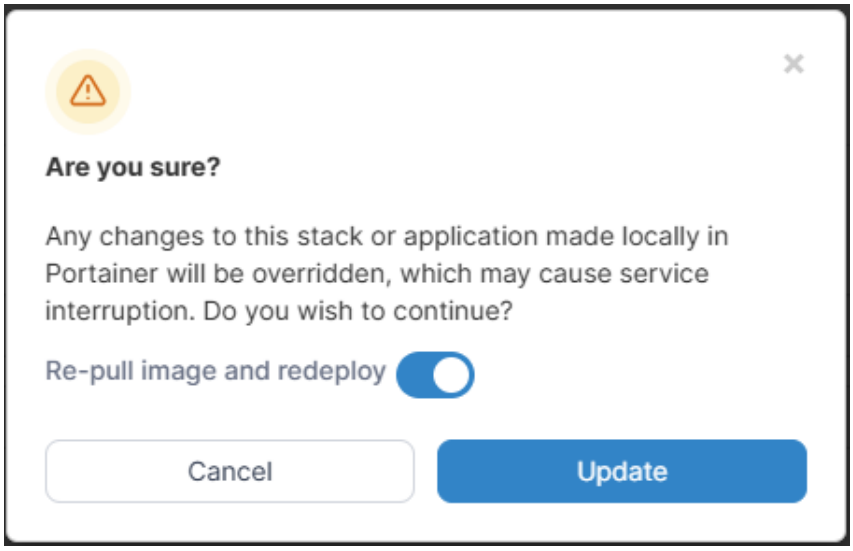


Figura 5.6: *Pull* e *Redeploy* da Stack

Name	State Filter	Quick Actions	Stack	Image
db_tese_base	healthy		orquestrador	postgres:18-alpine
tese_gateway	running		orquestrador	plexhub.azurecr.io/tese_gateway:0.5.0
tese_auth0	running		orquestrador	plexhub.azurecr.io/tese_auth0:0.1.0
tese_frontend	running		orquestrador	plexhub.azurecr.io/tese_frontend:0.19.0
tese_erp	running		orquestrador	plexhub.azurecr.io/tese_erp:0.4.0
pgadmin_tese	running		orquestrador	dpage/pgadmin4:latest
tese_base	running		orquestrador	plexhub.azurecr.io/tese_base:1.25.0

Figura 5.7: Demonstração da ocorrência do *pull* da imagem do `Tese.Base`

Por fim, demonstra-se a aplicação funcional:

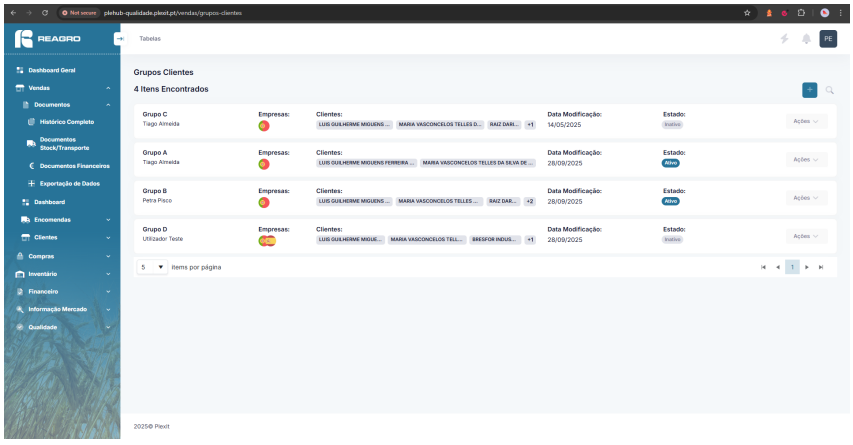


Figura 5.8: Demonstração da aplicação funcional com a versão `1.25.0` do `Tese.Base`

5.1.2 Caso de Teste 2 - Versionamento consistente

Este caso de teste reutilizou o fluxo demonstrado no anterior, tendo-se que, da sua execução, de acordo com os *logs* do *workflow* gerou-se a seguinte versão:

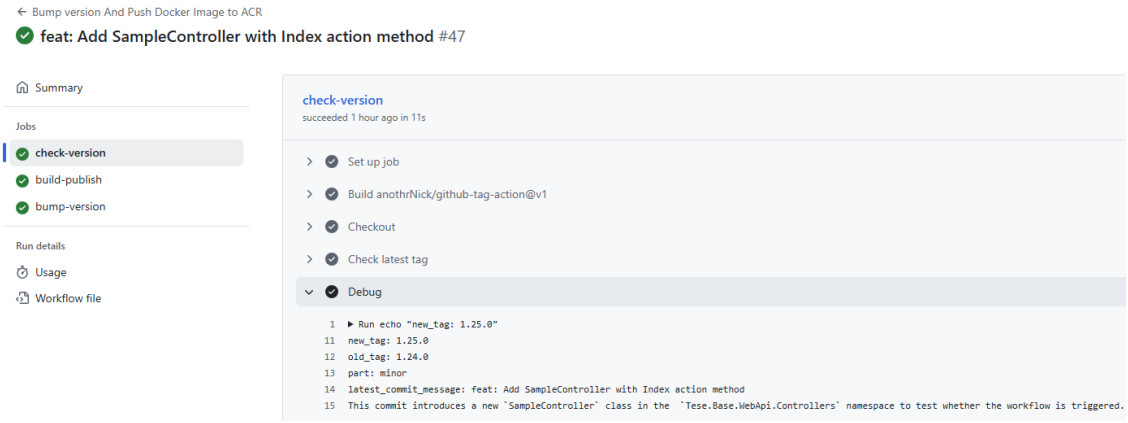


Figura 5.9: *Logs* da execução do *workflow*

Esta mesma versão foi registada nas *tags* do repositório:

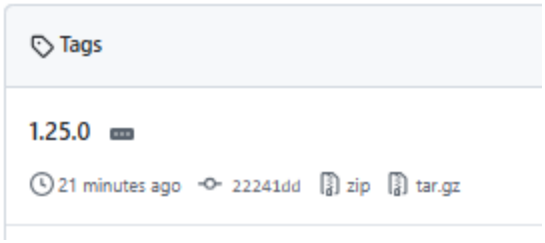


Figura 5.10: Tag *criada no repositório*

Acedendo ao *Azure Container Registry*, é possível verificar que a imagem foi adicionada e as versões são coincidentes:

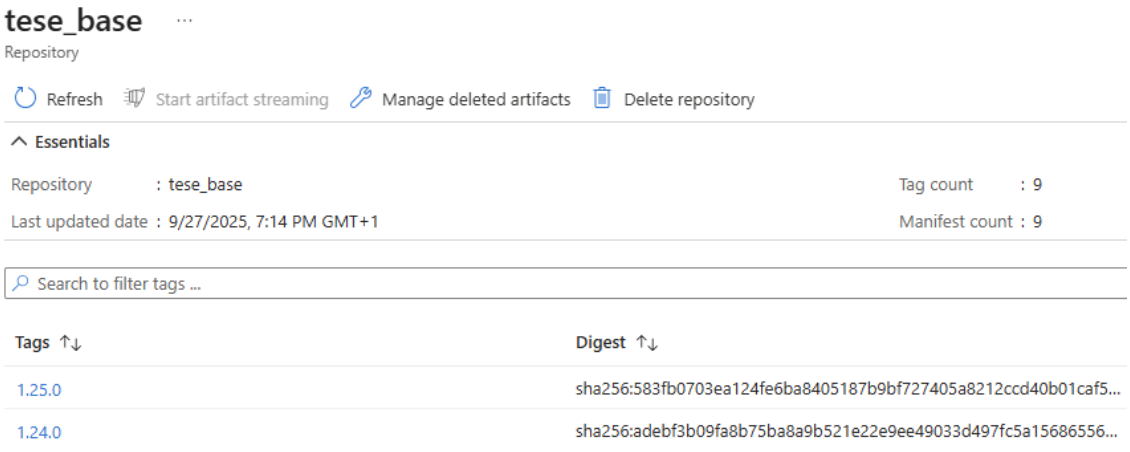


Figura 5.11: Imagem publicada no *Azure Container Registry*

5.1.3 Caso de Teste 3 - Comportamento de Falha da Pipeline

No decorrer da construção deste cenário utilizou-se o repositório `Tese.Frontend`.

O primeiro passo coincidiu com a introdução de uma falha no sistema, no caso comentou-se um pedido `HTTP GET` que é essencial para o funcionamento da aplicação, dado ser utilizado num grande número de componentes.

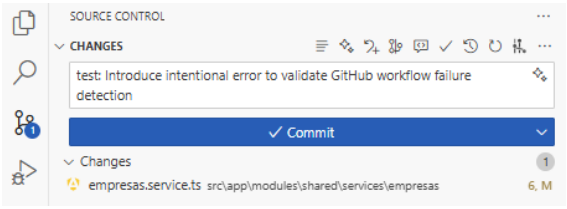


Figura 5.12: *Commit* que introduz o erro

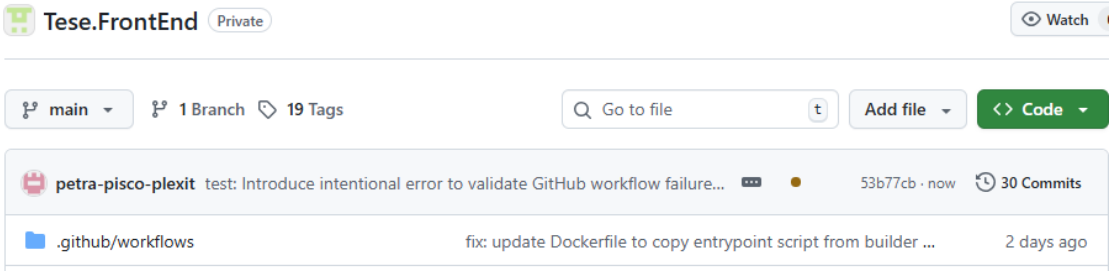


Figura 5.13: Demonstração de que o *push* despoletou o *workflow*

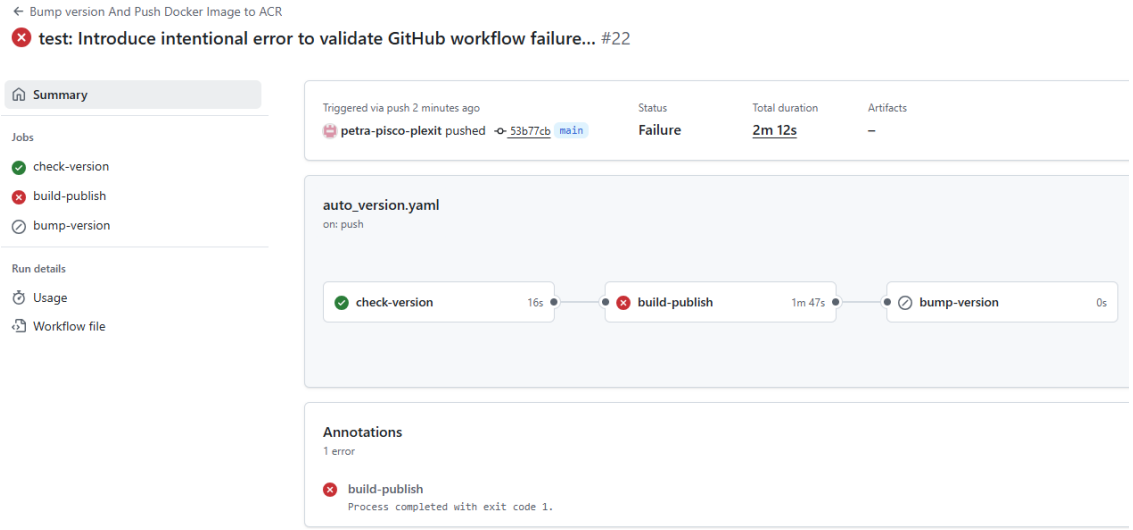


Figura 5.14: Demonstração da execução falhada do *workflow*

Denote-se que a ação que verifica a nova versão determinou que se criaria a tag **0.20.0**:

Figura 5.15: *Logs* do *workflow*

Contudo esta não foi nem incluída nas *tags* do repositório, nem adicionada ao ACR:

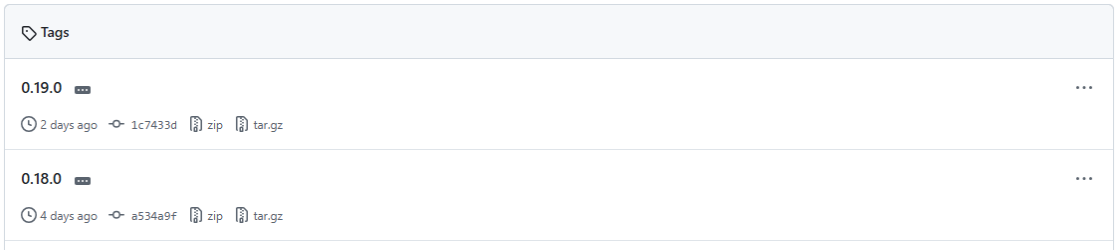


Figura 5.16: *Tags* inalteradas do repositório `Tese.FrontEnd`

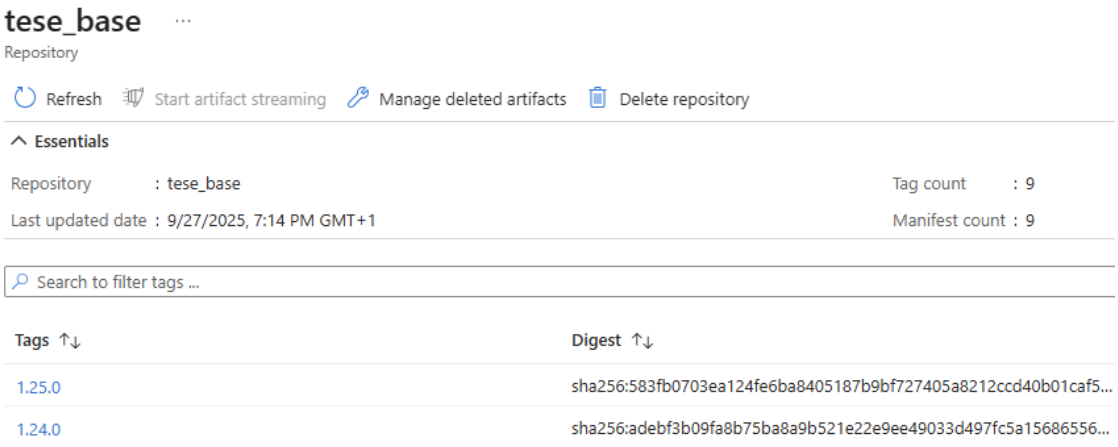


Figura 5.17: Versões inalteradas no ACR

Considerando que não foi criada qualquer nova versão no ACR e que na instalação da VM as novas versões das imagens apenas são integradas manualmente (é preciso atualizar a variável de ambiente correspondente), conclui-se que não ocorreu qualquer impacto sobre esta.

5.1.4 Caso de Teste 4 - Parametrização Dinâmica

Para a execução deste caso de teste recorreu-se ao `Tese.FrontEnd`, tendo-se, no caso, modificado as cores do tema.

Inicialmente apresentavam os seguintes valores que correspondem às utilizadas na imagem 5.8:

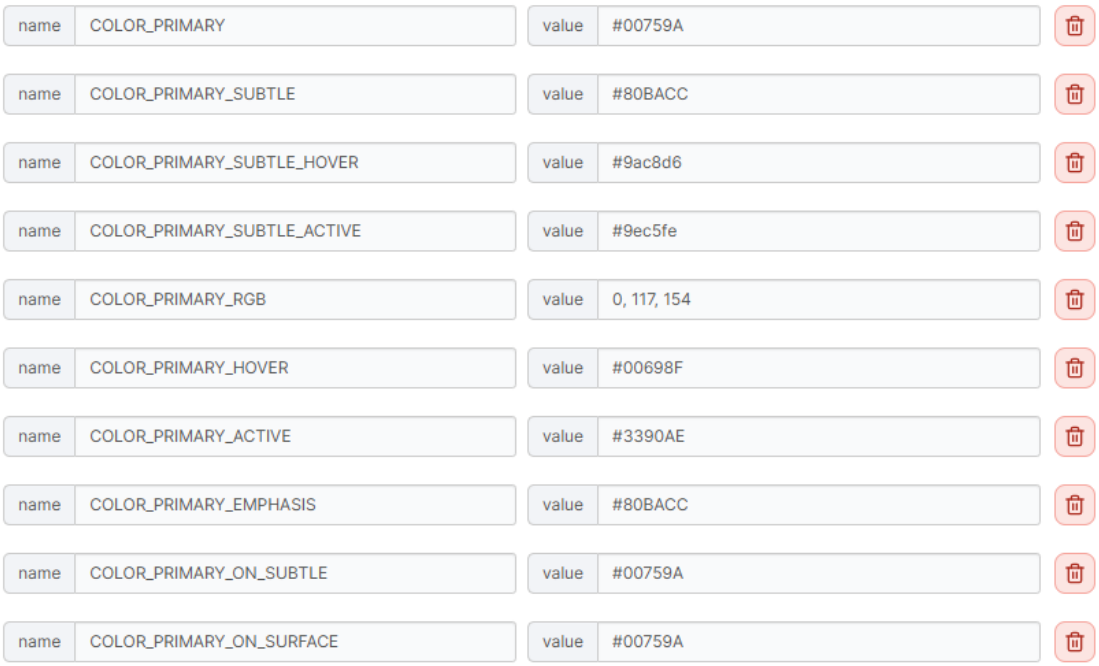


Figura 5.18: Configuração Inicial das Cores

Estes foram substituídos para:

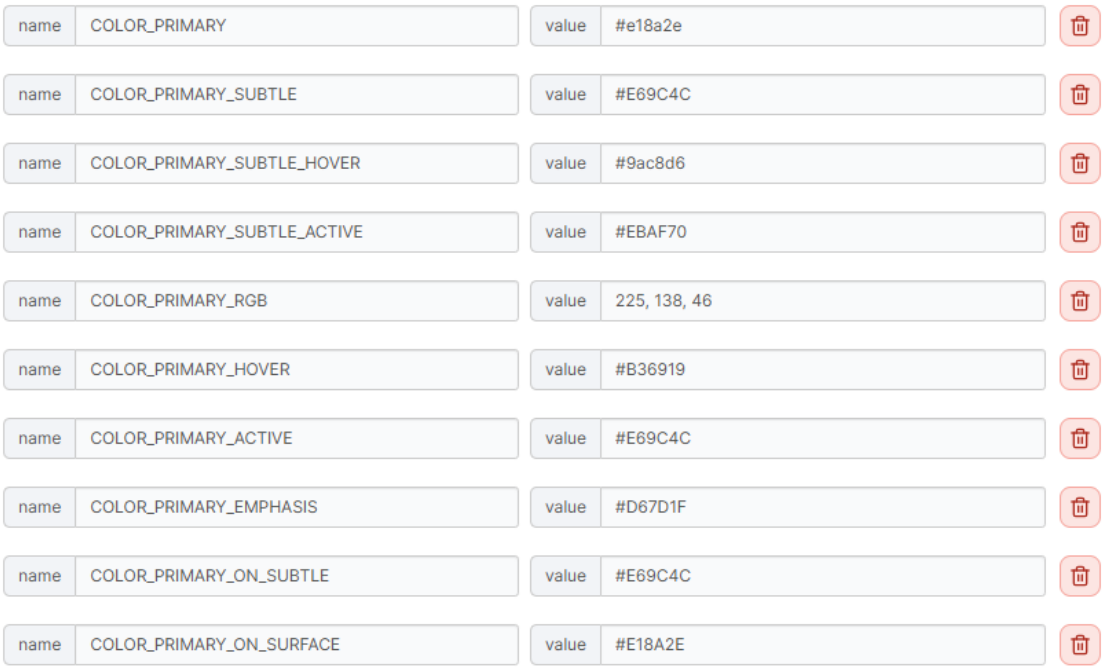


Figura 5.19: Configuração das Cores utilizando um Tema distinto

Efetuou-se novamente o *pull* e *redeploy*, no caso sem a função *Re-pull image and redeploy* ativa, dado que não foram introduzidas mudanças que precisem de atualizar as imagens base. Tendo-se que:

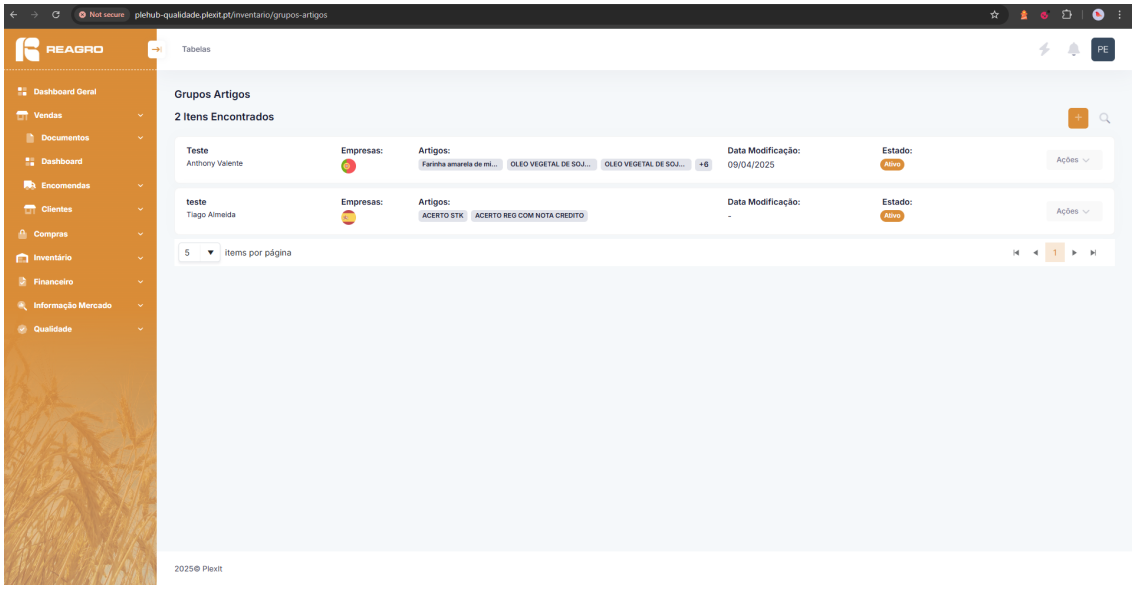


Figura 5.20: Aplicação após a mudança de cores do tema

Conforme é evidente, as cores da aplicação foram devidamente modificadas, cumprindo as métricas estabelecidas.

5.1.5 Caso de Teste 5 - Base de Dados em Container Funcional

O primeiro passo para a efetuação deste teste corresponde à inicialização do *container* que contém a base de dados e demonstração de que este executa com sucesso:

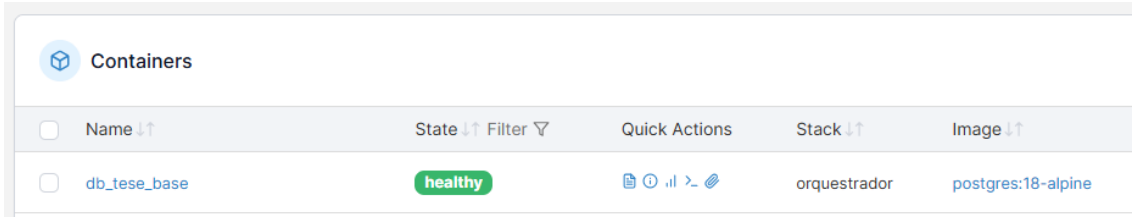


Figura 5.21: Demonstração do *Container* da Base de dados em Funcionamento

Acedendo diretamente ao *pgAdmin* é ainda possível observar que o *server* para a base de dados Tese.Base foi criado com sucesso através das configurações do servers.json:

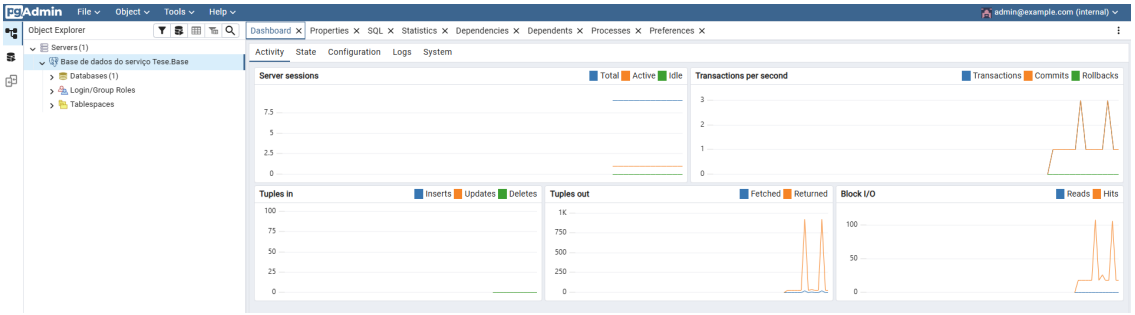


Figura 5.22: Estado inicial do *pgAdmin*

Seguiu-se a modificação da *connection string* usada na ferramenta de migrações (*Tese.Base.Migrations*) para que executasse as migrações *EF Core Migrations* diretamente para a base de dados:

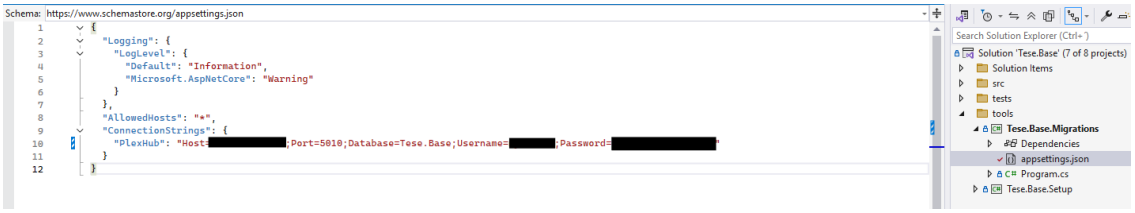


Figura 5.23: Modificação do `appsettings.json` da Ferramenta de migrações

Através dos *logs* de execução é possível verificar que começou por criar a base de dados Tese.Base, seguindo para a migração das tabelas:

```
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (490ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      CREATE DATABASE "Tese.Base";
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (33ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      CREATE TABLE "__EFMigrationsHistory" (
          "MigrationId" character varying(150) NOT NULL,
          "ProductVersion" character varying(32) NOT NULL,
          CONSTRAINT "PK___EFMigrationsHistory" PRIMARY KEY ("MigrationId")
      );
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (93ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT EXISTS (
    SELECT 1 FROM pg_catalog.pg_class c
    JOIN pg_catalog.pg_namespace n ON n.oid=c.relnamespace
    WHERE n.nspname='public' AND
          c.relname='__EFMigrationsHistory'
)
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (25ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT "MigrationId", "ProductVersion"
      FROM "__EFMigrationsHistory"
      ORDER BY "MigrationId";
info: Microsoft.EntityFrameworkCore.Migrations[20402]
      Applying migration '20250905172913_1.20.0-Migration'.
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (23ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      CREATE TABLE "AcoesRapidasComponentes" (
```

Figura 5.24: Base de Dados `Tese.Base` criada e todas as ex

Na captura de ecrã abaixo comprova-se a execução bem sucedida da ferramenta:

```
      CREATE INDEX "IX_Utilizadores_TipoUtilizadorId" ON "Utilizadores" ("TipoUtilizadorId");
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (20ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      CREATE INDEX "IX_UtilizadoresEntidades_CodigoEmpresa" ON "UtilizadoresEntidades" ("CodigoEmpresa");
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (15ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      CREATE INDEX "IX_UtilizadoresEntidades_UtilizadorId" ON "UtilizadoresEntidades" ("UtilizadorId");
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (15ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      INSERT INTO "__EFMigrationsHistory" ("MigrationId", "ProductVersion")
      VALUES ('20250905172913_1.20.0-Migration', '7.0.18');

C:\Projetos\Tese\Tese.Base\tools\Tese.Base.Migrations\bin\Debug\net6.0\Tese.Base.Migrations.exe (process 51260) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Figura 5.25: Demonstração da conclusão das migrações com sucesso

Novamente acedendo ao *pgAdmin* denota-se que a base de dados foi criada, bem como as suas tabelas:

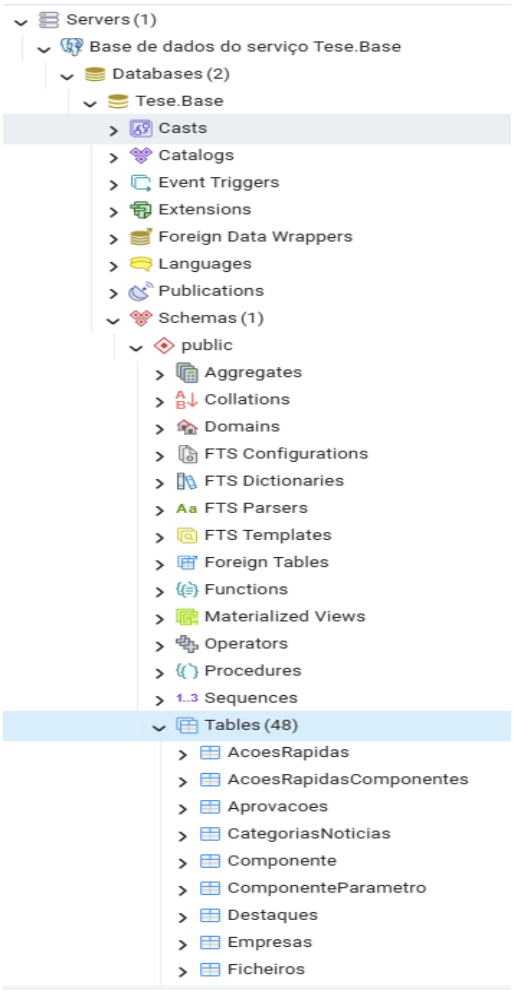


Figura 5.26: Demonstração das Tabelas persistidas

Dado que ainda não foi executado o *setup* dos dados, efetuando uma *query* à tabela de Tipos de Utilizadores verifica-se que esta se encontra vazia, como esperado:

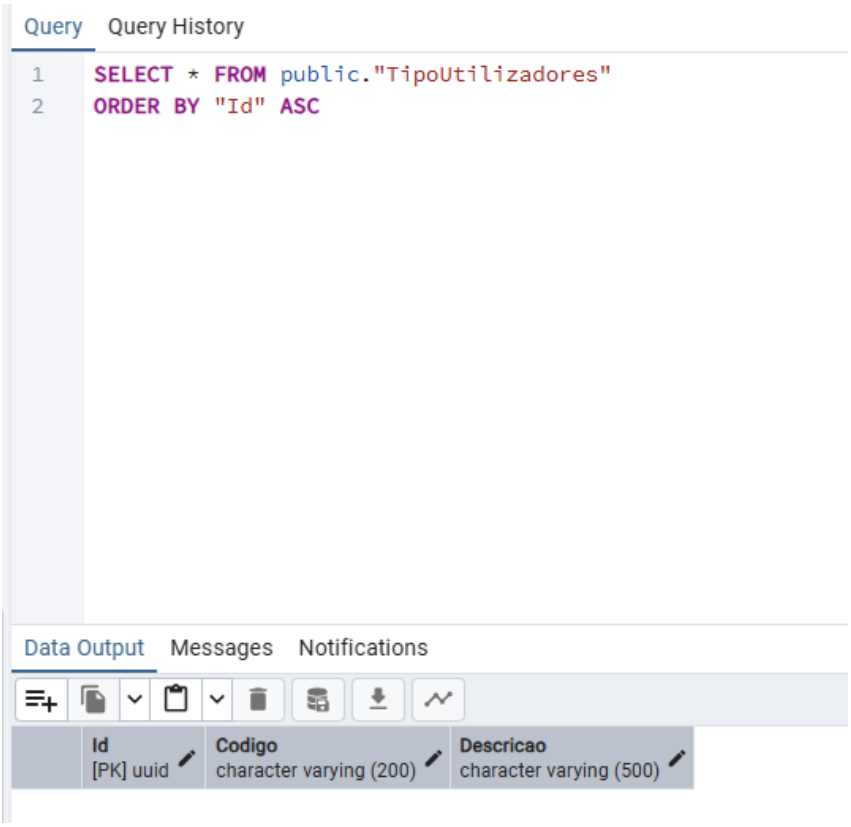


Figura 5.27: Tabela de TiposUtilizadores vazia

Foi executada a ferramenta `Tese.Base.Setup` para a introdução dos dados, tendo-se que:

```
[11:14:02 INF] Seeding finished. Added: 152, Skipped: 0
[11:14:02 INF] Successfully completed setup: MenusItensTraducoes
[11:14:02 INF] Running setup: Empresas
[11:14:02 INF] Seeding finished. Added: 2, Skipped: 0
[11:14:02 INF] Successfully completed setup: Empresas
[11:14:02 INF] Running setup: AcoesRapidasComponentes
[11:14:03 INF] Seeding finished. Added: 7, Skipped: 0
[11:14:03 INF] Successfully completed setup: AcoesRapidasComponentes
[11:14:03 INF] All setups completed.
[11:14:03 INF] Application terminated.

C:\Projetos\Tese\Tese.Base\tools\Tese.Base.Setup\bin\Debug\net8.0\Tese.Base.Setup.exe (process 12412) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Figura 5.28: Demonstração do sucesso da execução da ferramenta de *Setup*

Verificou-se o preenchimento da tabela:

Query Query History

```
1  SELECT * FROM public."TipoUtilizadores"
2  ORDER BY "Id" ASC
```

Data Output Messages Notifications

	Id [PK] uuid	Codigo character varying (200)	Descricao character varying (500)
1	19605a99-d1c1-444b-a29a-075f0fb87...	Cliente	Cliente
2	26757b77-9f23-4e1b-9432-a63e7f4b5...	Interno	Interno
3	33d193ac-0caf-49cf-a87a-e4fd429004...	Fornecedor	Fornecedor
4	7c3b6d11-c876-4bc4-8cd5-79452b799...	Administrador	Administrador
5	dd9b7f31-7023-4671-8bee-3c01fa29e1...	ClienteFornecedor	Cliente/Fornecedor

Figura 5.29: Demonstração do preenchimento da tabela de Tipos de Utilizador

Até este ponto foi possível assegurar a criação da base de dados e o sucesso na migração das tabelas e dos seus dados iniciais, tendo-se cumprido duas das três métricas estabelecidas.

Para verificar a comunicação correta da aplicação com a base de dados (última métrica definida), optou-se por efetuar a criação de um atalho (Ação Rápida) na aplicação. Denote-se que, como o utilizador não contém qualquer atalho definido, a criação de um irá gerar primeiramente um menu de atalhos para o utilizador (tabela MenusAcoesRapidas apresentada de seguida).

Para tal, torna-se preponderante apresentar o estado inicial da tabela:

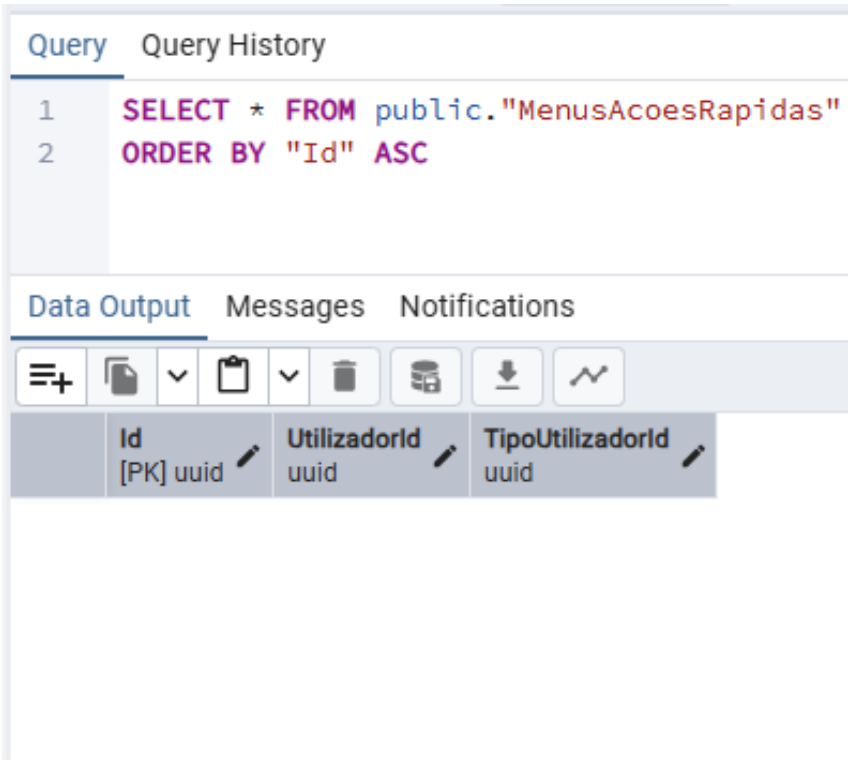


Figura 5.30: Tabela de Ações Rápidas Vazia

Acedendo à aplicação, comprova-se igualmente a ausência de dados:

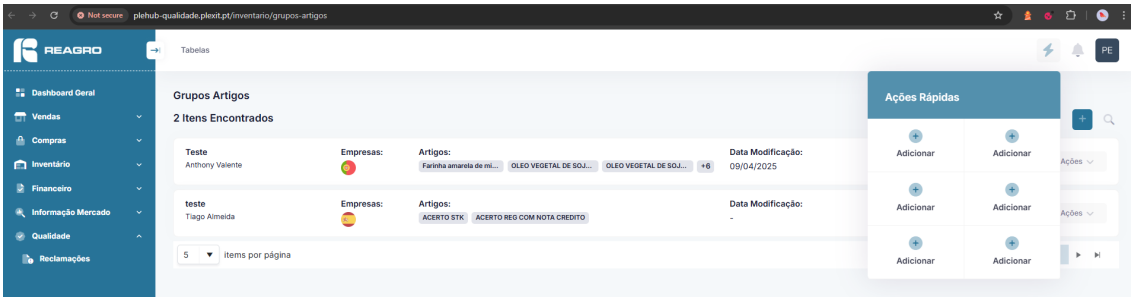


Figura 5.31: Ações Rápidas vazias na UI

Procede-se à criação de um atalho para a adição de notícias:

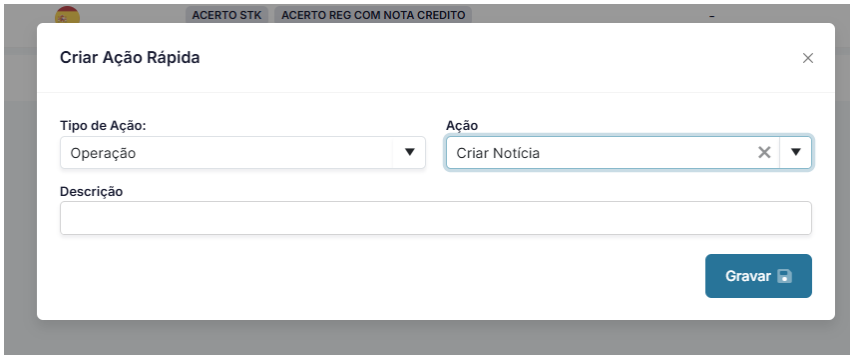


Figura 5.32: Preenchimento do Formulário de Criação de uma Ação Rápida

Verifica-se que a alteração foi adicionada à interface visual:

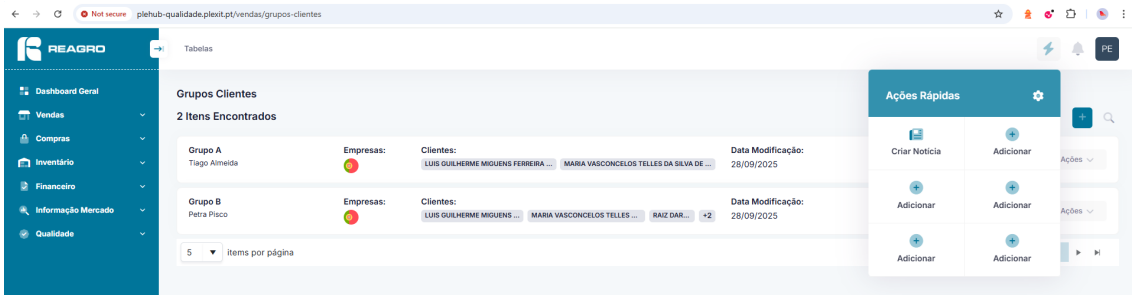


Figura 5.33: Demonstração da Ação Rápida criada na UI

Por fim, voltou a executar-se a *query* para listar os Menus de Ações Rápidas, tendo-se que, foi criada uma nova entrada para o utilizador autenticado:

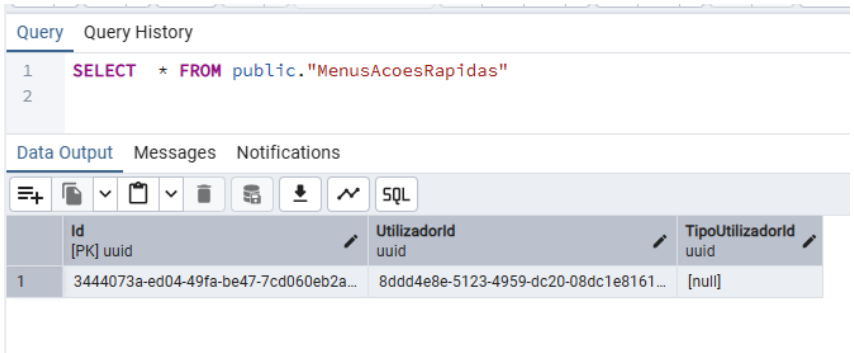


Figura 5.34: Registo

5.1.6 Caso de Teste 6 - Base de Dados Externa Funcional

Acedendo ao *Portainer*, mais especificamente à *Stack* em que se incluem todos os *containers* da aplicação adicionaram-se entradas para as variáveis de ambiente relativas à *connection string*, sendo estas `SS_HOST`, `SS_DB`, `SS_UID` e `SS_PASSWORD`:

Figura 5.35: Configuração da Conexão à Base de Dados SQL Server

Acedendo à aplicação e verificando os pedidos HTTP efetuados, bem como os dados apresentados, é possível determinar que a comunicação com a base de dados está a decorrer com normalidade:

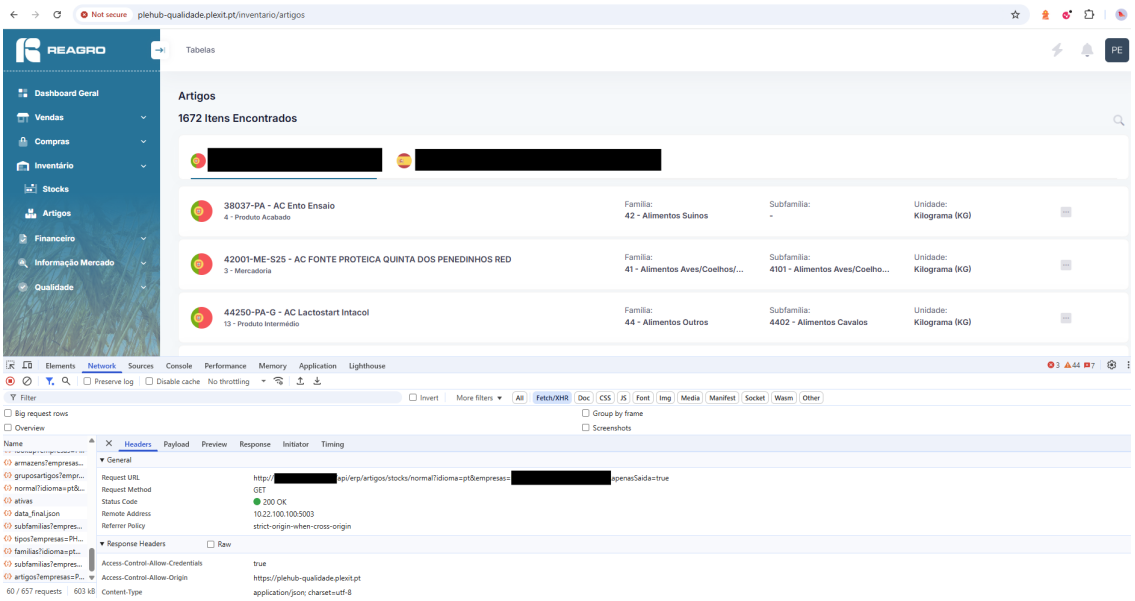


Figura 5.36: Demonstração da execução de pedidos diretos ao *Tese.Erp*

5.1.7 Caso de Teste 7 - Recuperação Automática dos Containers

Para a execução do presente teste, o primeiro passo configurou a paragem da execução da *Stack*, conforme:

Figura 5.37: Demonstração da inativação da *Stack*

Acedendo ao *url* da aplicação, verificou-se que esta deixou de ter a aplicação acessível tal como era pretendido:

Figura 5.38: Aplicação não funciona pois a *stack* está inativa

Procedeu-se,então, à reativação da *stack*, tendo-se que os *containers* retomaram a sua execução:

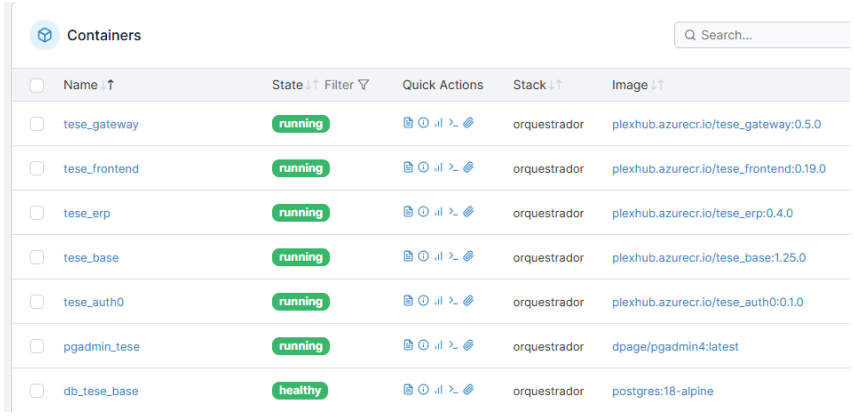


Figura 5.39: Demonstração dos *Containers* a executar

Acedendo novamente à aplicação, continuam a ser visíveis todos os menus do utilizador (configurados na base de dados do `Tese.Base`) e é possível ler os Grupos de Clientes (provenientes da base de dados do `Tese.Erp`):

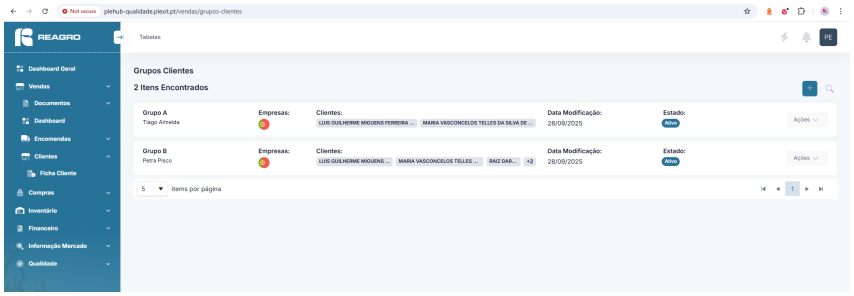


Figura 5.40: Demonstração do normal funcionamento da Aplicação

Comprova-se, assim, que a aplicação permanece funcional mesmo após a paragem e retoma da execução de todos os *containers*.

5.2 Resultados

Tabela 5.2: Resultados obtidos por Caso de Teste

Casos de Teste	Objetivo	Resultado Esperado	Resultado Obtido	Observações
CT1	Validar que a imagem é gerada automaticamente	Pipeline concluída em menos de 5 minutos, sem falhas, Inclusão da imagem na instalação não afeta o funcionamento	Sucesso	Processo parcialmente automatizado (deve definir-se a versão da imagem manualmente nas variáveis de ambiente da *stack* e efetuar-se o pull e *re-deploy*)
CT2	Validar consistência do versionamento entre ambientes	Coerência de versionamento	Sucesso	Processo totalmente automatizado
CT3	Validar comportamento de falhas na Pipeline	Falha detetada, sem impacto no ambiente do cliente	Sucesso	*Logs* indicam a falha, o *logout* é efetuado, não deixando a conexão ao ACR aberta. Processo totalmente automatizado.
CT4	Validar parametrização	Alterações refletidas sem ter que se gerar nova versão da imagem *docker*	Sucesso	Processo manual, requer a modificação das variáveis e o *re-deploy* do *container*
CT5	Validar base de dados *containerizada*	Aplicação funciona corretamente	Sucesso	Processo parcialmente automatizado, requer execução das migrações manual e na instalação requer a configuração das variáveis de ambiente
CT6	Validar base de dados externa	Aplicação funciona corretamente	Sucesso	Processo parcialmente automatizado, requer execução manual das migrações e na instalação requer a configuração das variáveis de ambiente
CT7	Validar recuperação automática	Estado recuperado	Sucesso	Processo totalmente automatizado

A aplicação da solução desenvolvida a um ambiente de testes controlado revelou que todos os casos de teste foram concluídos com sucesso. A automatização demonstrou capacidade de gerar imagens *docker*, aplicar versionamento, configurar ambientes e recuperar *containers* sem intervenção manual significativa.

Contudo, foi identificado que algumas operações, especialmente relacionadas com migrações de bases de dados ou configuração de bases de dados externas, ainda requerem supervisão manual para garantir integridade dos dados e compatibilidade com configurações específicas de cliente (portas, nome das bases de dados, etc.). Estes aspetos refletem as limitações apontadas na discussão crítica, evidenciando que, embora a solução maximize a automatização, certos processos dependem de intervenção humana controlada.

Globalmente, a avaliação demonstra que a automação implementada reduz significativamente o tempo de *deployment*, melhora a consistência entre ambientes e diminui a propensão a erros, mantendo, no entanto, atenção a situações de migração que exigem revisão manual.

5.3 Discussão Crítica

A execução dos testes em ambiente de demonstração comprovou a viabilidade técnica da solução, validando o seu funcionamento de ponta a ponta. Embora não tenham sido ainda totalmente integrados nos projetos reais da PlexIT, estes testes foram suficientes para demonstrar a conformidade com os requisitos definidos e a adequação da abordagem ao contexto da organização.

Um primeiro aspeto relevante foi a velocidade de execução das *pipelines*. O Caso de Teste 1 demonstrou que a geração automática de imagens e publicação no repositório ocorre em menos de cinco minutos, contrastando com o processo manual que exigia compilações locais demoradas e transporte manual de artefactos.

A consistência de versionamento verificada no Caso de Teste 2 reforça a confiança na solução, ao garantir que a *tag* gerada no *workflow* corresponde exatamente à publicada no *Azure Container Registry* e visível no repositório. O Caso de Teste 3 confirmou ainda que, perante falhas de *build*, não é criada qualquer nova versão. Este comportamento assegura que apenas imagens válidas ficam disponíveis para utilização, eliminando ambiguidades no processo de *deployment*. Simplifica, igualmente, o processo anterior em que sempre que se pretendia efetuar um *deploy* era necessário verificar as últimas *tags* publicadas em cada um dos repositórios e incrementar manualmente a versão.

O Caso de Teste 4 revelou-se particularmente impactante. A possibilidade de parametrizar dinamicamente variáveis de ambiente permitiu adotar uma única imagem universal para todos os clientes, eliminando a necessidade anterior de compilar e empacotar o projeto repetidas vezes para cada instalação a atualizar. Este resultado traduz-se numa redução drástica de esforço manual e numa simplificação profunda do processo, dado que as diferenças entre clientes passam a estar totalmente encapsuladas em ficheiros de configuração e nos seus próprios ambientes de produção.

Nos Casos de Teste 5 e 6, relativos às bases de dados, confirmou-se que a aplicação funciona corretamente tanto com instâncias em *containers* como com bases de dados externas, garantindo flexibilidade face às restrições técnicas específicas da PlexIT. Ainda assim, observou-se que as migrações de dados, embora funcionais, não são totalmente autónomas e exigem intervenção manual - é necessário especificar a *connection string* adequada e manualmente executar a ferramenta.

O Caso de Teste 7 validou a capacidade de recuperação automática dos *containers* após interrupção, demonstrando que os serviços regressam ao estado estável sem inconsistências.

Para além dos testes formais, foi igualmente recolhido *feedback* qualitativo da equipa de desenvolvimento durante a apresentação da solução. Entre os pontos mais destacados estiveram:

- a forte valorização do versionamento automático, que retira uma carga significativa de trabalho e reduz potenciais erros

- a perceção de que a instalação no servidor é extremamente facilitada, dado que basta configurar o ambiente uma única vez e, daí em diante, especificar apenas a versão da aplicação pretendida
- como limitação, a observação de que a solução ainda requer algum conhecimento técnico na sua utilização inicial, o que poderá ser uma barreira para perfis não especializados

Em síntese, a interpretação dos resultados evidencia ganhos claros em três dimensões: tempo (redução significativa face ao processo manual), consistência (eliminação de divergências de versionamento) e simplificação (uso de uma única imagem adaptável a todos os clientes). O *feedback* qualitativo obtido reforça esta perceção, funcionando como uma validação adicional do valor prático da solução no contexto da PlexIT.

Capítulo 6

Conclusão

O presente capítulo apresenta uma reflexão crítica sobre os resultados alcançados ao longo da dissertação. São analisados os objetivos inicialmente definidos, identificando quais foram cumpridos e em que medida. Em seguida, são discutidas as limitações da solução implementada, bem como propostas de evolução que poderão aumentar o seu valor em contextos futuros.

6.1 Objetivos Atingidos

O objetivo central deste trabalho consistia em conceber uma solução automatizada para o processo de *deployment* do PlexHub, reduzindo a dependência de tarefas manuais, aumentando a consistência entre ambientes e diminuindo o tempo de entrega. Nesse sentido, a utilização de *pipelines* CI/CD com *GitHub Actions*, aliada ao uso de *Docker* e *Docker Compose*, permitiu alcançar resultados relevantes:

- **O1 — Identificação de boas práticas de deployment de microsserviços:** A revisão sistemática da literatura e a análise de casos de estudo forneceram uma base sólida para selecionar as técnicas mais adequadas ao contexto da PlexIT.
- **O2 — Desenvolvimento de uma ferramenta para deployment remoto:** Foi criado um repositório orquestrador, centralizando a gestão de *containers* e configurações, o que simplificou significativamente o processo de instalação.
- **O3 — Automatização de versionamento e publicação de serviços:** Implementou-se um sistema de versionamento automático e publicação no *Azure Container Registry*, reduzindo falhas humanas e assegurando coerência entre diferentes ambientes.
- **O4 — Automatização de migrações de base de dados:** Este objetivo ficou apenas parcialmente atingido. Embora tenham sido validadas migrações em *containers* e testada a compatibilidade com bases de dados externas, a execução continua a exigir intervenção manual, não estando ainda integrada de forma autónoma no pipeline.

6.2 Limitações e Trabalho Futuro

Apesar dos resultados encorajadores, o projeto apresenta algumas limitações que indicam caminhos claros para melhorias futuras:

Atualmente, os mecanismos de *Continuous Integration / Continuous Delivery* focam-se essencialmente na construção e publicação de imagens. Um próximo passo lógico seria a introdução de *pipelines* mais completos, capazes de lidar com *pre-release tags*, automatizar a

criação de migrações a cada *push* e permitir testes mais frequentes no ambiente de qualidade. Esta evolução não só reduziria o tempo de validação, como também aproximaria o processo do ideal de *DevOps*.

As migrações de base de dados representam ainda um ponto de fricção no processo. Apesar de já existirem mecanismos para as aplicar em ambiente containerizado, a execução continua dependente de comandos manuais e de alguma supervisão técnica. Esta característica contrasta com o objetivo de tornar o ciclo de entrega mais linear e autónomo. No futuro, seria desejável que as migrações fossem tratadas como um componente natural do *deployment*, disparadas de forma controlada sempre que uma nova versão é disponibilizada. Para além de reduzir a intervenção humana, esta evolução permitiria reforçar a confiança no processo, garantindo que a aplicação e a base de dados permanecem alinhadas em todas as fases do ciclo de vida.

Os testes existentes concentram-se em regras arquiteturais, deixando descobertas dimensões críticas como integração entre serviços. Esta lacuna implica que, muitas vezes, a validação real só acontece na instalação junto ao cliente, o que aumenta o risco de múltiplas versões intermédias. Ao desenvolver esta solução pretendeu-se criar um processo mais linear e menos permissivo deste tipo de uso, para que se possa fomentar a necessidade de desenvolver mecanismos adequados como os referidos no parágrafo anterior.

A solução atual assenta em *Docker Compose*, adequado ao baixo volume de utilizadores previsto. No entanto, esta escolha limita a introdução de mecanismos avançados de tolerância a falhas, *load balancing* e *auto-scaling*. Tecnologias como *Docker Swarm* ou *Kubernetes*, estudadas no Estado da Arte, ofereceriam maior robustez e escalabilidade caso o projeto venha a ser adotado por um público mais amplo.

Apesar de já existirem múltiplas camadas de segurança — incluindo a utilização de máquinas virtuais isoladas, acesso via VPN, autenticação com utilizador e password, bem como o alojamento seguro em *Portainer* —, é recomendável reforçar as medidas de salvaguarda. Entre as possibilidades encontram-se a encriptação das bases de dados e volumes, a gestão centralizada de segredos (por exemplo, com *Azure Key Vault* ou *HashiCorp Vault*), entre outros. Estas práticas complementariam a arquitetura atual e aumentariam a confiança na solução em cenários de maior criticidade.

Em suma, a solução atinge o objetivo principal de reduzir a complexidade e o esforço humano no processo de entrega de software. Ainda assim, permanece como um ponto de partida: o seu valor está tanto na resolução imediata de problemas concretos quanto na abertura de caminho para a introdução progressiva de práticas *DevOps* mais maduras.

Bibliografia

[1] PlexIT. *PlexIT*. url: https://www.plexit.pt/.

[2] Cegid. *Cegid*. url: https://www.cegid.com/ib/pt/.

[3] Lianping Chen. «Microservices: Architecting for Continuous Delivery and DevOps». Em: *Proceedings - 2018 IEEE 15th International Conference on Software Architecture, ICSA 2018* (jul. de 2018), pp. 39–46. doi: 10.1109/ICSA.2018.00013.

[4] Instituto Politécnico do Porto. *Código de Boas Práticas e de Conduta do IPP*. url: https://www.ipp.pt/comunidade/menu-comunidade/missao-equidade-diversidade-inclusao/DespachoP.PORTOP0402020CodigodeBoasPraticasedeConduta.pdf/view.

[5] Conselho de Administração do IEEE. *IEEE - IEEE Code of Ethics*. url: https://www.ieee.org/about/corporate/governance/p7-8.html.

[6] Don Gotterbarn, Keith Miller e Simon Rogerson. «Software engineering code of ethics». Em: *Communications of the ACM* 40 (11 1997). issn: 00010782. doi: 10.1145/265684.265699. url: https://ethics.acm.org/code-of-ethics/software-engineering-code/.

[7] M. Salama, R. Bahsoon e N. Bencomo. «Chapter 11 - Managing Trade-offs in Self-Adaptive Software Architectures: A Systematic Mapping Study». Em: *Managing Trade-Offs in Adaptable Software Architectures*. Ed. por Ivan Mistrik et al. Boston: Morgan Kaufmann, 2017, pp. 249–297. isbn: 978-0-12-802855-1. doi: https://doi.org/10.1016/B978-0-12-802855-1.00011-3. url: https://www.sciencedirect.com/science/article/pii/B9780128028551000113.

[8] PRISMA Group. *PRISMA 2020 Flow Diagram Generator*. 2023. url: https://estech.shinyapps.io/prisma_flowdiagram/.

[9] Docker Inc. *Docker Documentation*. 2025. url: https://docs.docker.com/.

[10] Docker Inc. *Docker Compose Documentation*. 2025. url: https://docs.docker.com/compose/.

[11] GitHub. *GitHub Actions Documentation*. 2025. url: https://docs.github.com/en/actions.

[12] Xin Zhou et al. «A cross-company ethnographic study on software teams for DevOps and microservices: organization, benefits, and issues». Em: *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. Association for Computing Machinery, 2022, pp. 1–10. isbn: 9781450392266. doi: 10.1145/3510457.3513054. url: https://doi.org/10.1145/3510457.3513054.

[13] Nasreen Azad. «Understanding DevOps critical success factors and organizational practices». Em: *Proceedings of the 5th International Workshop on Software-Intensive Business: Towards Sustainable Software Business*. Association for Computing Machinery, 2023, pp. 83–90. isbn: 9781450393027. doi: 10.1145/3524614.3528627. url: https://doi.org/10.1145/3524614.3528627.

[14] Ricardo Amaro, Rúben Pereira e Miguel da Silva. «DevOps Metrics and KPIs: A Multivocal Literature Review». Em: *ACM Comput. Surv.* 56 (9 abr. de 2024). issn: 0360-0300. doi: 10.1145/3652508. url: https://doi.org/10.1145/3652508.

[15] Ricardo Amaro, Rúben Pereira e Miguel Mira da Silva. «Mapping DevOps capabilities to the software life cycle: A systematic literature review». Em: *Inf. Softw. Technol.* 177 (C jan. de 2025). issn: 0950-5849. doi: 10.1016/j.infsof.2024.107583. url: https://doi.org/10.1016/j.infsof.2024.107583.

[16] Henry van Merode. *Continuous Integration (CI) and Continuous Delivery (CD)*. Apress, 2023. doi: 10.1007/978-1-4842-9228-0.

[17] Alfredo Barrientos, Jesus Gonzalo Duran Huanca e Henry Albert Mayta Segovia. «Implementation of a Software Engineering Model with DevOps on Microsoft Azure». Em: *Proceedings of the 2023 8th International Conference on Information Systems Engineering*. Association for Computing Machinery, 2024, pp. 1–6. isbn: 9798400709173. doi: 10.1145/3641032.3641037. url: https://doi.org/10.1145/3641032.3641037.

[18] Mojtaba Shahin e M Ali Babar. «On the Role of Software Architecture in DevOps Transformation: An Industrial Case Study». Em: *Proceedings of the International Conference on Software and System Processes*. Association for Computing Machinery, 2020, pp. 175–184. isbn: 9781450375122. doi: 10.1145/3379177.3388891. url: https://doi.org/10.1145/3379177.3388891.

[19] Yiwen Wu et al. «Understanding and Predicting Docker Build Duration: An Empirical Study of Containerized Workflow of OSS Projects». Em: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, 2023. isbn: 9781450394758. doi: 10.1145/3551349.3556940. url: https://doi.org/10.1145/3551349.3556940.

[20] Kalvin Eng, Abram Hindle e Eleni Stroulia. «Patterns of multi-container composition for service orchestration with Docker Compose». Em: *Empirical Software Engineering* 29 (3 mai. de 2024), p. 65. issn: 1382-3256. doi: 10.1007/s10664-024-10462-8. url: https://link.springer.com/10.1007/s10664-024-10462-8.

[21] Işl Karabey Aksakalli et al. «Deployment and communication patterns in microservice architectures: A systematic literature review». Em: *J. Syst. Softw.* 180 (C out. de 2021). issn: 0164-1212. doi: 10.1016/j.jss.2021.111014. url: https://doi.org/10.1016/j.jss.2021.111014.

[22] Mandepudi Nobel Chowdary et al. «Automated Pipeline for the Deployment using OpenShift». Em: *Procedia Comput. Sci.* 215 (C jan. de 2022), pp. 220–229. issn: 1877-0509. doi: 10.1016/j.procs.2022.12.025. url: https://doi.org/10.1016/j.procs.2022.12.025.

[23] Jacob Nørbjerg e Yvonne Dittrich. «The never-ending story–How companies transition to and sustain continuous software engineering practices». Em: *J. Syst. Softw.* 213 (C jul. de 2024). issn: 0164-1212. doi: 10.1016/j.jss.2024.112056. url: https://doi.org/10.1016/j.jss.2024.112056.

[24] Zikuan Wang et al. «An Ethnographic Study on the CI of A Large Scale Project». Em: *ACM International Conference Proceeding Series* (abr. de 2024), pp. 287–297. doi: 10.1145/3639477.3639750. url: https://dl.acm.org/doi/10.1145/3639477.3639750.

[25] Alex Mwotil, Engineer Bainomugisha e Stephen G M Araka. «mira: an Application Containerisation Pipeline for Small Software Development Teams in Low Resource Settings». Em: *Proceedings of the Federated Africa and Middle East Conference on Software Engineering*. Association for Computing Machinery, 2022, pp. 31–38. isbn: 9781450396639. doi: 10.1145/3531056.3542769. url: https://doi.org/10.1145/3531056.3542769.

[26] Sara Aissat et al. «JuNo-OPS: A DevOps Framework for the Engineering of Digital Twins for Built Assets». Em: *Proceedings of the ACM/IEEE 27th International*

Conference on Model Driven Engineering Languages and Systems. Association for Computing Machinery, 2024, pp. 496–506. isbn: 9798400706226. doi: 10.1145/3652620.3688266. url: https://doi.org/10.1145/3652620.3688266.

[27] Daniel Sokolowski. «Deployment coordination for cross-functional DevOps teams». Em: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, 2021, pp. 1630–1634. isbn: 9781450385626. doi: 10.1145/3468264.3473101. url: https://doi.org/10.1145/3468264.3473101.

[28] Muhammad Imran et al. «Migration of CMSWEB cluster at CERN to Kubernetes: a comprehensive study». Em: *Cluster Computing* 24 (4 dez. de 2021), pp. 3085–3099. issn: 1386-7857. doi: 10.1007/s10586-021-03325-0. url: https://doi.org/10.1007/s10586-021-03325-0.

[29] Metronic. *Metronic*. url: https://keenthemes.com/metronic.

[30] Redis. *Redis*. url: https://redis.io/.

[31] Docker Inc. *Building best practices*. url: https://docs.docker.com/build/building/best-practices/.

[32] Tom Preston-Werner. *Semantic Versioning 2.0.0*. url: https://semver.org/.

[33] GitHub. *Workflow syntax for GitHub Actions - Concurrency*. GitHub, Inc. 2025. url: https://docs.github.com/en/actions/reference/workflows-and-actions/workflow-syntax#concurrency.

[34] Nick Sjostrom. *Github Tag Bump*. 2025. url: https://github.com/marketplace/actions/github-tag-bump.

[35] *Docker Login*. url: https://github.com/Azure/docker-login.

[36] *Customize containers in Visual Studio*. url: https://learn.microsoft.com/en-us/visualstudio/containers/container-build?view=vs-2022.

[37] *Containerize an Angular Application*. url: https://docs.docker.com/guides/angular/containerize/.