INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Desenvolvimento Serverless : Soluções, Impacto e Futuro

PEDRO LUCAS MOREIRA ROCHA
outubro de 2021

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Desenvolvimento Serverless: Soluções, Impacto e Futuro

Pedro Rocha

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

Orientador: Dr. Paulo Maio

Porto, Outubro de 2021

Dedicatória

Dedico esta dissertação ao meu avô Manuel e avó Adelina.

Resumo

Os processos de implantação de software tradicionais, i.e baseados num modelo *on-premises*, com um conjunto de servidores mais ou menos estático e bem definido tem cada vez mais vindo a revelar-se problemático quer seja pela rápida obsolescência dos dispositivos, quer seja pelo aumento da necessidade de equipamentos quer seja pelos custos de manutenção, ou outros.

Assim, para dar resposta a estes problemas, os processos de implantação tem evoluído e conduzido a novos modelos de serviço que visam então simplificar esta implantação. Deixou de ser necessário preparar e alocar espaço para máquinas físicas e equipamentos de rede, passando-se a alugar máquinas virtuais e a delegar a preocupação de gerir os recursos físicos que estas acarretam a uma outra entidade.

Apesar desta evolução, as empresas continuam ainda a pensar e a despender tempo útil nos seus sistemas em termos de servidores, deixando por vezes para segundo plano o que é efetivamente mais relevante para si em termos operacionais: que as suas aplicações executem a lógica que é suposto e que os seus dados estejam seguros e corretos.

Neste sentido, desde 2012, têm surgido estudos sobre o que significaria gerir sistemas e não servidores. A esta forma de trabalhar deu-se o nome de *serverless*, nome esse que não se deve ao facto de não existirem servidores mas sim ao facto de não se pensar tanto neles.

Por outro lado, esta evolução nos processos de implantação também tem impactado diretamente a forma como o próprio software é desenvolvido de modo a suportar estes novos modelos e paradigmas (e.g *serverless*).

Assim o principal objetivo desta dissertação é aferir este impacto e identificar e desenhar um conjunto de recomendações que devem ser adotadas no desenvolvimento de aplicações e respetivas arquiteturas com vista a eliminar ou minimizar as dificuldades que condicionam a adoção de *serverless*.

De forma a aferir este impacto foi desenvolvido um caso de estudo, inserido no contexto de uma loja *e-Commerce* de artigos de arte, que permitiu identificar as diferenças existentes entre o desenvolvimento *serverless* e os métodos de desenvolvimento atuais como é o caso de desenvolvimento orientado a microserviços.

Para além disso, os resultados obtidos na avaliação deste caso de estudo permitiram concluir que, quando aplicado em contextos específicos, o desenvolvimento *serverless* ainda que apresente um rendimento superior e melhores tempos de resposta quando exposto a cargas de utilização maiores quando comparado com arquiteturas atuais como é o caso de arquiteturas orientadas a microserviços esta diferença nos resultados ainda não é significativa, havendo espaço para evolução neste aspecto.

Palavras-chave: Processo de desenvolvimento, Nuvem, *Serverless*, E-Commerce

Abstract

The traditional software deployment processes, i.e., based on an on-premises model, with a more or less static and well-defined set of servers, have been increasingly problematic, whether due to the rapid obsolescence of the devices, the increased need for equipment, or the maintenance costs, or others.

Thus, in response to these problems, the deployment processes have evolved and led to new service models that aim to simplify this deployment. It is no longer necessary to prepare and allocate space for physical machines and network equipment, but to rent virtual machines and delegate the concern of managing the physical resources that these entail to another entity.

Despite this evolution, companies still continue to think and spend time on their systems in terms of servers, sometimes leaving to the background what is actually more relevant to them in operational terms: that their applications run the logic they are supposed to and that their data is secure and correct.

In this sense, since 2012, there have been studies about what it would mean to manage systems and not servers. This way of working has been called serverless, a name that is not due to the fact that there are no servers, but to the fact that we don't think about them so much.

On the other hand, this evolution in the deployment processes has also directly impacted the way the software itself is developed in order to support these new models and paradigms (e.g. serverless).

The main goal of this thesis is to assess this impact and to identify and design a set of recommendations that should be adopted in the development of applications and respective architectures in order to eliminate or minimize the difficulties that condition the adoption of serverless.

In order to assess this impact, a case study was developed, within the context of an art e-Commerce store, which allowed the identification of the existing differences between serverless development and current development methods, such as microservices oriented development.

In addition, the results obtained in the evaluation of this case study allowed to conclude that, when applied in specific contexts, even though serverless development presents superior performance and better response times when exposed to greater use loads when compared to current architectures such as microservices-oriented architectures, the difference in results is still not significant, and there is room for evolution in this aspect.

Agradecimentos

Primeiramente, quero agradecer aos meus pais e irmãos, que são um suporte e uma fonte de motivação que me faz querer alcançar os meus objetivos todos os dias.

Quero também agradecer aos meus restantes familiares e amigos chegados, em particular à Sofia, ao Cláudio e ao Rui, pelo apoio e motivação durante este percurso.

Finalmente, mas não menos importante, agradeço ao Instituto Superior de Engenharia, do Politécnico do Porto, e aos seus docentes pela oportunidade de frequentar um mestrado deste nível e pelos conhecimentos que me foram transmitidos. No que diz respeito aos docentes, destaco o Dr. Paulo Maio por ter orientado este trabalho, por toda a disponibilidade que demonstrou ao longo deste período e por todo o conhecimento transmitido.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Lista de Acrónimos

AHP	*Analytic Hierarchy Process*.
API	*Application Programming Interface*.
CaaS	*Container as a Service*.
CVV	*Card Verification Value*.
FaaS	*Function as a Service*.
FEI	*Front End of Innovation*.
GC	Grau de Consistência.
HTTP	*Hypertext Transfer Protocol*.
I/O	*Input/Output*.
IA	Índice Aleatório.
IaaS	*Infrastructure as a Service*.
IAC	*Infrastructure as Code*.
IC	Índice de Consistência.
IDE	*Integrated Development Environment*.
IDG	International Data Corporation.
IoT	*Internet of Things*.
ISEP	Instituto Superior de Engenharia do Porto.
LIFO	*Last In, First Out*.
NCD	*New Concept Development*.
NoSQL	*Not Only SQL*.
NPD	*New Product Development*.
PaaS	*Platform as a Service*.
REST	Representational State Transfer.
SaaS	*Software as a Service*.
SAM	*Serverless Application Model*.
SSL	*Secure Sockets Layer*.
TI	Tecnologia da Informação.
TMDEI	Tese de Mestrado do Departamento de Engenharia Informática.

Capítulo 1

Introdução

Este trabalho foi desenvolvido no âmbito da unidade curricular de Tese de Mestrado do Departamento de Engenharia Informática (TMDEI) e tem como propósito apresentar e analisar o estado atual e futuro das tecnologias *serverless*. Este capítulo tem como finalidade contextualizar o leitor com o problema a ser resolvido e descrever os objetivos a serem atingidos e a estrutura do documento.

1.1 Contexto

No processo seguido tradicionalmente no desenvolvimento de software as empresas são inteiramente responsáveis pelas operações relacionadas com a aquisição, manutenção de servidores e posterior implantação dos seus produtos e aplicações nesses mesmos servidores. Inicialmente eram utilizados servidores *on-premises*, isto é, equipamentos físicos que são colocados dentro da empresa, geralmente no seu *data center*. Com este tipo de servidores, existe a possibilidade de as equipas acederem fisicamente aos dados e de configurarem e gerirem a sua infraestrutura de computação [1].

No entanto, com o passar dos anos, começaram a surgir problemas na utilização deste processo tais como:

- aumento do armazenamento de dados não estruturados (dados do tipo *flat files* ou armazenados em bases de dados *Not Only SQL* (NoSQL))
- o aumento do número de acessos aos recursos informáticos, que tem como consequência o aumento da carga de *Input/Output* (I/O)
- a rápida obsolescência dos dispositivos
- a necessidade de uma equipa de Tecnologia da Informação (TI) especializada na operação e manutenção dos equipamentos
- o aumento da necessidade de equipamentos que extravasam os limites físicos do data center

Uma grande parte destes problemas levou à necessidade do aumento do número de servidores, o que em alguns casos se viria a tornar insustentável. Nas últimas duas décadas o aparecimento e crescimento da *cloud* veio revolucionar a forma como pensamos sobre a operação dos produtos e aplicações. Deixou de ser necessário preparar e alocar espaço para máquinas físicas e equipamentos de rede, passando-se a alugar máquinas virtuais e a delegar a preocupação de gerir os recursos físicos que estas acarretam a uma outra entidade

[2]. Desta forma, passamos de preocuparmo-nos com a aquisição, gestão e manutenção de servidores para nos preocuparmos apenas com o seu aluguer.

1.2 Problema

Ao nível da computação em cloud, existem vários tipos de serviço oferecido [3], tais como:

- *Infrastructure as a Service* (IaaS) - neste modelo a organização contrata uma capacidade de *hardware* (memória, processamento, armazenamento,etc) e o *software* associado, como a virtualização de sistemas operativos e sistemas de ficheiros. Deste contrato pode então resultar, por exemplo, a aquisição de um servidor, *router*, *racks*, entre outros. [4] Desta forma, este modelo oferece à organização um suporte na sua infraestrutura, sendo da responsabilidade da organização a configuração e gestão da plataforma e ambiente e de implantar aplicações no mesmo (e.g. AWS [5])
- *Platform as a Service* (PaaS) - consiste num conjunto de *software* e ferramentas de desenvolvimento hospedadas nos servidores do fornecedor do serviço. Neste modelo o serviço fornece o *hardware* (semelhante a um IaaS) em conjunto com uma aplicação que pretende suportar a resolução de um problema específico, como a integração com um conjunto de funções ou de base de dados que são expostas na forma de uma camada base sobre a qual se podem construir outras aplicações. [4] Este serviço facilita assim o desenvolvimento e implantação de aplicações, com a redução do custo e complexidade da compra e gestão da infraestrutura subjacente, fornecendo todas as ferramentas necessárias para dar suporte ao ciclo de vida completo do desenvolvimento de aplicações e serviços (e.g. App Engine [6], Heroku [7])
- *Container as a Service* (CaaS) - um *container* é uma unidade de *software* executável no qual o código das aplicações é empacotado em conjunto com as suas bibliotecas e dependências de forma a que possa ser executado a partir de qualquer lugar (*desktop*, nuvem, entre outros). Este modelo permite que os utilizadores organizem e giram *containers*, aplicações e clusters através de uma virtualização baseada em *container*, *Application Programming Interface* (API) ou interface web (e.g. Kubernetes [8])
- *Function as a Service* (FaaS) - é um tipo de serviço que permite a execução de código em resposta a eventos, sem a complexidade da infraestrutura normalmente associada à construção e lançamento de aplicações baseadas em microsserviços. Neste modelo o hardware físico e o sistema operativo da máquina virtual são geridos automaticamente pelo fornecedor, permitindo ao utilizador um maior foco na lógica de negócio (e.g. AWS Lambda [9])
- *Software as a Service* (SaaS) - distribui softwares completos, como servidores, código de aplicações e bases de dados sob a forma de serviço, sendo estes acessíveis através da internet e não necessitando, portanto, da instalação de nenhum programa nos equipamentos (e.g. Microsoft Office 365 [10])

Na figura 1.1 é apresentada a distinção entre cada serviço no que diz respeito às tarefas pelas quais cada serviço se responsabiliza.

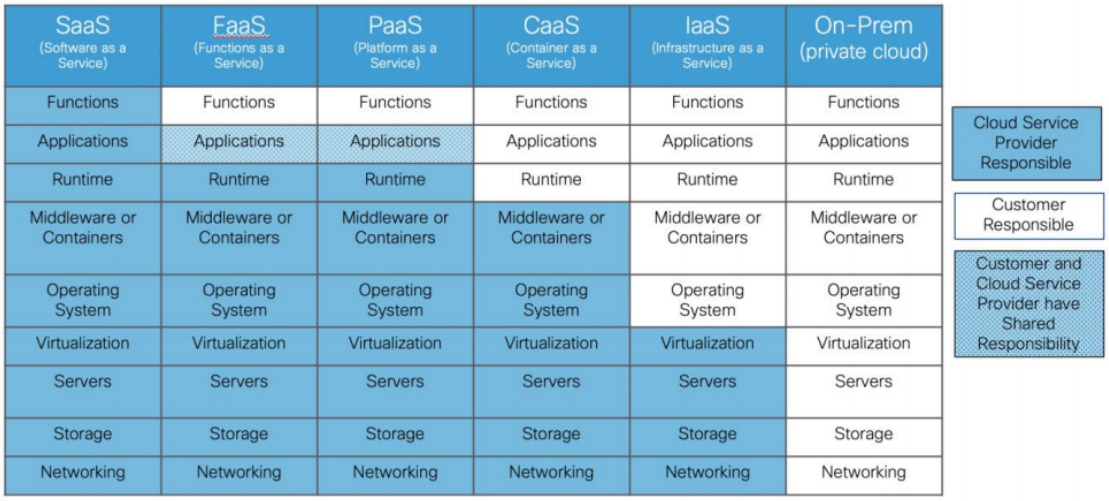

Figura 1.1: Tarefas pelas quais cada serviço *cloud* se responsabiliza). Fonte: Cisco [11]

Apesar desta diversidade e evolução, as empresas continuam ainda a pensar e a despender tempo útil nos seus sistemas em termos de servidores – componentes que alocamos, configuramos, implantamos, inicializamos e gerimos. Contudo, o que é efetivamente mais relevante para as empresas em termos operacionais é que as suas aplicações executem a lógica que é suposto e que os seus dados estejam seguros e corretos.

Neste sentido, desde 2012, têm surgido estudos sobre o que significaria gerir sistemas e não servidores [2]. Isto é, passar-se a pensar em aplicações como fluxos de trabalho, lógica distribuída e armazenamento de dados geridos externamente. A esta forma de trabalhar deu-se o nome de serverless [12] - este nome não se deve ao facto de não existirem servidores mas sim ao facto de não termos de pensar muito neles.

Por outro lado, esta evolução nos processos de implantação tem impactado diretamente também a forma como o próprio software é desenvolvido de modo a suportar estes novos modelos e paradigmas (e.g *serverless*).

Torna-se então importante aferir este impacto, perceber como é que a *cloud* pode ajudar as empresas a dar e acelerar este salto de paradigma e o que é que isso implica no processo de desenvolvimento das suas aplicações e respetivas arquiteturas.

1.3 Objetivos

De forma a compreender as dimensões deste problema será identificado e desenhado um conjunto de recomendações (e.g. ajustes) que devem ser adotadas no desenvolvimento de aplicações e respetivas arquiteturas com vista a eliminar ou minimizar as dificuldades/razões previamente identificadas que condicionam a adoção de *serverless*. Para além disso, também será desenhada uma solução *serverless*, como caso de estudo de forma a aferir a utilidade destas recomendações.

De forma a cumprir este objetivo, serão realizadas as seguintes tarefas:

- T1 - Identificar e sistematizar em que consiste o paradigma *serverless* e práticas comuns de adoção

- T2 - Identificar e sistematizar as principais dificuldades/razões que atualmente condicionam a adoção de *serverless* e a sua relevância dependendo do tipo de projeto/aplicação
- T3 - Identificar, descrever e sintetizar características/qualidades que as aplicações/-sistemas devem reunir de modo a que desenvolver as mesmas em/para *serverless* seja efetivamente útil
- T4 - Identificar, descrever e sintetizar as abordagens, ferramentas e tecnologias que suportam o desenvolvimento *serverless*, fazendo uma distinção entre elas, por exemplo, ao nível do grau de (in)dependência da plataforma cloud utilizada
- T5 - Avaliar com base na literatura e alguma experimentação a pertinência e adequação das abordagens, ferramentas e tecnologias que suportam o desenvolvimento *serverless*
- T6 - Avaliar os efeitos/limitações/consequências de adoção de tecnologias dependentes da plataforma cloud quando comparadas com tecnologias independentes
- T7 - Identificar e desenhar, com base no método de Design Science, um conjunto de recomendações (e.g. ajustes) que devem ser adotadas no desenvolvimento de aplicações e respetivas arquiteturas com vista a eliminar e/ou minimizar as dificuldades/razões previamente identificadas que condicionam a adoção de *serverless*
- T8 - Desenhar uma solução *serverless* para as aplicações usadas como casos de estudo
- T9 - Construir a solução *serverless* anteriormente desenhada para as aplicações usadas como casos de estudo
- T10 - Recolher informação, da solução desenvolvida,de alguns requisitos de qualidade como a performance, desempenho, disponibilidade e escalabilidade que permita fazer uma comparação com os valores normalmente obtidos em aplicações "no-serverless" semelhantes

1.4 Estrutura do Documento

Este documento está dividido em 8 capítulos:

1. Introdução - uma breve apresentação ao tema, problema e objetivos da dissertação.
2. Contexto e estado de arte - neste capítulo são dados alguns detalhes sobre o contexto e o problema abordado nesta dissertação e é feito um estado de arte das plataformas, tecnologias e soluções *serverless* existentes. Desta forma, nesta secção serão analisados os problemas e respondidas as questões que suportam as primeiras seis tarefas identificadas na secção anterior.
3. Análise de valor - nesta secção é avaliado o produto a desenvolver através do modelo *New Concept Development*, definidos o valor, valor percecionado e valor para o cliente oferecidos pela solução e aplicado o modelo de Verna Alee, de forma a perceber melhor o modelo de negócio da solução, e o método AHP com o objetivo de decidir qual a ideia a desenvolver.
4. Especificação do Caso de Estudo - neste capítulo são levantados os requisitos da solução através do processo de engenharia de requisitos, apresentado o modelo de domínio do caso de estudo e os casos de uso a implementar.

5. Design - neste capítulo é apresentado o design da arquitetura e dos casos de uso a implementar tendo em conta duas abordagens, uma abordagem tradicional de micro-serviços e uma abordagem *serverless*.
6. Implementação - neste capítulo é apresentada a stack tecnológica da solução desenvolvida e descrita a implementação de serviços *serverless* e dos casos de uso da solução.
7. Avaliação - nesta fase são descritas as hipóteses a testar relativamente ao objetivo da dissertação bem como apresentada a metodologia seguida nesta avaliação, os resultados obtidos e feita uma análise dos resultados obtidos e, por este motivo, esta secção está diretamente relacionada com a antepenúltima tarefa identificada.
8. Conclusão - neste último capítulo são apresentadas as conclusões do trabalho. desenvolvido, os objetivos concluídos e feita uma reflexão sobre limitações e trabalho futuro desta dissertação.

Capítulo 2

Contexto e Estado da Arte

Neste capítulo é analisado com detalhe o contexto e o problema em análise neste documento, bem como descrita a tecnologia *serverless* e em que é que esta consiste, feita uma análise das preocupações e benefícios que esta acarreta/oferece e finalmente feito um estado de arte das ferramentas existentes no mercado e que estão diretamente relacionadas com este paradigma.

2.1 Detalhes do contexto e problema

Nesta secção utiliza-se algumas estatísticas de forma a fazer uma análise detalhada do contexto e problema e é descrita a tecnologia *serverless* e todas as suas envolventes.

2.1.1 Contexto

Nos últimos anos, tem-se verificado um crescimento enorme da computação e do uso de tecnologia no geral. Dados do Eurostat, apresentados na figura 2.1, mostram que Portugal tem acompanhado a União Europeia naquilo que é o aumento de percentagem de pessoas com acesso à Internet. Se em 2011 apenas 57% da população portuguesa tinha acesso a esta rede, no final de 2020 atingiu-se a marca de 82%, o que demonstra claramente este aumento.

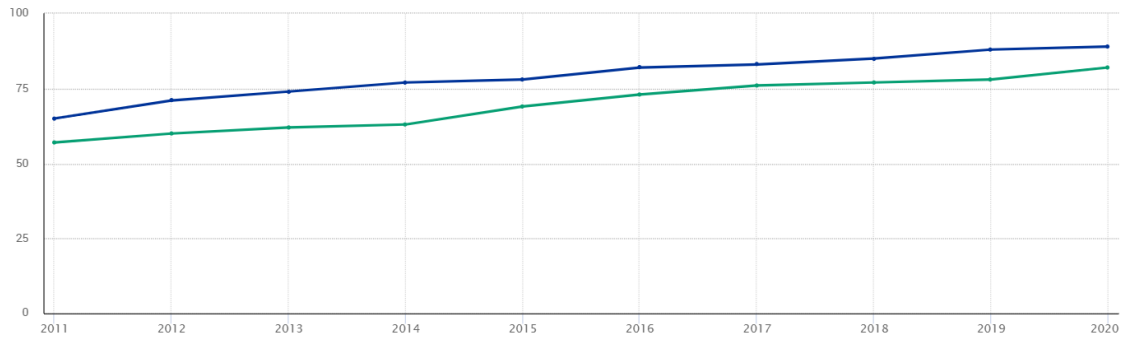

Figura 2.1: Acesso à internet por parte da população portuguesa (a verde) e europeia (a azul). Fonte: Eurostat [13]

O crescimento da percentagem de pessoas com acesso à Internet trouxe uma consequência quase que inevitável: existindo mais pessoas com acesso à Internet, tendem a existir mais pessoas a aceder aos diferentes serviços e plataformas desta rede, o que por sua vez tende a que exista um aumento das interações destas pessoas com estas plataformas, que

resulta num aumento da carga de I/O que, em último caso, leva à necessidade das empresas utilizarem mais servidores para as suas aplicações, de forma a distribuir esta carga de interações. Este aumento do número de servidores implicou uma maior despesa, o que para algumas empresas se tornaria insustentável. Entre 2006 e 2008, começaram a surgir as primeiras soluções de computação em nuvem, nome genérico dado à computação em servidores disponíveis na Internet a partir de diferentes fornecedores.

Para além da vantagem óbvia de não ser necessário alocar espaço físico para máquinas e equipamentos de rede, este tipo de computação trouxe outros benefícios bastante importantes:

- Manutibilidade - é facilitada pois as aplicações não precisam de ser instaladas em cada máquina e são acessíveis a partir de qualquer lugar
- Custo - um estudo feito pela Sherweb [14] revelou que num período de 7 anos, e assumindo que os servidores *on-premises* requerem uma atualização/mudança de 5 em 5 anos, uma empresa pouparia 79% ao adotar uma arquitetura em nuvem. Esta poupança deve-se, por exemplo, ao facto de não ser necessário despender tempo na instalação e configuração dos servidores e ao desaparecimento de custos extra em sistemas de ar condicionado
- Produtividade - os utilizadores deixam de precisar de instalar atualizações do software no seu computador

Estas vantagens têm-se tornado cada vez mais aliciantes e prova disso é o aumento nos últimos anos em cerca de 12% da percentagem de população portuguesa que usufrui deste tipo de serviços, acompanhando assim o crescimento verificado na União Europeia. Este aumento está demonstrado na figura 2.2.

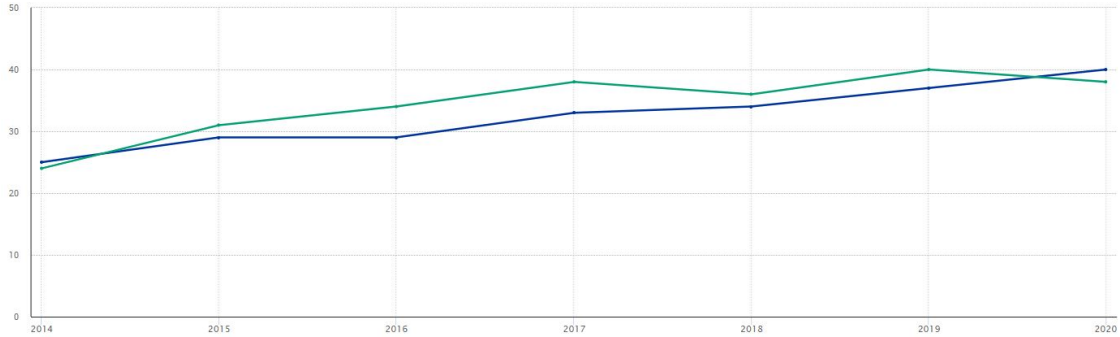

Figura 2.2: Uso de serviços *cloud* por parte de indivíduos portugueses (a verde) e da União Europeia (a azul). Fonte: Eurostat [15]

Crescimento esse que também é verificado na percentagem de empresas, quer em Portugal quer na globalidade da União Europeia, que adquiriu serviços *cloud* nos últimos anos, como se pode verificar na figura 2.3.

Este crescimento levanta então uma questão: a *cloud* solucionou já todos os problemas relativos à implantação de aplicações?

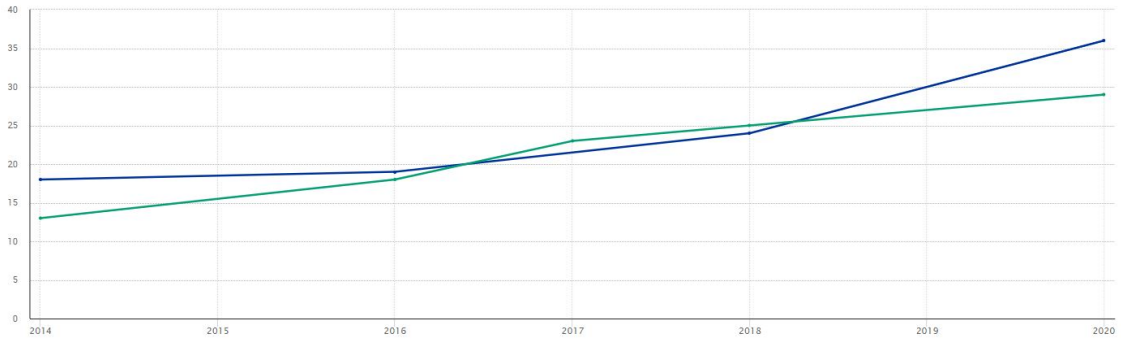

Figura 2.3: Percentagem de empresas que compraram serviços *cloud* em Portugal (a verde) e na União Europeia (a azul). Fonte: Eurostat [16]

2.1.2 Problema

Nesta secção responder-se-á então à questão levantada. Com este crescimento da computação em nuvem foram surgindo várias tentativas de exploração da mesma, estando as principais numeradas na secção 1.2. Estes tipos de serviço têm tido bastante sucesso, havendo uma grande aderência por parte das empresas aos mesmos, como se pode verificar na figura 2.4. Este sucesso e crescimento foram reforçados, indiretamente, com a chegada da pandemia COVID-19. O facto de as empresas precisarem que os seus trabalhadores trabalhem à distância e que, ainda assim, consigam gerir os seus serviços e servidores fez com que muitas investissem na computação em nuvem, como se pode verificar na figura 2.5. Nesta figura é feita uma comparação desta alteração à adesão a serviços *cloud* entre pequenas e médias empresas com menos de 1000 colaboradores (SMB) e organizações com mais de 1000 funcionários (*enterprise*).

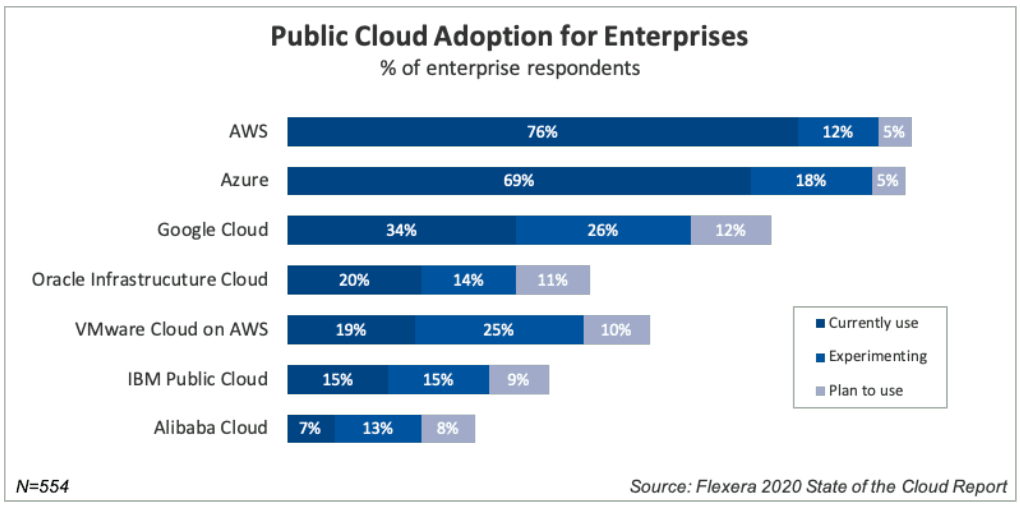

Figura 2.4: Percentagem de empresas que adotaram serviços *cloud*. Fonte: Flexera [17]

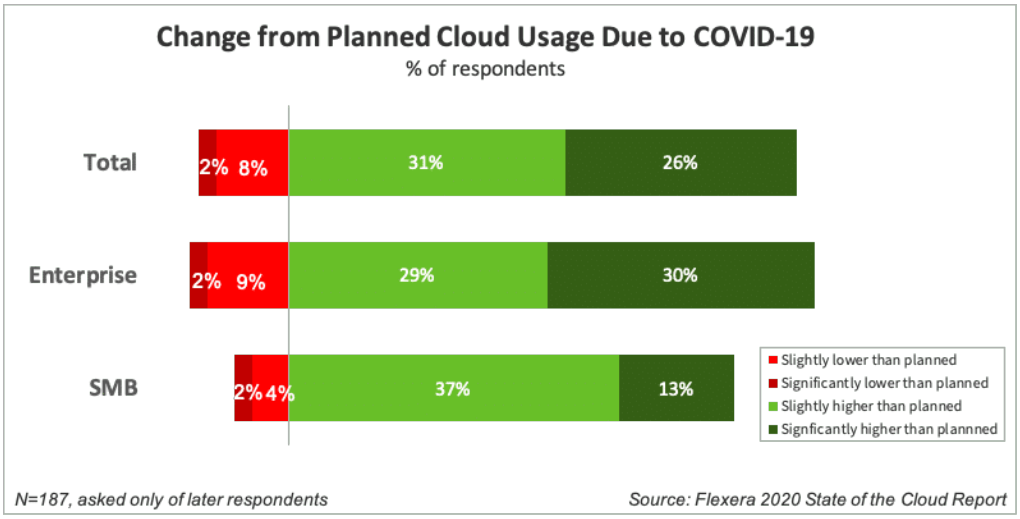

Figura 2.5: Alterações ao uso de computação em nuvem devido à pandemia COVID-19 Fonte: Flexera [17]

Ainda que este crescimento seja notório, há um problema que grande parte destes serviços não conseguiu ainda resolver: a dependência de operacionais (e tempo útil) na gestão destes sistemas. Dados da International Data Corporation (IDG), apresentados na figura 2.6, mostram que é necessário um investimento forte em novos cargos e papéis nas empresas após a adoção deste tipo de tecnologias nas mesmas. [18] Este investimento em conjunto com o tempo gasto nas tarefas menos relevantes, anteriormente mencionadas, acaba por ter um peso financeiro muito grande. Isto é, não é do interesse das empresas despenderem tempo na configuração, implantação e inicialização dos servidores na nuvem, pois estes não trazem valor "visível"para os seus produtos.

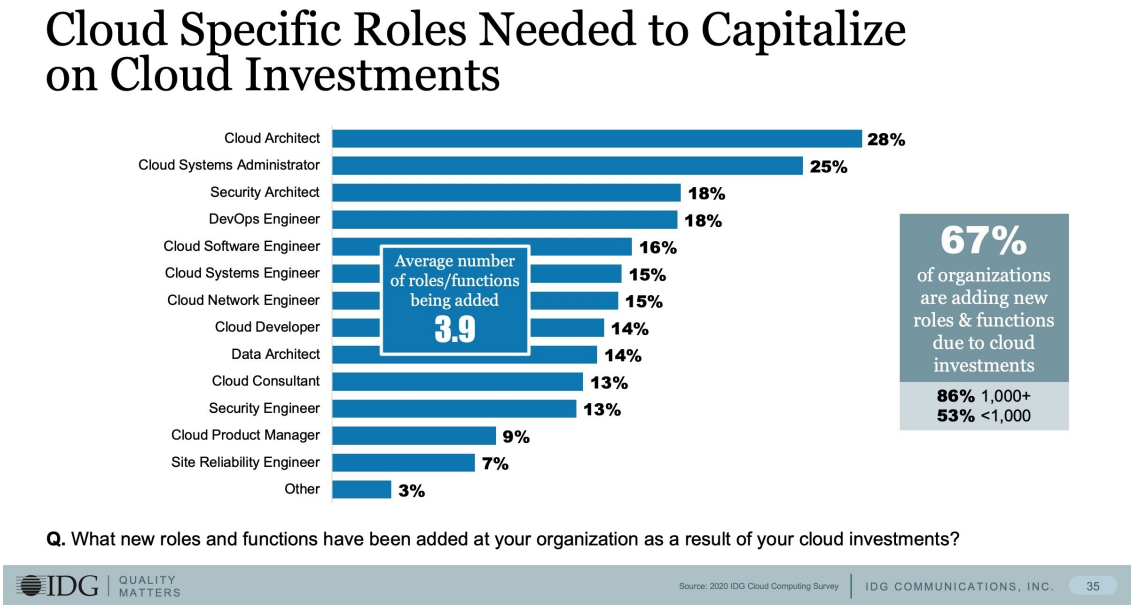

Figura 2.6: Funções e cargos adicionados às empresas após a introdução da computação em nuvem nas mesmas. Fonte: IDG [18]

Desta conclusão resulta então uma questão: há já alguma forma de combater esta complexidade e custo de implantação de soluções na *cloud*?

2.1.3 Serverless

Nos últimos anos surgiu um conceito que tem como principal objetivo fazer desaparecer esta complexidade ao nível da gestão de servidores na nuvem, conhecido como computação *serverless*. Esta é uma metodologia orientada a eventos, isto é, promove a produção, detecção, consumo e reação a eventos. Existem então alguns conceitos relacionados com este paradigma que é importante que sejam compreendidos:

- **evento** - um evento pode ser definido como uma mudança no estado da aplicação [19]. Tomando como exemplo uma aplicação de venda de automóveis, um evento possível é a venda de um automóvel, onde o automóvel passa do estado "para venda"para o estado "vendido"
- **trigger** - o *trigger* é o responsável pela produção dos eventos e tipicamente aparece na forma de pedidos *Hypertext Transfer Protocol* (HTTP) realizados através de uma API fornecida pelo fornecedor do serviço
- **função** - é responsável pelo consumo dos eventos, sendo por isso o local onde reside a lógica de negócio das aplicações

Depois de compreendidos estes conceitos, é relativamente simples compreender o processo seguido tradicionalmente neste tipo de computação: um programador escreve uma função na linguagem que pretende, de forma a responder a um determinado evento. De seguida é feito o *upload* desta função para o serviço na nuvem e é retornada por este serviço uma API, que permite invocar a função remotamente, funcionando os pedidos feitos à API como *triggers* de eventos [19].

Sempre que esta função recebe algum tipo de dados, é computado e retornado o resultado da aplicação deste input na função [19]. Desta forma o programador não despende tempo na configuração dos servidores, apenas em tarefas vitais para o desenvolvimento do produto. Para além disso, os recursos da nuvem oferecem uma maior flexibilidade, aumentando ou diminuindo a sua capacidade conforme o uso e permitindo às organizações um pagamento ajustado ao uso dos seus serviços[19]. Este é o principal avanço que a metodologia *serverless* propõe para o uso de serviços na nuvem.

Utilidade do uso de tecnologia serverless

Apesar deste avanço proposto pelo pensamento *serverless*, é importante notar que, como qualquer tecnologia, esta é mais eficaz quando aplicada em determinados contextos. De forma a perceber quais são estes contextos torna-se relevante analisar algumas estatísticas relativas à utilização de tecnologias *serverless* nos dias de hoje e aos fatores que condicionam as empresas na adoção deste tipo de tecnologias e sintetizar as características que os sistemas devem reunir de modo a que desenvolver os mesmos em *serverless* seja efetivamente útil.

Num estudo feito pela organização O'Reilly em 2019 [20], foram recolhidas algumas informações relativas à adoção de tecnologias *serverless* por parte das empresas. Quando questionadas sobre se as suas organizações já tinham adotado tecnologias *serverless*, com adotado a significar o estabelecimento de um contrato com um serviço de nuvem de forma

a fornecer recursos *serverless*, 40% das empresas sondadas responderam afirmativamente, uma percentagem bastante elevada para aquilo que é uma tecnologia tão recente. Destas organizações que já possuem tecnologias *serverless* nas suas arquiteturas mais de metade afirmou que esta adoção foi feita entre 2015 e 2018 e 15% iniciou esta utilização antes de 2018.

No que diz respeito a benefícios da adoção deste tipo de tecnologias, apresentados na figura 2.7, a redução de custos, a escalabilidade automática e a redução da preocupação com a manutenção dos servidores foram os benefícios mais mencionados.

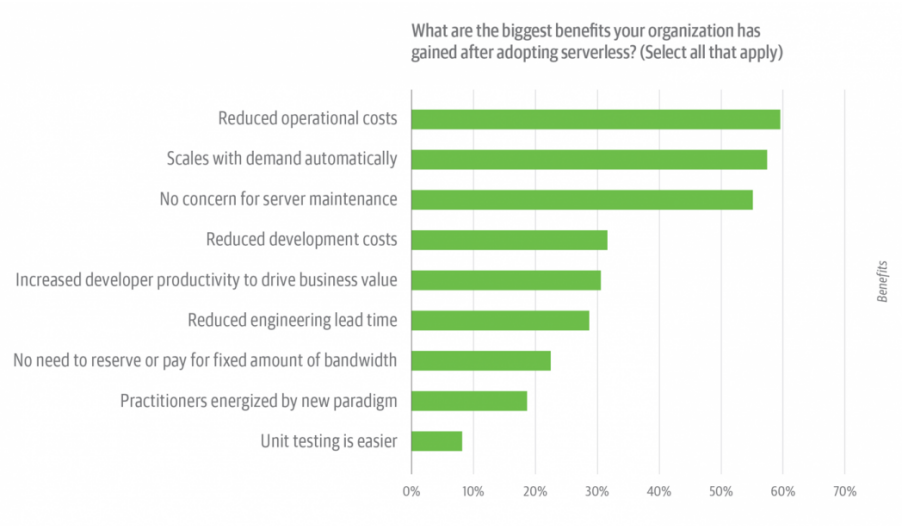

Figura 2.7: Benefícios da adoção de tecnologias *serverless* Fonte: [20]

Já em termos de preocupações, que esta adoção implicou, apresentadas na figura 2.8, imperou a necessidade de educar os colaboradores. Sendo o pensamento *serverless* relativamente recente, é mais complicado encontrar pessoas especializadas capazes de formar os funcionários, sendo também relativamente pequena a quantidade de documentação e casos de estudo existentes. Este factor a juntar ao aparecimento constante de novas tecnologias neste mercado, torna a educação de colaboradores relativamente a estas tecnologias um processo árduo. Para além desta preocupação foram enumeradas também a dependência de um serviço externo e a dificuldade em testar e depurar as aplicações.

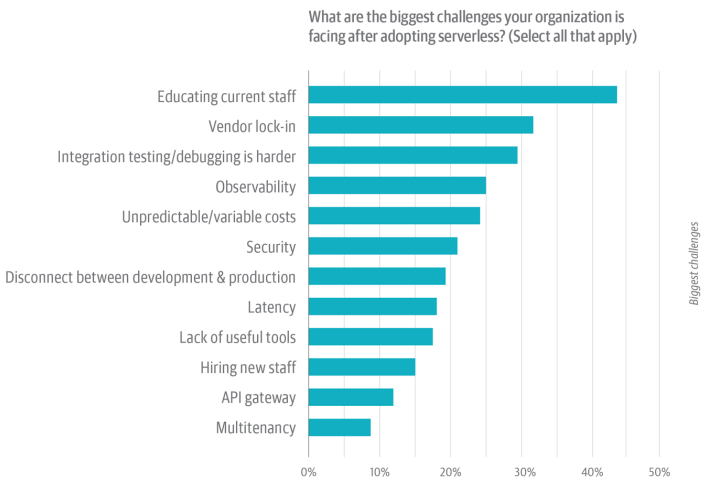

Figura 2.8: Preocupações que a adoção de tecnologias *serverless* implicou para as empresas Fonte: [20]

Em contrapartida, quando questionadas as empresas que ainda não aderiram às tecnologias *serverless* sobre o porquê desta não adesão, a maior parte indicou as preocupações com a segurança das suas aplicações, o medo do desconhecido e a necessidade de completar a mudança para o paradigma de arquitetura em *cloud* como as principais razões. A tecnologia *serverless* introduz uma mudança no que diz respeito à gestão dos dados das aplicações, onde os dados sensíveis são bastantes dinâmicos, o que pode colocar algumas questões sobre quem tem acesso a estes e sobre o quão seguros estão [20]. Na figura 2.9 encontra-se apresentada a lista completa de razões enumeradas pelas empresas para a não adesão a arquiteturas *serverless*.

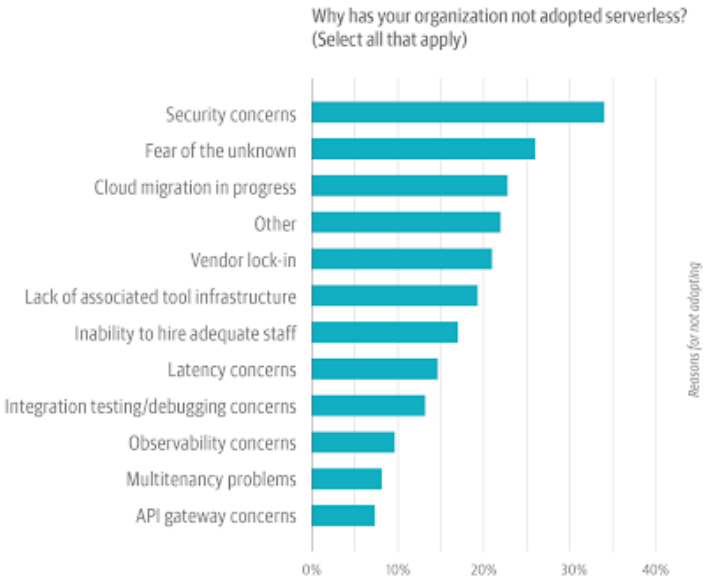

Figura 2.9: Razões pelas quais as empresas ainda não adotaram tecnologias *serverless* Fonte: [20]

As dinâmicas de mercado mudam continuamente devido à proliferação de tecnologias como a inteligência artificial, *machine learning*, *Internet of Things* (IoT), entre outros [21]. Com estas mudanças tão sistemáticas, as empresas estão sobre pressão para lançar novos produtos e ferramentas num curto espaço de tempo, de forma a atender às expectativas do consumidor. A tecnologia *serverless* consegue potencializar e melhorar o processo de implantação de aplicações pelo que não é de estranhar que se preveja que o tamanho de mercado deste tipo de tecnologia aumente. Esta tendência foi também prevista num relatório realizado em 2018 [21], que antevê que em 2025 a automação e integração, gestão de APIs e monitorização possuam a maior percentagem do mercado. Na figura 2.10 é possível visualizar o gráfico relativo a esta previsão.

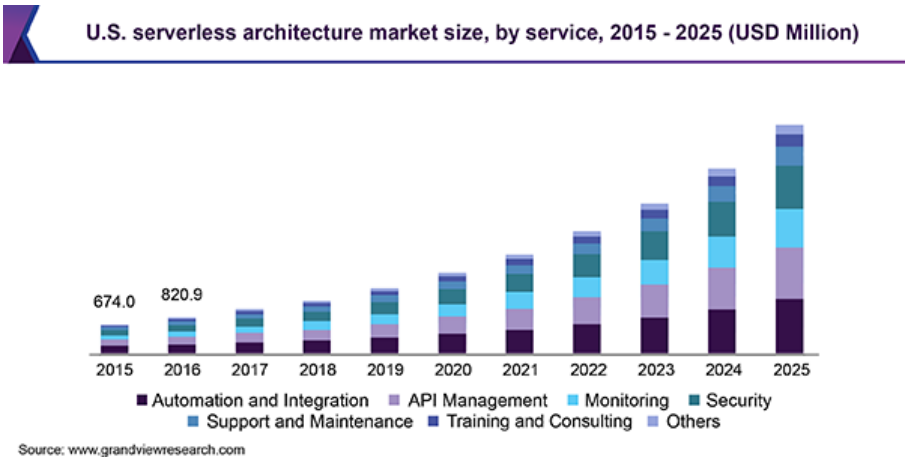

Figura 2.10: Tamanho de cada mercado no que à presença de arquitetura *serverless* diz respeito Fonte: [21]

O segmento da banca, serviços finaceiros e seguros (BFSI) tem dominado o mercado de arquiteturas *serverless*, como é possível comprovar na figura 2.11, e é esperado que assim continue [21]. Muitas empresas deste segmento possuem uma base de utilizadores bastante significativa pelo que a adoção de arquiteturas *serverless* permitiu, a estas organizações, principalmente uma escalabilidade automática consoante a demanda, que é uma das maiores preocupações de uma aplicação deste tipo. Isto permitiu optimizar o fluxo de trabalho dos bancos, fornecer respostas rápidas aos clientes e consequentemente aumentar a satisfação dos mesmos.

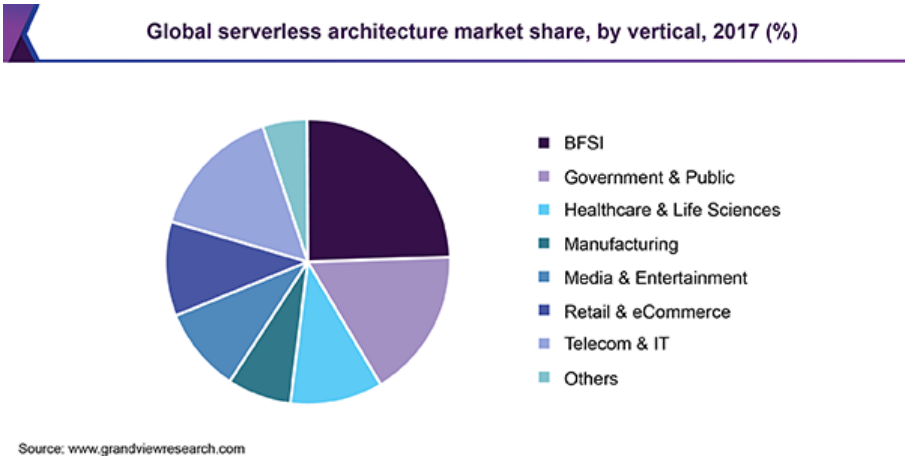

Figura 2.11: Segmentação do mercado com presença de arquiteturas *serverless* Fonte: [21]

Estas estatísticas permitem recolher, tal como sugerido pela recomendação R1, algumas características que as arquiteturas atuais não possuem e que se espera que a arquitetura *serverless* resolva:

- grande variedade na quantidade de acessos às aplicações para a qual uma escalabilidade automática é essencial
- necessidade de desenvolvimento de atualizações sistemáticas em que a possibilidade de se focar apenas na lógica de negócio é essencial
- complexidade significativa da lógica de negócio, que envolva a implantação de vários serviços e servidores, implantação essa que é então facilitada pela abordagem *serverless*

Nestas características encaixam-se mercados como: sistemas bancários, governamentais, da área da saúde, jornais e restantes media e, por fim, retalho e *e-Commerce*.

2.2 Estado da Arte

Da secção anterior resultam algumas questões que importa responder como: Há motivos para as preocupações expressas pelas empresas? Se sim, que suporte está a ser dado hoje em dia de forma a minimizar e resolver estes problemas? Nesta secção serão analisadas algumas tecnologias que visam minimizar estas dificuldades levantadas pelas organizações.

2.2.1 Arquitetura

No que diz respeito à arquitetura, existem já varias soluções que têm como principal objetivo simplificar o processo de criação e montagem de uma arquitetura *serverless*, fornecendo inclusive a lógica FaaS. Nesta secção serão analisadas as três soluções mais conceituadas a este nível: AWS Lambda [9], Azure Functions [22] e Google Functions [23].

AWS Lambda

A Amazon foi o primeiro grande fornecedor de serviços da nuvem a lançar recursos *serverless*, na forma das funções Lambda. O processo, apresentado na figura 2.12, é relativamente

simples. Inicialmente faz-se upload para a plataforma do código da função numa das linguagens suportadas (Node.js, Python, Go, Java, etc). [24] De seguida, define-se o *trigger* que irá despoletar a execução desta função.

Este *trigger* pode ser feito através de pedidos HTTP à API fornecida pela plataforma, de ações realizadas diretamente na aplicação da plataforma ou de um dos 140 serviços da Amazon [24], onde se inclui, a título de exemplo, alterações dos dados armazenados numa base de dados DynamoDB [25] ou transmissões na plataforma Kinesis [26].

Finalmente o Lambda define de forma precisa os recursos computacionais necessários para a execução da função e executa-a apenas quando acionado o seu *trigger*, pagando-se apenas pelo tempo de computorização utilizado. [9]

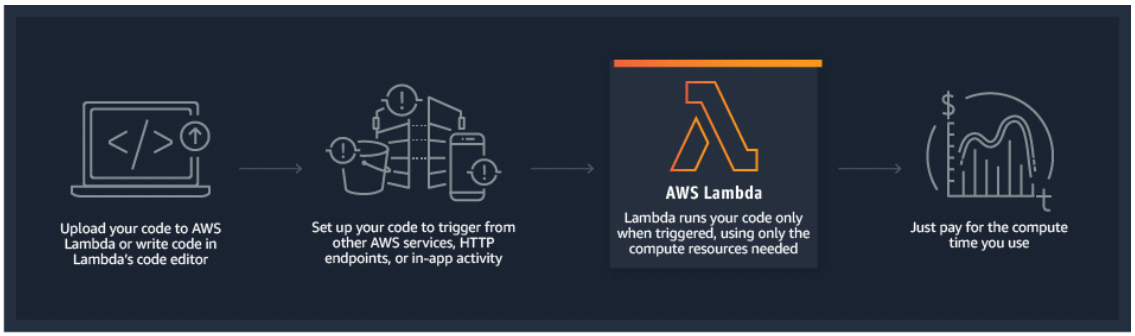

Figura 2.12: Processo de utilização do AWS Lambda [9]

Um exemplo prático da aplicação deste serviço é o do jornal americano Seattle Times que usa o Lambda para o redimensionamento de fotografias de forma a serem apresentadas em diferentes tipos de dispositivos. Este processo, que se encontra apresentado de forma resumida na figura 2.13, consiste no *upload* de uma fotografia para um dos serviços da Amazon que despoleta automaticamente o Lambda que retorna como *output* a fotografia redimensionada.

Figura 2.13: Processo de utilização do AWS Lambda por parte do Seattle Times [9]

Azure Functions

Ainda que tenha lançado a sua plataforma um pouco mais tarde, o Azure tem depositado grandes esforços no desenvolvimento da mesma. Visto que se trata também ela de um serviço FaaS funciona, à semelhança do que acontece no caso da AWS, à base de funções. Analisando uma utilização comum deste tipo de funções, que é apresentado na figura

2.14, é possível perceber que este processo, que segue a lógica da metodologia *serverless* apresentada anteriormente, se divide em 3 etapas principais [24]:

- *Trigger* - despoleta a execução de uma função, e pode variar entre pedidos HTTP à API do Azure,criação de eventos através do Azure Event Hubs, notificações *blob* enviadas a partir do Azure Storage ou pela calendarização deste mesmo *trigger*
- Reação - como reação ao *trigger* é executada uma função que pode ser codificada em diferentes linguagens (Node.js, Python, C#, etc)
- Resposta - o output da execução desta função é armazenado na base de dados Cosmos, armazenamento esse que origina a execução de uma segunda função que é responsável por enviar a resposta para o cliente

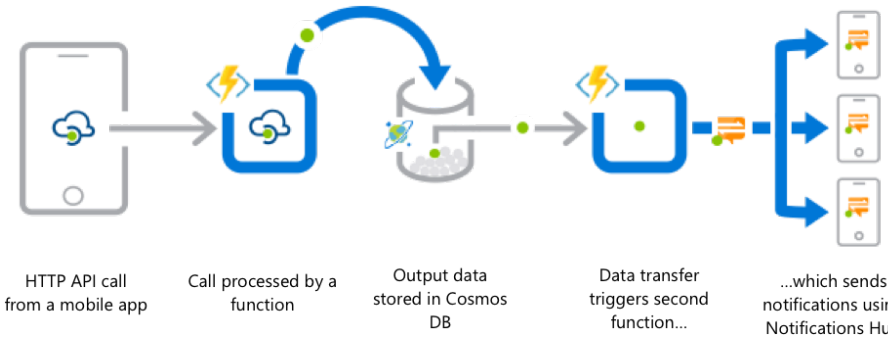

Figura 2.14: Processo de utilização convencional de uma função Azure [27]

Apesar dos esforços depositados no desenvolvimento desta solução, ainda há aspetos a serem melhorados por parte desta plataforma como o suporte operacional, pois as métricas de monitorização estão disponíveis apenas quando se utiliza o plano App Service, que requere que seja executada uma função numa máquina virtual gerida pelo próprio cliente [24], fazendo desaparecer assim um pouco daquilo que é a atração principal da computação *serverless*.

Google Functions

Das três organizações, a Google foi a que implementou mais tarde a sua componente *serverless* e, por esse mesmo motivo, ainda está numa fase mais embrionária em comparação com as restantes. Como é possível verificar na figura 2.15, continua a existir a noção de *trigger*, que neste caso é ativado por eventos através do Google Cloud Pub/Sub [28] mas também pode funcionar, por exemplo, na forma de pedidos HTTP ou notificações da Google Cloud Storage [29] [24]. Este *trigger* espoleta então a execução de funções que podem ou não armazenar os seus dados numa entidade externa, por exemplo a Google Cloud Storage. Finalmente é retornado o resultado para o cliente através da execução de uma função de saída.

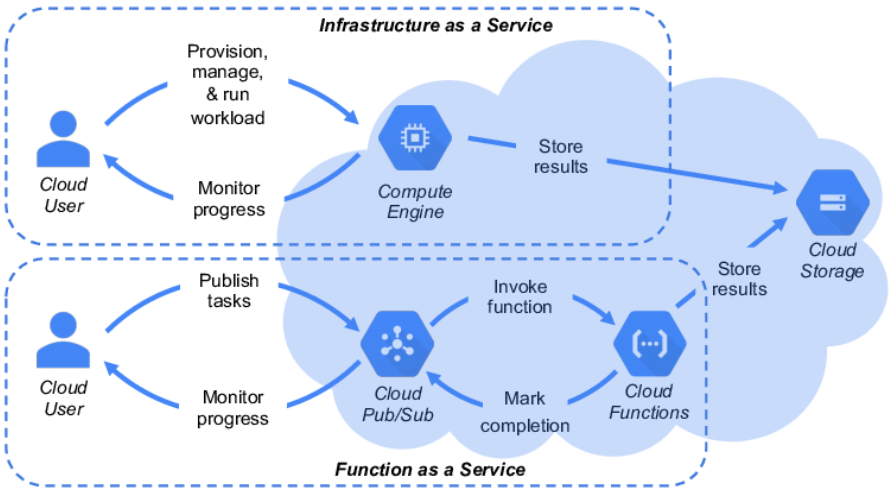

Figura 2.15: Processo de utilização convencional de uma função Google [30]

Para além disso, a Google oferece uma automatização do processo de monitorização, através do Pacote de operações do Google Cloud [31]. Neste processo as *logs* são automaticamente armazenadas nesta plataforma [24] e é fornecido um depurador que permite ao cliente investigar o funcionamento do seu código.

Comparação

De forma a fazer uma breve comparação entre as três plataformas, analisou-se algumas das principais características de cada uma, com base nos seus planos mais básicos, sendo estas características apresentadas na tabela 2.1.

Tabela 2.1: Comparação entre plataformas *serverless* [32]

	AWS Lambda	Azure Functions	Google Functions
Linguagens suportadas	Java,Go,PowerShell, Node.js,C#,Python e Ruby	Node.js,C#,F#,Java, PowerShell, Python e Typescript	Node.js,Python,Go, Java,.NET e Ruby
Persistência	Variáveis de ambiente	n/a	Variáveis de ambiente
Sistema Operativo	Linux x64	Windows x32	Linux x64
Tipo de software	*closed source*	*open source*	*closed source*
Tempo máximo de execução por função	5 minutos	10 minutos	9 minutos
Número máximo de funções por projeto	Ilimitado	100	Ilimitado
Número máximo de funções concorrentes	1000	Ilimitado	1000 (quando não ativadas via pedido HTTP)
Recursos reservados	Memória reservada por função	Memória reservada por serviço	n/a

Desta tabela é possível perceber que não há grandes diferenças entre as três plataformas no que diz respeito às linguagens suportadas. Naquilo que é relativo às características globais do software, o Azure foi o único a optar por Windows como Sistema Operativo, não suportando

a persistência de configurações e mantendo a sua plataforma *open source* enquanto que a Amazon e a Google apresentam características semelhantes entre si neste pontos.

2.2.2 Outras tecnologias e metodologias

Ainda que o pensamento *serverless* seja relativamente recente, com o passar dos anos foram sendo levantadas algumas problemáticas por parte das empresas, como visto na 2.1.3:

- o processo de implantação é complexo para pessoas sem conhecimento na área
- o processo de teste de uma aplicação implantada na *cloud* é complicado e custoso
- os dados de uma aplicação implantada na *cloud* não estão seguros

Importa então perceber até que ponto estas preocupações são válidas e que ferramentas existem de forma a combater estes problemas. Desta forma, com o objetivo de simplificar o processo de implantação foram desenvolvidas as seguintes ferramentas:

Serverless Framework

A Serverless Framework é a ferramenta mais popular no que diz respeito à computação *serverless* [33]. Permite o desenvolvimento de aplicações compostas por microserviços que são executados em resposta a eventos, com escalonamento automático e pagando-se apenas pelo período de computação utilizado, seguindo assim os princípios *serverless* mais comuns.

Fornece também a integração automática com as principais plataformas *serverless* (AWS Lambda, Google Functions e Azure Functions), não estando por isso dependente de uma plataforma em específico. Para além disso, suporta uma grande variedade de linguagens tais como Node.js, Python, Java, Go, C#, Ruby, Swift, Kotlin, PHP, Scala e F# e gere todo o ciclo de vida de uma arquitetura *serverless*.

Em 2020 iniciou também o desenvolvimento daquilo a que chamam "componentes", que consistem na configuração simplificada da implantação de aplicações de domínios específicos, onde se incluem:

- FullStack - uma aplicação completa construída com AWS Lambda, API HTTP do AWS, Express.js, React e DynamoDB
- Express.js - facilita a implantação de aplicações Express.js em plataformas *serverless*, fornecendo também suporte ao nível de domínios personalizados, certificados *Secure Sockets Layer* (SSL), entre outros
- WebSite - implantação automática de um *website* estático numa infraestrutura *serverless*, sem necessidade de configurações adicionais

Com estes componentes, a dificuldade na configuração de uma arquitetura *serverless* é reduzida e simplificada, sendo por isso uma grande mais valia e que faz esta *framework* destacar-se das demais.

Architect

A Architect [34] autodefine-se como uma *framework Infrastructure as Code* (IAC), que consiste no processo de gerir e providenciar *data centers* através de ficheiros ao invés da configuração física de equipamentos ou ferramentas de configuração interactivas [35]. Desta forma, esta ferramenta permite definir um arquivo *manifest* de alto nível, em formatos

variados de texto aberto, e transformar toda a complexidade de uma infraestrutura cloud neste único ficheiro [34]. Depois de definido este ficheiro é feita a sua passagem para o sistema de controlo de versões. Por fim, esta ferramenta compila este arquivo na plataforma AWS CloudFormation [36] e implanta a arquitetura definida no mesmo. Esta é por isso uma *framework* dependente da plataforma, neste caso o AWS.

Para além desta lógica, a *framework* oferece também uma *sandbox* local. Isto permite que se implemente funções Lambda numa máquina de desenvolvimento sem ter que implantá-las antecipadamente no AWS [37]. Esta capacidade de executar e depurar num ambiente local aumenta significativamente a velocidade de desenvolvimento e permite encontrar problemas numa fase inicial do desenvolvimento.

Up

A Up[38] tem como principal objetivo simplificar a implantação de aplicações *web serverless*. Atualmente fornece suporte a Node.js, Golang, Python, Java, Crystal, Clojure e aplicações estáticas. Apesar de ter como objetivo vir a tornar-se uma plataforma agnóstica [38], está, hoje em dia, mais enriquecida no que diz respeito a suportar componentes da AWS.

Em conjunto com uma variada lista de comandos que aceleram esta implantação, esta biblioteca oferece também [38]:

- isolamento de pedidos - erros individuais lançados durante um pedido à aplicação são isolados em contentores, garantindo que nenhum outro cliente é afectado por uma falha
- escalonamento infinito - escalonamento automático, não sendo necessária a escrita de código ou a configuração do mesmo
- infraestrutura imutável - as implantações feitas pela Up são imutáveis, sendo possível a reversão para versões anteriores de forma quase instantânea

Para além disso, tem também um plano *premium* através do qual são disponibilizadas mais funcionalidades, como a monitorização e escrita de *logs* durante o processo de implantação.

Pulumi

A Pulumi é uma plataforma IAC que tem como objetivo facilitar a implantação de aplicações em diferentes plataformas *serverless* [39], onde se inclui Google Functions, Azure Functions, AWS Lambda e Kubernetes.

Esta ferramenta permite a escrita de código em variadas linguagens (JavaScript, TypeScript, Python, Go e .NET) convertendo-o de forma automática em recursos dos diferentes serviços *cloud*. Estes programas descrevem então como a infraestrutura da aplicação deve ser composta. Para isso, são alocados recursos dos serviços *cloud* cujas propriedades correspondem ao estado desejado da infraestrutura. Estas propriedades também podem ser utilizadas para lidar com dependências entre recursos. [39]

Cada aplicação tem um projeto associado, que consiste num diretório que contém o código-fonte e metadados sobre como executar a aplicação. Depois de desenvolvido o código fonte, é executado um comando que cria uma instância modular e configurável, conhecida como *stack*. Esta *stack* é semelhante a um ambiente de implantação (e.g dev, qa, produção) que pode ser utilizado para testar a aplicação em diferentes estados de implantação, de forma a assegurar o bom funcionamento da mesma antes desta ser disponibilizada ao utilizador.

Jets

Nos primórdios da computação *serverless* não havia qualquer tipo de suporte para a linguagem Ruby. Foram precisos 4 anos para que este suporte surgisse, na forma de *templates Serverless Application Model* (SAM) [40]. No entanto o processo de desenvolvimento de uma aplicação *serverless* em Ruby ainda era bastante complexo, o que vai contra as ideologias da própria linguagem. De forma a combater esta complexidade surgiu a *framework* Ruby on Jets. [41]

Para isso, esta *framework* combina a experiência que é implementar uma aplicação Rails [42], uma *framework* que facilita o desenvolvimento de aplicações *web* em Ruby, com a implantação desta no AWS Lambda e serviços relacionados onde se inclui a API Gateway e a DynamoDB. Esta combinação é feita através da conversão automática de código Rails em funções Lambda, que depois é integrado com os restantes serviços [41]. Na figura 2.16 é possível visualizar um exemplo típico de uma arquitetura desenvolvida em Jets, sendo possível observar também as diferentes interações entre os diferentes serviços da Amazon e o próprio utilizador.

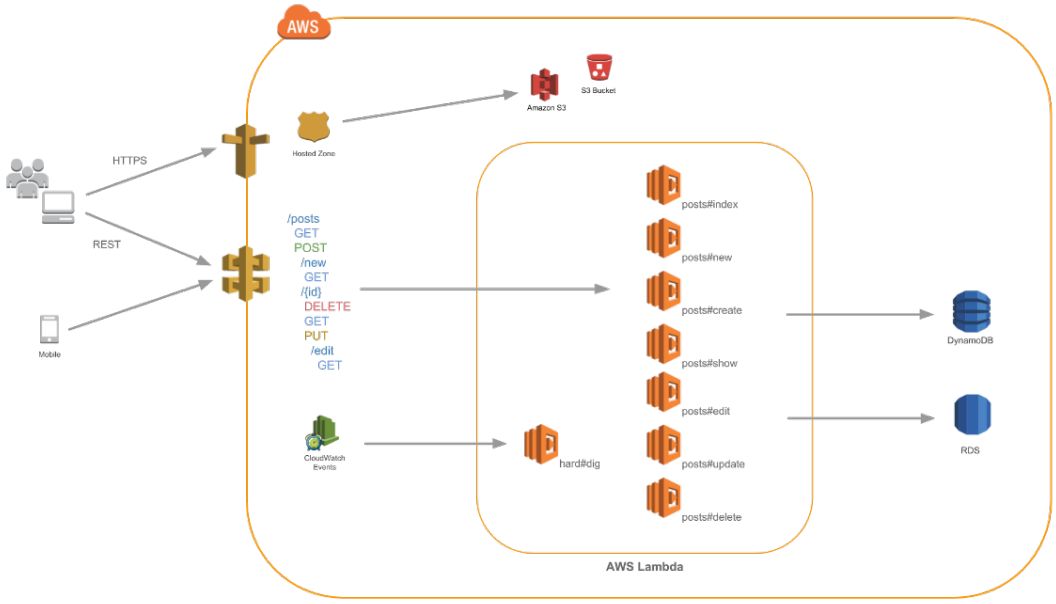

Figura 2.16: Exemplo de arquitetura de uma aplicação desenvolvida em Jets [41]

Middy

A Middy[43] é um *middleware* que permite simplificar o código do AWS Lambda quando se utiliza Node.js. Permite ao utilizador focar-se unicamente na lógica de negócio e numa fase posterior anexar elementos comuns como autenticação,autorização e validação de uma forma modular e reutilizável [43] .

Em *serverless* existe um maior foco no desenvolvimento da lógica de negócio das aplicações. No entanto há preocupações que ainda afastam os utilizadores deste desenvolvimento, como

a validação de *inputs* e a gestão de erros. Muitas das vezes estas necessidades e preocupações acabam por poluir aquela que é a lógica de negócio mais pura, fazendo com que o código seja mais difícil de ler e manter [43].

Nas *frameworks web* este problema é normalmente resolvido através do padrão *middleware*. Este padrão permite isolar estas preocupações técnicas em etapas modulares independentes que são anexadas à aplicação como se de uma configuração se tratasse, evitando assim a poluição do código da lógica de negócio [43]. O objetivo do Middy passa por implementar este tipo de padrão nas funções Lambda, que até aqui não tinha este tipo de suporte.

Esta *framework* implementa uma lógica semelhante a uma cebola, apresentada na figura 2.17. Como é possível observar, os *middlewares* têm duas etapas: antes e depois da execução da função lambda. Isto permite que cada *middleware* possa interagir com o pedido (evento) e com a resposta. Neste padrão de cebola os *middlewares* são adicionados à camada de lógica de negócio numa metodologia *Last In, First Out* (LIFO) [43]. Assim, se existirem três *middlewares* associados a uma função, como na figura abaixo, a ordem de execução é a seguinte: *middleware* 1, *middleware* 2, *middleware* 3, função, *middleware* 3, *middleware* 2, *middleware* 1.

De notar então que os *middlewares* são executados em ordem invertida. Desta forma o primeiro *middleware* adicionado é o primeiro a conseguir alterar o pedido e o último a conseguir alterar a resposta antes de ser enviada para o utilizador [43]. Na figura 2.17 é apresentada então a ordem de execução dos *middlewares* neste tipo de padrão.

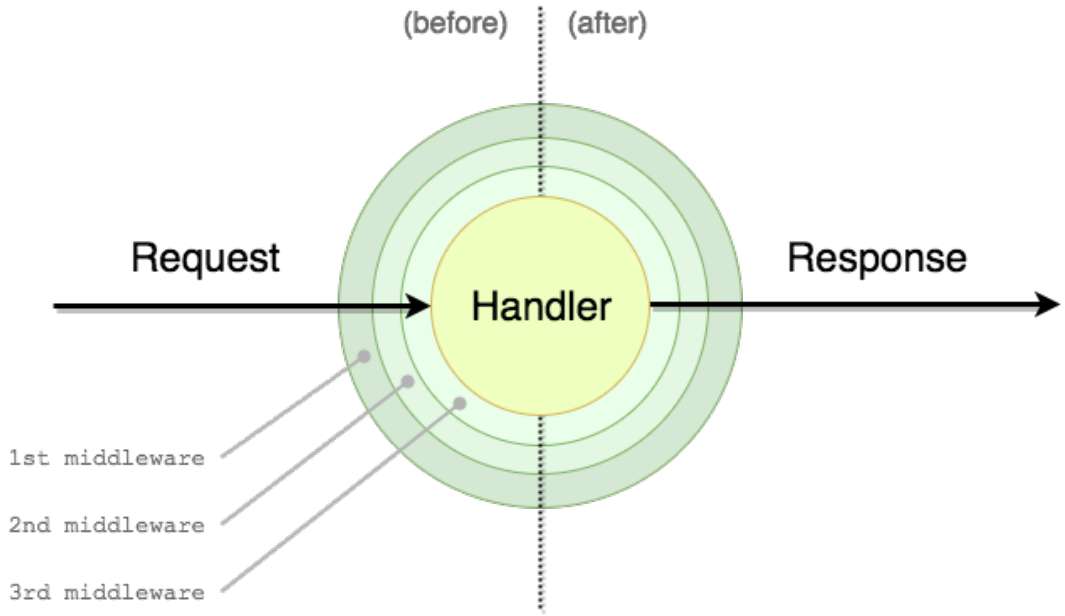

Figura 2.17: Padrão de *middlewares* em cebola [43]

Sigma

No âmbito de teste e depuração de aplicações *serverless*, ainda que este seja um tema pouco estudado e desenvolvido, existem já algumas ferramentas que visam fornecer suporte nesta área, como é o caso do Sigma. O Sigma é um *Integrated Development Environment* (IDE) baseado em *browser* desenvolvido de forma a facilitar o desenvolvimento, teste, depuração e implantação de funções Lambda [44]. Atualmente suporta o desenvolvimento em Node.js e

Python e oferece a geração de código através da funcionalidade de *drag and drop* para muitos dos recursos da AWS, entre os quais API Gateway, S3, DynamoDB, Kinesis, EventBridge e IoT.

Possui um ambiente de teste que permite aos utilizadores testar o código à medida que o escrevem, com a possibilidade de testar o código na plataforma AWS quase em tempo real [44]. Oferece também um kit de ferramentas baseado em comandos que oferece suporte na construção e implantação dos projetos existentes. Para além disso é também o primeiro IDE a oferecer a capacidade de percorrer e depurar funções Lambda à medida que são executadas ao vivo na plataforma AWS.

PureSec

No desenvolvimento em nuvem uma das principais preocupações dos utilizadores é a segurança das suas aplicações. Assim, de forma a minimizar e combater o pressuposto de que os dados de uma aplicação *cloud* são mais suscetíveis a ataques, surgiu uma ferramenta a que se deu o nome de PureSec.

A PureSec criou uma plataforma cuja *firewall* deteta e previne ataques na camada das funções sem impactar a performance da aplicação [45]. Para isso o seu motor de deteção é capaz de inspecionar diferentes *triggers* de eventos como Cloud Storage, sistema de mensagens Pub/Sub, entre outros.

A sua biblioteca FunctionShield permite aos programadores reforçar os mecanismos de segurança de forma a implementar os casos de uso mais comuns [45]. Suporta o desenvolvimento em Node.js, Python e Java e alguns dos seus benefícios são:

- previne a exposição de dados sensíveis através da monitorização do tráfico de rede das funções
- previne a exposição de código fonte da lógica de negócio
- possibilidade de configurar um modo de alerta que permite bloquear a execução quando algum tipo de política é violado
- adiciona apenas um milissegundo de latência à execução geral da aplicação
- suporta várias plataformas *cloud*, como AWS Lambda, Google Functions, IBM Functions e Azure Functions

Na tabela 2.2 é apresentado um resumo de todas as ferramentas e dos seus objectivo.

Tabela 2.2: Comparação entre as ferramentas e soluções analisadas

Ferramenta	Objetivo	Dependência de uma plataforma *cloud*	Implantação	Teste	Segurança
Serverless Framework	Facilitar a implantação das aplicações	-	X		
Architect	Simplificar a implantação de aplicações através de uma metodologia IAC	AWS	X		
Up	Simplificar a implantação de aplicações *web serverless*	AWS	X		
Pulumi	Simplificar a implantação de aplicações através de uma metodologia IAC	-	X		
Jets	Facilitar o desenvolvimento e implantação de aplicações *serverless* em Ruby	AWS	X		
Middy	Implementar o padrão de *middlewares* em cebola nas funções Lambda	AWS	X		
Sigma	Facilitar o desenvolvimento, teste, depuração e implantação de aplicações *serverless*	AWS	X	X	
PureSec	Detetar e prevenir ataques na camada das funções sem impactar a performance da aplicação	-			X

2.2.3 Dependência das plataformas cloud

Como foi possível perceber na secção anterior existe uma grande diversidade no que diz respeito à agnosticidade das diferentes plataformas, ainda que devido ao facto de o AWS Lambda ter sido lançado numa fase primária e estar, por isso, mais avançado é ainda considerável o número de *frameworks* dependentes desta plataforma. Importa então perceber como está a evoluir o mundo neste aspecto e quais são os efeitos e limitações da adoção de tecnologias dependentes da plataforma *cloud*.

A principal consequência da utilização de tecnologias não agnósticas é a obrigatoriedade de ficar preso a um único serviço quando as plataformas estão constantemente a evoluir e a adicionar funcionalidades que alteram o seu valor para o utilizador final, sendo por isso uma

mais valia a exploração das diferentes vantagens de cada plataforma de forma a obter o melhor de cada uma [46]. Um outro exemplo é o de uma empresa que compra uma empresa menor que usa um serviço *cloud* diferente e decide manter os dois pois na maioria das vezes a migração para o mesmo serviço é mais custosa que manter os dois serviços distintos [47]. Neste ponto há uma pergunta que importa responder: que soluções existem se quisermos compor serviços *cloud* distintos numa única aplicação?

De forma a dar resposta a esta pergunta começaram a surgir, nos últimos anos, reflexões sobre um conceito a que se deu o nome de *multi-cloud*, que consiste na utilização de múltiplos serviços de computação em nuvem numa única arquitetura. Assim, o objetivo passa por eliminar a dependência de um único serviço da nuvem através da distribuição de todo o *software* e aplicações por diferentes serviços [48]. Um ambiente *multi-cloud*, exemplificado na figura 2.18, pode ser todo ele privado, público ou uma combinação dos dois.

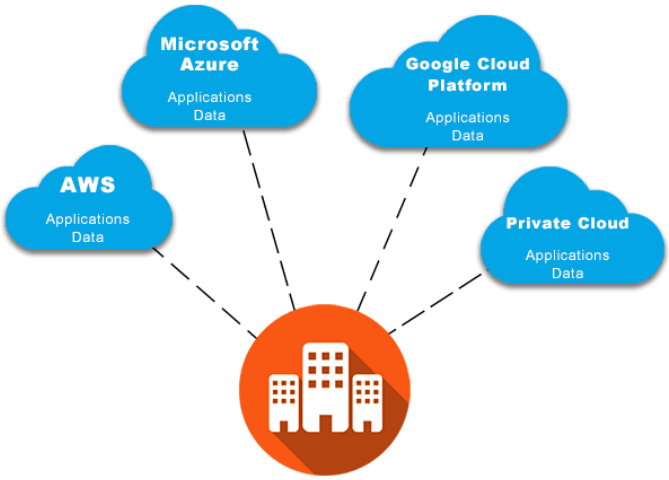

Figura 2.18: Arquitetura *multi-cloud* Fonte: [48]

Uma estratégia *multi-cloud* permite às empresas selecionar diferentes serviços para realizar diferentes tarefas, maximizando assim as qualidades de cada serviço. Para além deste, há outros benefícios que devem ser mencionados [48]:

- flexibilidade e capacidade de escolha, com a consequente independência de um serviço de nuvem em específico e seleção de diferentes de arquiteturas para diferentes componentes da lógica de negócio
- prevenção de desastres, pois a existência de vários ambientes assegura que existam diferentes recursos de computação e de armazenamento de dados disponíveis de forma a evitar o tempo de inatividade

Apesar destas vantagens, há alguns pontos que são aconselhados [48] quando é utilizada uma abordagem *multi-cloud* tais como:

- a utilização de uma única linguagem, tipicamente JavaScript, na codificação das funções nas diferentes plataformas, de forma a aumentar a manutibilidade, havendo algumas exceções quando a lógica de negócio assim o exige

- uma definição bem clara das responsabilidades de cada serviço no que à segurança diz respeito, pois a existência de diferentes serviços pode complicar a tarefa de manter a informação segura e consistente entre as diferentes plataformas

2.3 Transição para uma arquitetura serverless

Ainda que a transição de uma arquitetura atual, como é o caso das arquiteturas orientadas a microserviços, para uma arquitetura *serverless* aparente oferecer algumas vantagens (que serão aferidas ao longo do documento), o processo de mudança ainda não é trivial e acarreta algumas dificuldades. Por este motivo, existe um conjunto de recomendações [49] que devem ser adotadas no desenvolvimento das aplicações e respetivas arquiteturas com vista a eliminar ou minimizar as dificuldades sentidas na adoção do desenvolvimento *serverless*, tais como:

- R1 - Devem ser identificados os problemas que a arquitetura atual não consegue resolver e que se espera que sejam resolvidos pela arquitetura *serverless* (por exemplo custos de operação reduzidos, escalabilidade, etc.)
- R2 - Depois de identificados estes problemas, torna-se relativamente mais fácil de identificar as tecnologias que permitem resolver os problemas identificados. Devem ser identificadas estas tecnologias e oferecida formação às equipas da organização que permita que estas equipas aprendam estas tecnologias e sejam educadas acerca do desenvolvimento *serverless*
- R3 - De forma a validar se estas tecnologias permitem resolver os problemas identificados, deve ser criada uma prova de conceito que permita validar a solução proposta
- R4 - Deve ser tido em conta que a mudança para uma arquitetura *serverless* implica uma mudança para uma arquitetura orientada a eventos, pelo que por vezes esta mudança implica repensar a forma como a arquitetura está organizada e reorganizá-la de acordo com este paradigma
- R5 - A automatação é um dos princípios do desenvolvimento *serverless* pelo que a fase de testes deve ser automatizada tanto quanto possível
- R6 - Devem ser criados *scripts* de implantação que facilitem ainda mais a implantação de novos serviços e funções

Capítulo 3

Análise de Valor

Neste capítulo é feita uma análise de valor da solução a desenvolver através do modelo *New Concept Devolopment* e descritos o valor, valor para o cliente, valor percecionado e proposta de valor desta solução. Para além disso é apresentado um modelo Canvas que descreve o modelo de negócio da organização e solução, é analisado o modelo de negócio através do modelo de Verna Allee e utilizado o método AHP como suporte para a decisão da ideia a desenvolver como caso de estudo.

3.1 New Concept Development Model

O processo de inovação, apresentado na figura 3.1, pode ser dividido em 3 fases distintas: *Front End of Innovation* (FEI), *New Product Development* (NPD) e comercialização. [50]

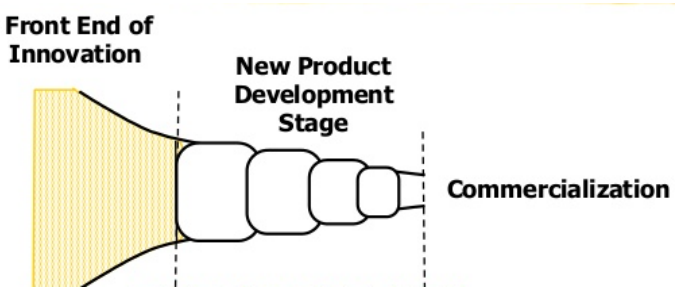

Figura 3.1: Etapas do processo de inovação Fonte: [50]

A primeira etapa (FEI) é onde as oportunidades são identificadas e os conceitos desenvolvidos, antes de entrar no processo formal de desenvolvimento do produto. O modelo *New Concept Development* (NCD), apresentado na figura abaixo, tem como objetivo principal fornecer uma linguagem simples e a terminologia necessária para compreender o FEI [51] .É composto por 3 partes fundamentais [51]:

- motor - corresponde à liderança, cultura e estratégia de negócio da organização que conduz os elementos chave
- elementos chave - *Opportunity Identification*, *Opportunity Analysis*, *Idea Generation and Enrichment*,*Idea Selection* e *Concept Definition*
- fatores ambientais - responsáveis por influenciar o motor e os elementos chave, sendo alguns exemplos as têndencias de mercado,as tecnologias e a competição, e geralmente não controlados pela organização

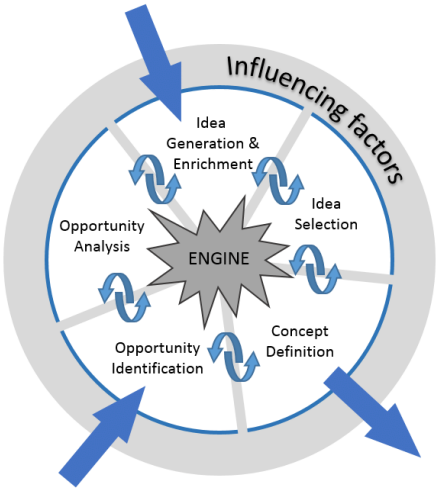

Figura 3.2: Modelo NCD Fonte: [52]

De seguida serão enquadrados os elementos chave deste modelo com o projeto que é aqui relatado.

3.1.1 Opportunity Identification

Fase em que são identificados problemas e oportunidades aos quais a organização pretende dar resposta. É assim definido o mercado e ou tecnologia em que a organização tem potencial interesse [51].

Para este projeto foi identificada a oportunidade de desenvolvimento e implantação de software seguindo a metodologia *serverless*.

3.1.2 Opportunity Analysis

Nesta etapa é confirmado se vale realmente a pena explorar a oportunidade identificada [51].

Como já demonstrado na secção 2.2.4, o mercado *serverless* está em expansão e tende a crescer cada vez mais, pelo que faz sentido perceber onde é que este tipo de tecnologia se encaixa e quais são os processos e abordagens para tirar o máximo partido das suas vantagens.

3.1.3 Idea Generation and Enrichment

Momento em que as oportunidades identificadas são transformadas no nascimento, desenvolvimento e maturação de uma ideia concreta, aliada às necessidades dos clientes e às possibilidades da organização [51].

De acordo com os mercados para os quais a tecnologia *serverless* é mais útil, levantados na secção 2.2.4, foram identificados alguns problemas que pudessem ser resolvidos neste contexto. Assim, foram identificadas as seguintes ideias para o caso de estudo a desenvolver:

- portal de notícias
- sistema de gestão de um hospital
- plataforma *e-Commerce*

3.1.4 Idea Selection

Fase em que são seleccionadas as ideias a desenvolver de forma a obter o maior valor possível para as entidades envolvidas no projecto [51].

Para esta seleção foi utilizado o método *Analytic Hierarchy Process* (AHP), apresentado na secção 3.8, sendo obtido como resultado o desenvolvimento de uma plataforma *e-Commerce*.

3.1.5 Concept Definition

Etapa onde a definição de conceito é apresentada formalmente e onde se evidenciam as vantagens do produto.

O caso de estudo a ser desenvolvido nesta dissertação está bastante bem definido, uma aplicação *web* na área do *e-Commerce*, para o qual a escalabilidade automática e a rapidez no desenvolvimento de novas funcionalidades e na execução de tarefas proporcionadas pela tecnologia *serverless* são uma grande mais valia.

3.2 Valor

O valor pode ser definido como "todos os fatores, qualitativos e quantitativos, subjetivos e objetivos, que compõe a experiência completa de compra"[53].

Se forem tidos em conta os possíveis resultados da solução a desenvolver, há varios fatores que podem então trazer valor, onde se inclui:

- com a possibilidade de a equipa de desenvolvimento se focar na lógica de negócio, vão ser desenvolvidas novas funcionalidades mais rapidamente e com maior taxa de sucesso, diminuindo os erros e permitindo uma disponibilidade maior, o que traz uma maior satisfação dos utilizadores
- a escalabilidade automática que as tecnologias *serverless* oferecem, principalmente neste segmento de mercado em que a procura é grande, permite uma maior disponibilidade das aplicações o que faz com que os clientes fiquem mais satisfeitos com os produtos

3.3 Valor para o cliente

O valor para o cliente é a diferença entre as percepções do cliente quanto aos benefícios e quanto aos custos da compra e uso de um produto ou serviço [54].

O cliente espera que:

- a solução possua uma interface intuitiva e responsiva
- a segurança seja uma prioridade (e.g definição de pagamento limite a partir de uma conta, requisição do *Card Verification Value* (CVV) do cartão de crédito, entre outros)
- o sistema possua alta disponibilidade
- exista uma garantia de resposta em tempo útil

3.4 Valor percecionado

O valor percecionado é a razão entre benefícios percecionados e sacrifícios percecionados pelo cliente, onde sacrifícios percecionados inclui todos os custos, tais como preço de compra, instalação e manutenção, riscos de falha ou de mau desempenho, enquanto que os benefícios percecionados são uma combinação dos atributos físicos, atributos do serviço e suporte ao consumidor, entre outros indicadores de qualidade [53].

De acordo com alguns dos benefícios referidos anteriormente, importa referir que a tecnologia *serverless* permite reduzir alguns dos sacrifícios tais como: o risco de falha ou de mau desempenho, devido à maior disponibilidade que esta tecnologia oferece, e os custos no geral, pois como a organização terá uma redução nos custos de desenvolvimento e implantação bastante significativa o investimento que se quer ver retornado é menor levando a que os custos não precisem de ser tão altos.

3.5 Proposta de Valor

"A proposta de valor de uma marca é uma afirmação dos benefícios funcionais, emocionais e de auto-expressão oferecidos pela marca que proporcionam valor ao cliente. A proposta de valor eficiente deverá conduzir a um relacionamento marca-cliente e impulsionar as decisões de compra"[55].

Desta forma, a proposta de valor desta solução passa por oferecer aos consumidores um processo de compra intuitivo, seguro e rápido. Para além disso é também um progresso naquilo que é a documentação e estudo do desenvolvimento em *serverless*.

3.6 Canvas

O modelo de negócio pode ser definido como a descrição do valor que uma organização oferece a um ou várias segmentos de cliente e a arquitetura da empresa e a sua rede de parceiros para criar, comercializar e entregar esse valor a fim de gerar fluxos de receitas rentáveis e sustentáveis [56].

Em 2010, Osterwalder e Pigneur [57] redefiniram os componentes e configurações de um modelo de negócio com o modelo a que deram o nome de Canvas. Este modelo, apresentado na figura 3.3, divide a organização em nove tópicos relacionados com a geração de valor por parte da mesma [57].

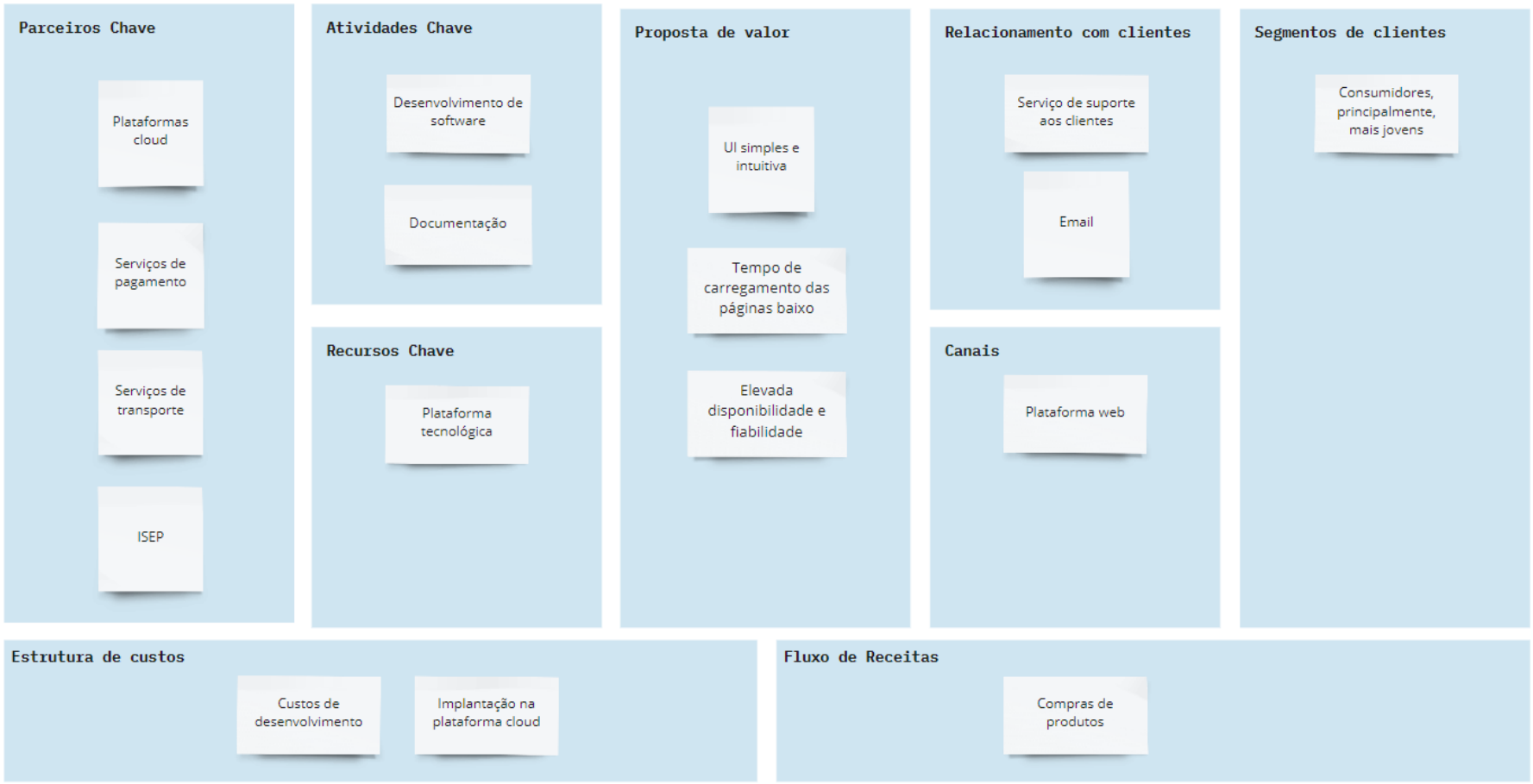

Figura 3.3: Modelo Canvas

Enumerando então os nove tópicos deste modelo temos:

- **Parceiros chave**: Instituto Superior de Engenharia do Porto (ISEP) e parceiros do mercado *e-Commerce* e *serverless*, como as plataformas *cloud* e os serviços de pagamento e transporte
- **Atividades chave**: as principais atividades desta solução são o desenvolvimento de software e a sua documentação
- **Recursos chave**: o recurso principal da solução a desenvolver é a sua plataforma tecnológica e arquitectura
- **Proposta de valor**: como referido na subsecção anterior, a proposta de valor passa por ter uma solução com uma interface simples e intuitiva, tempos de espera reduzidos e uma alta disponibilidade e fiabilidade
- **Relacionamento com clientes**: de forma a fazer um acompanhamento dos clientes será utilizado um serviço de suporte e o *email*
- **Canais**: a solução é disponibilizada aos clientes através da plataforma *web* a desenvolver
- **Segmentos de clientes**: como clientes alvo temos os consumidores no geral, com principal foco para os mais jovens visto que são os que mais utilizam este tipo de plataformas
- **Estrutura de custos**: em termos de custos estão previstos os custos de desenvolvimento e os de utilização e implantação da solução na plataforma *cloud*
- **Fluxo de receitas**: as receitas, hipoteticamente, previstas serão as compras de produtos por parte de clientes

3.7 Modelo de Verna Allee

O modelo apresentado por Verna Allee descreve os negócios como redes de valor, que podem ser definidas como "papéis e interações em que as pessoas se envolvem em trocas tangíveis e intangíveis de forma a obter um bem estar económico e social"[58].

Desta forma, é criado um mapa que permite analisar esta rede de valor, que contém 3 componentes principais [58]:

- **nós** - os nós da rede de valor são representados por uma oval e simbolizam entidades específicas que contribuem para a atividade da organização, desempenhando um papel na mesma
- **trocas** - cada seta entre nós simboliza uma transacção de valor entre duas entidades, que dependendo da forma como ocorre a entrega de valor pode fazer com que a seta seja desenhada de forma contínua ou descontínua
- **entregas** - as descrições em cada linha descrevem a forma específica de como a transação ocorre, sendo que se considera como tangíveis trocas que envolvam bens, serviços e dinheiro e como intangíveis aqueles que representam ganhos que vão para além do produto ou serviço

No contexto da solução a desenvolver, a rede é relativamente simples, com a presença de três identidades: o consumidor, a solução e a(s) plataforma(s) *cloud* utilizada(s). Na figura 3.4, é possível visualizar as relações entre as diferentes identidades.

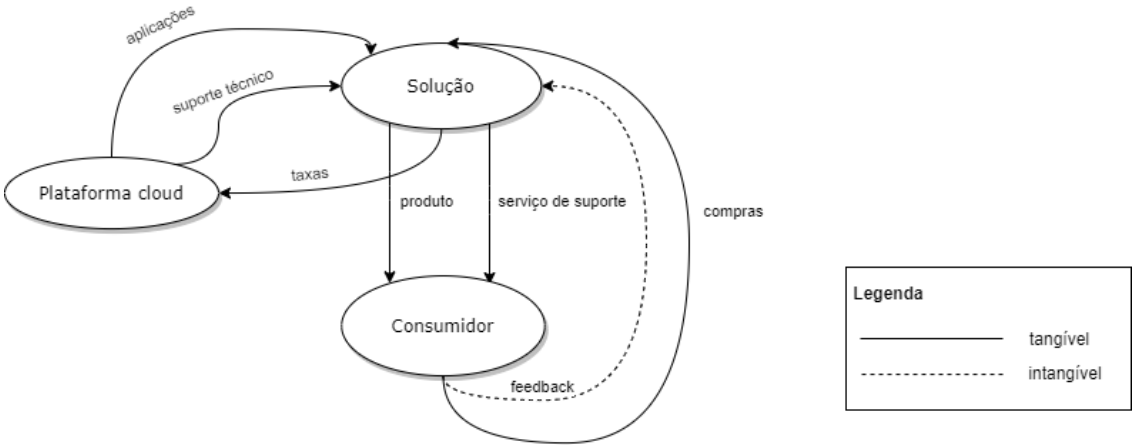

Figura 3.4: Modelo de Verna Allee

3.8 Método AHP

Introduzido por Thomas Saaty, o método AHP é uma teoria geral de medição. É utilizado para derivar escalas de razão de comparações pareadas discretas e contínuas. Estas comparações podem ser feitas a partir de medidas reais ou de uma escala fundamental que reflita a força relativa de preferências e sentimentos. [59]

A primeira etapa deste método, consiste na construção de uma árvore hierárquica, que é constituída pelo objetivo da decisão no topo, seguido pelos critérios de avaliação associados ao problema e por fim as alternativas disponíveis para solucionar o problema. [59]

Neste contexto, temos então como objetivo da decisão a seleção da ideia a desenvolver,como alternativas um portal de notícias, um sistema de gestão de um hospital e uma plataforma *e-Commerce* e como critérios:

- **tempo e esforço de desenvolvimento** - é desejado que o tempo e esforço de desenvolvimento seja relativamente baixo, pois permite que exista também tempo para realizar outras atividades como a análise do desempenho e dos requisitos não funcionais da solução, de forma a obter uma comparação entre o desenvolvimento *serverless* e tradicional
- **relevância** - a solução a desenvolver deve ter alguma relevância para o estudo a realizar e deve permitir obter comparações fidedignas com as soluções existentes no mercado
- **capacidade serverless** - representa a utilidade que a tecnologia *serverless* terá na solução, com base em factores como a necessidade de escalabilidade e de atualizações constantes

Na figura 3.5 está representada a árvore hierárquica de decisão deste problema.

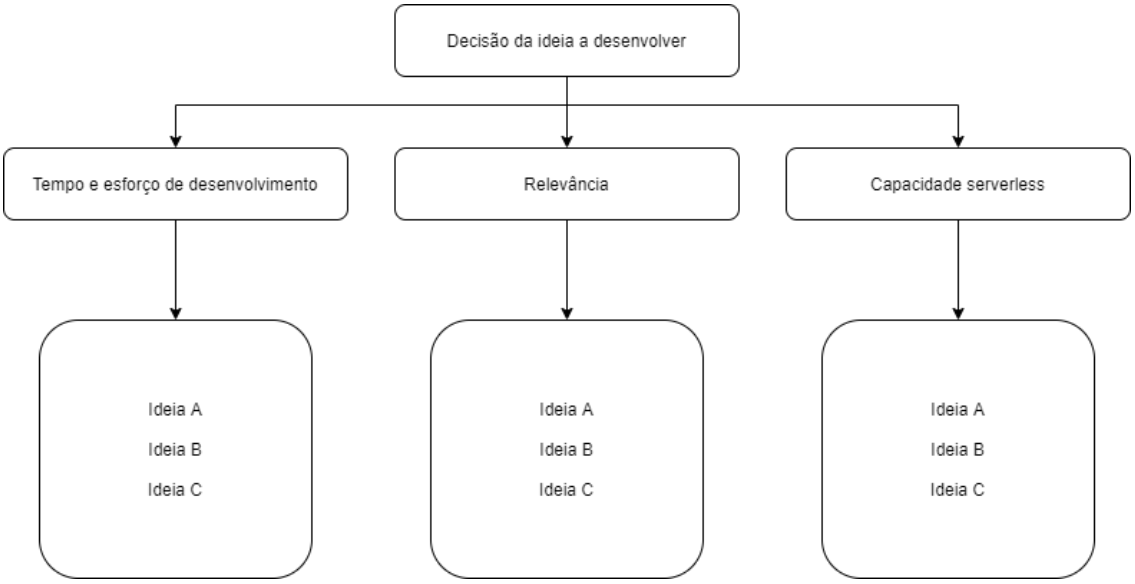

Figura 3.5: Árvore hierárquica de decisão, constituída por três critérios

Posteriormente, são estabelecidas as prioridades entre os diferentes critérios de decisão, através de uma matriz de comparação. O valor de prioridade pode variar entre 1 e 9 e o significado de cada valor está definido numa escala a que Saaty deu o nome de Escala Fundamental, que pode ser visualizada na tabela 3.1.

Tabela 3.1: Escala Fundamental [59]

Escala	**Avaliação**	**Recíproco**	**Descrição**
Igual importância	1	1	Os dois critérios contribuem igualmente para os objetivos
Importância moderada	3	1/3	A experiência e o julgamento favorecem um critério levemente sobre outro
Mais importante	5	1/5	A experiência e o julgamento favorecem um critério fortemente em relação ao outro
Muito importante	7	1/7	Um critério é fortemente favorecido em relação a outro e pode ser demonstrado na prática
Importância extrema	9	1/9	Um critério é favorecido em relação a outro com o mais alto grau de certeza
Valores intermediários	2,4,6,8		Quando se procura condições de compromisso entre duas definições

Com base nesta escala foi então desenvolvida a matriz de comparação, apresentada na tabela 3.2.

Tabela 3.2: Matriz de comparação de critérios

Critério	**Tempo e esforço de desenvolvimento**	**Relevância**	**Capacidade serverless**
Tempo e esforço de desenvolvimento	1	1/3	1/2
Relevância	3	1	2
Capacidade serverless	2	1/2	1

Depois de aplicado o método de Eigenvector [59], foram obtidas as seguintes prioridades relativas para cada critério, que estão apresentadas na tabela 3.3.

Tabela 3.3: Prioridade relativa de cada critério

Critério	**Prioridade relativa**
Tempo e esforço de desenvolvimento	0.163
Relevância	0.54
Capacidade serverless	0.297

Seguidamente é calculado o Grau de Consistência (GC) de forma a avaliar a consistência destas prioridades relativas. Este é obtido através da razão entre o Índice de Consistência (IC) e um Índice Aleatório (IA). Os cálculos necessários para obter estes valor encontram-se apresentados abaixo.

IC pode ser obtido a partir da seguinte função:

$$IC = \frac{\lambda \max - n}{n - 1}$$

O valor de n corresponde ao número de critérios e λmax pode ser calculado a partir do seguinte cálculo:

$$\begin{bmatrix} 1 & 1/3 & 1/2 \\ 3 & 1 & 2 \\ 2 & 1/2 & 1 \end{bmatrix} \times \begin{bmatrix} 0.163 \\ 0.54 \\ 0.297 \end{bmatrix} = \begin{bmatrix} 0.491 \\ 1.623 \\ 0.893 \end{bmatrix}$$

$$\lambda \max = média\{\frac{0.491}{0.163}, \frac{1.623}{0.54}, \frac{0.893}{0.297}\} = 3,00816$$

Depois de calculado o valor de λmax, é possível calcular o valor do indíce IC.

$$IC = \frac{3,00816 - 3}{3 - 1} = 0.004$$

Finalmente é calculado o valor de *GC*. Para isso é necessário obter o valor de IA correspondente a n = 3, definido na tabela de Saaty [59], que é 0.58 .

$$GC = \frac{0.004}{0.58} = 0.00690$$

Obteve-se assim um grau de consistência de aproximadamente 0.00690, que é menor que 0.1, pelo que se pode afirmar que os valores das prioridades relativas são consistentes e de confiança.

De seguida é feita uma comparação entre cada ideia, sendo atribuído para cada critério um valor da escala fundamental. Para esta comparação foi atribuído uma letra a cada ideia:

- **A** - portal de notícias
- **B** - sistema de gestão de um hospital
- **C** - plataforma *e-Commerce*

Na tabela 3.4 é apresentada a matriz de comparação para o critério tempo e esforço de desenvolvimento. Neste ponto a plataforma *e-Commerce* acabou por obter uma maior prioridade pois as outras soluções são sistemas que possuem múltiplos elementos arquiteturais complexos, pelo que exigem mais tempo de desenvolvimento.

Tabela 3.4: Matriz de comparação de ideias relativamente a tempo e esforço de desenvolvimento

	A	**B**	**C**	**Prioridade relativa**
A	1	2	1/2	0.297
B	1/2	1	1/3	0.163
C	2	3	1	0.54

Na tabela 3.5 é apresentada a matriz de comparação para o critério relevância. Nesta comparação as ideias A e C obtiveram resultados semelhantes, com a ideia B a obter uma menor prioridade pois as funcionalidades de um sistema de gestão hospitalar variam bastante de acordo com as unidades hospitalares, pelo que seria bastante complicado generalizar este desenvolvimento ao ponto de o tornar um exemplo para este tipo de soluções, pelo que o seu desenvolvimento acaba por ser menos relevante.

Tabela 3.5: Matriz de comparação de ideias no que diz respeito à sua relevância

	A	**B**	**C**	**Prioridade relativa**
A	1	2	1	0.4
B	1/2	1	1/2	0.2
C	1	2	1	0.4

Na tabela 3.6 é apresentada a matriz de comparação para o critério relevância. Neste comparação as ideias A e C obtiveram novamente resultados semelhantes, com a ideia B a obter uma menor prioridade devido ao facto de num sistema de gestão hospitalar existir um número de utilizadores que acede ao sistema mais fixo do que em comparação com as outras duas ideias, onde há uma variação deste número bastante elevada.

Tabela 3.6: Matriz de comparação de ideias relativamente à capacidade *serverless*

	A	**B**	**C**	**Prioridade relativa**
A	1	3	1	0.429
B	1/3	1	1/3	0.143
C	1	3	1	0.429

Assim atualizando a árvore hierárquica de decisão com as prioridades relativas de cada ideia obtemos o resultado apresentado abaixo.

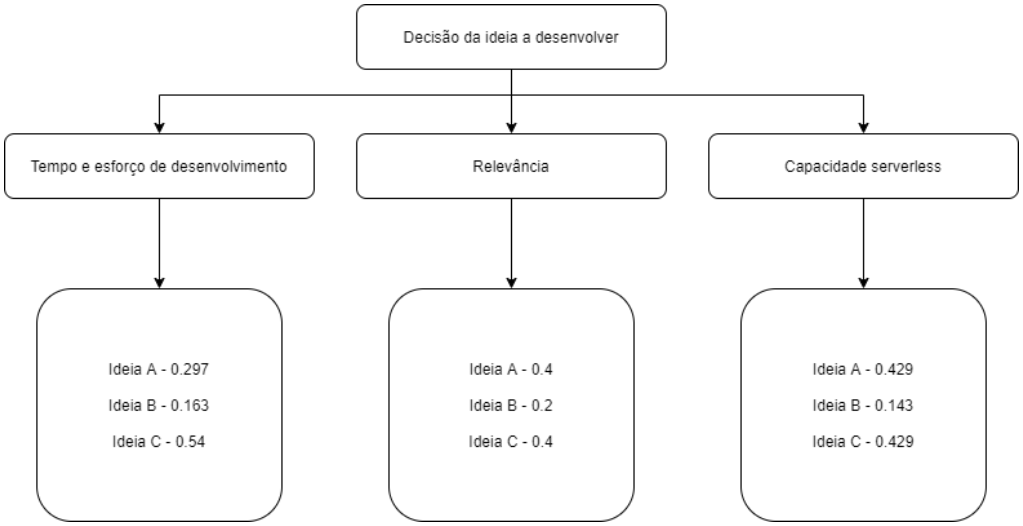

Figura 3.6: Árvore hierárquica de decisão, constituída por três critérios, atualizada com as prioridades relativas de cada ideia

Finalmente, para se obter a prioridade de cada ideia é multiplicada a matriz de prioridades relativas dos critérios pela matriz de prioridade relativas das alternativas. A ideia com a prioridade maior é a escolhida.

$$\begin{bmatrix} 0.297 & 0.4 & 0.429 \\ 0.163 & 0.2 & 0.143 \\ 0.54 & 0.4 & 0.429 \end{bmatrix} \times \begin{bmatrix} 0.163 \\ 0.54 \\ 0.297 \end{bmatrix} = \begin{bmatrix} 0.392 \\ 0.177 \\ 0.431 \end{bmatrix}$$

Resulta assim que a ideia escolhida é a C, ou seja, desenvolver uma plataforma *e-Commerce*.

Capítulo 4

Especificação do Caso de Estudo

Para o caso de estudo que permite avaliar a utilização de tecnologias *serverless*, tal como sugerido pela recomendação R4 descrita anteriormente, foi utilizado o exemplo de uma loja *e-Commerce*, mais concretamente uma loja de venda de artigos de arte.

Neste capítulo são apresentados os requisitos não funcionais e funcionais do caso de estudo a desenvolver, o modelo de domínio e os casos de uso do problema.

4.1 Engenharia de Requisitos

Os requisitos de um sistema consistem em especificações de serviços que o sistema deve oferecer, limitações do sistema e informações de *background* necessárias para desenvolver o sistema [60]. Em síntese, a engenharia de requisitos é o processo sistemático de descoberta, compreensão, análise e documentação destes requisitos [60].

Nesta secção é apresentada a análise de requisitos não funcionais e funcionais subjacentes ao caso de estudo. Estes requisitos são classificados de acordo com o modelo FURPS+, que é uma evolução do modelo FURPS que foi desenvolvido na HP por Robert Grady e Deborah Caswell [61]. A adoção deste modelo prende-se com o seu reconhecimento histórico e aprovação.

4.1.1 Requisitos não funcionais

Os requisitos não funcionais representam critérios específicos que podem ser utilizados para avaliar a operação do sistema, ao invés de comportamentos específicos [62], tais como usabilidade, confiabilidade, desempenho e suportabilidade.

Usabilidade

A usabilidade está relacionada com a interface gráfica que é apresentada ao utilizador e como esta reage quando interagida. [62] Neste contexto, existem alguns requisitos não funcionais relativos à usabilidade, tais como:

- **RNF01** - O sistema deve possuir um baixo tempo de espera para o utilizador
- **RNF02** - O sistema deve possuir uma interface simples e intuitiva
- **RNF03** - O utilizador deve conseguir realizar uma tarefa com rapidez depois de visualizar a interface
- **RNF04** - Em caso de falha, o utilizador deve ter uma noção clara do erro que ocorreu

Confiabilidade

Este atributo especifica a probabilidade de o sistema funcionar sem falha durante um determinado período de tempo num ambiente específico. Foram identificados os seguintes requisitos não funcionais relativos à confiabilidade:

- **RNF06** - O sistema deve possuir uma disponibilidade de 99%

Escalabilidade

A escalabilidade é a capacidade de um sistema lidar com um volume elevado de operações sem restrições ou *bottlenecks* [62]. No que diz respeito a esta característica, foram identificados os seguintes requisitos:

- **RNF07** - O sistema deve funcionar sem constrangimentos até um volume máximo de 1000 pedidos a cada função *serverless* por hora

Segurança

Consiste na proteção de sistemas e redes contra a divulgação de informações, roubo ou danos ao hardware, software e informação eletrónica, bem como da sua interrupção [62]. No que diz respeito à segurança, foram identificados os seguintes requisitos:

- **RNF09** - O sistema deve ser capaz de proteger toda a informação confidencial do utilizador, como por exemplo os dados de pagamento

Restrições de design

Como o nome indica, limita a forma como o sistema é desenhado. No que diz respeito a esta à segurança, foram identificados os seguintes requisitos:

- **RNF10** - O sistema deve ser desenhado e desenvolvido de forma iterativa e incremental

4.1.2 Requisitos funcionais

O requisito funcional representa uma função de um sistema, ou seja, representa o que o *software* faz em termos de tarefas e serviços [62]. De seguida são enumerados os requisitos funcionais da solução a desenvolver:

- **U01** - O sistema deve permitir que o utilizador visualize uma lista de todos os artigos
- **U02** - O sistema deve permitir que o utilizador adicione artigos ao carro de compras
- **U03** - O sistema deve permitir que o utilizador remova artigos do carro de compras
- **U04** - O sistema deve permitir que o utilizador faça uma encomenda através do seu carro de compras
- **U05** - O sistema deve permitir que o utilizador acompanhe o estado da sua encomenda

Com base nestes requisitos funcionais, foram identificados os casos de uso desta solução, apresentados na figura 4.1 através de um Diagrama de Casos de Uso.

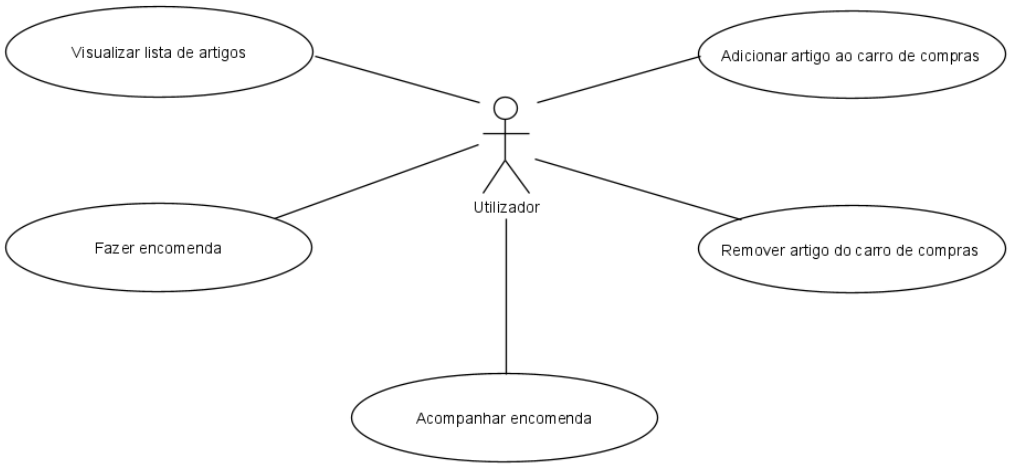

Figura 4.1: Diagrama de Casos de Uso

4.2 Modelo de Domínio

Um modelo de domínio consiste num sistema de abstracções que descreve determinados aspectos de um domínio e que pode ser utilizado para resolver problemas relacionados com este domínio [63].

O domínio deste caso de estudo, apresentado na figura 4.2, consiste numa loja que vende vários artigos de arte através de uma plataforma *e-Commerce*. O processo de compra inicia-se com a adição de itens ao carro de compras pelo utilizador, sendo que cada item corresponde a um artigo de arte e um artigo de arte pode ser referenciado em vários itens, por exemplo quando o utilizador pretende comprar dois exemplares do mesmo artigo. A partir deste carro de compras é iniciada uma nova encomenda, para a qual o utilizador define os seus dados de pagamento, concretizando o mesmo através de um dos serviços de pagamento disponíveis, e os seus dados de transporte, como a sua morada, código postal e país. A partir desta informação são definidos os serviços de transporte necessários para que o transporte da encomenda ocorra, sendo que para cada transporte é gerado um rastreador que permite ao utilizador acompanhar o estado da sua encomenda.

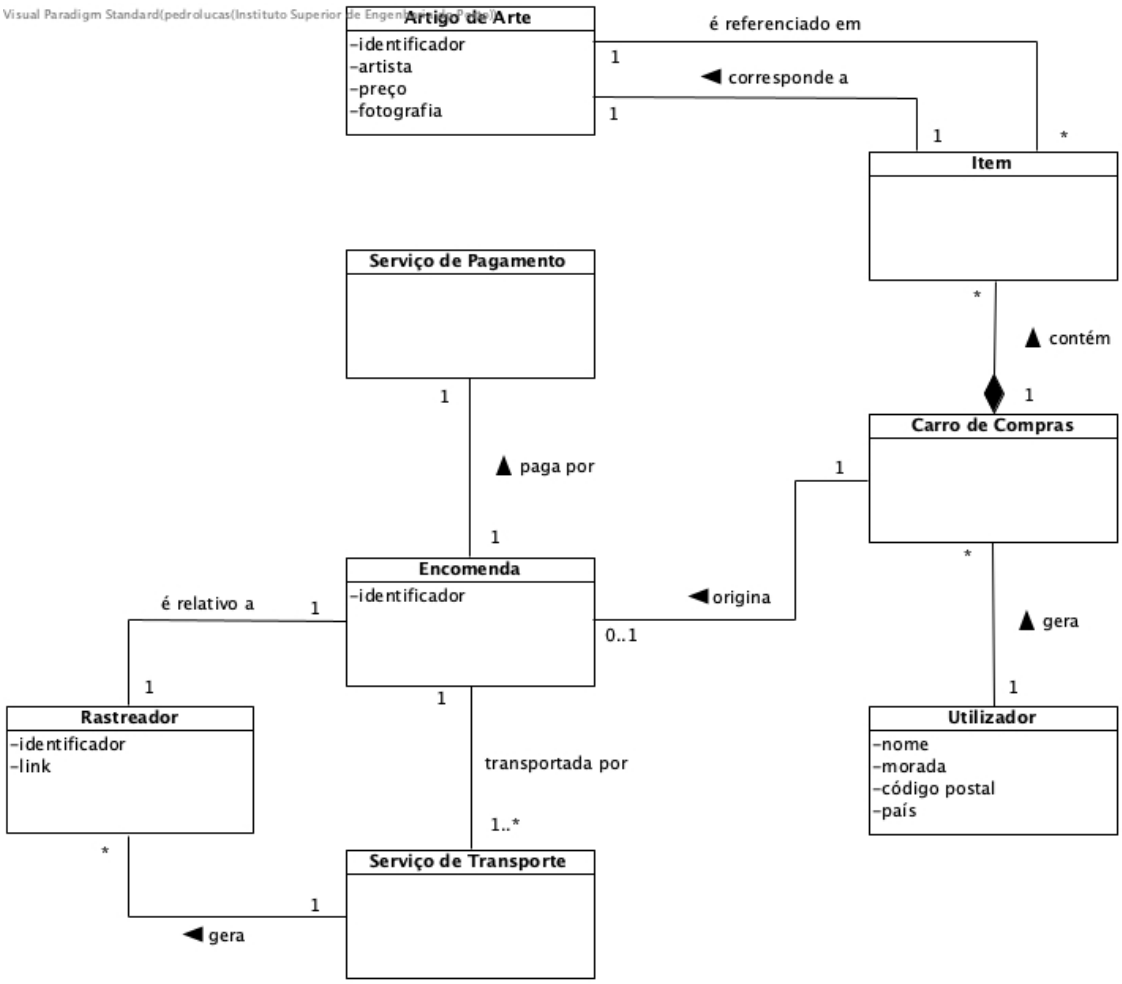

Figura 4.2: Modelo de domínio do caso de estudo

Capítulo 5

Design

Um dos objectivos desta dissertação passa por perceber o que une e o que distingue os métodos de desenvolvimento atuais, como é o caso de uma abordagem orientada a microserviços, e o desenvolvimento *serverless*. Ora, estas semelhanças e diferenças também existem ao nível do design e por esse motivo foi desenhado, sempre que possível, a solução a desenvolver para este caso de estudo tendo em conta duas abordagens, uma abordagem orientada a microserviços e uma abordagem *serverless*.

Este design feito para cada uma das abordagens permite compreender melhor as diferenças que a metodologia *serverless* traz e quais as principais mudanças de paradigma em relação aquilo que são os métodos de desenvolvimento atuais. Visto que alguns dos diagramas apresentados possuem uma dimensão considerável também são apresentados no Apêndice A, de forma a facilitar a sua leitura.

5.1 Arquitetura

Esta secção apresenta a arquitetura das duas abordagens e compara-as.

5.1.1 Abordagem orientada a microserviços

Uma possível arquitetura em microserviços seria a apresentada na figura 5.1. Esta arquitetura é composta por uma aplicação web que comunica com um API Gateway, componente esse que recebe todos os pedidos feitos pela aplicação web e redireciona-os para o serviço requisitado, abstraindo assim esta lógica de comunicação entre a a aplicação web e cada microserviço.

Esta arquitetura seria também composta por cinco microserviços:

- Serviço de Transporte - responsável pela lógica de transporte da encomenda até ao utilizador e que para gerir todo este processo comunicaria com os serviços de transporte externos suportados pela aplicação.
- Serviço de Pagamento - responsável pelo logística de pagamento da encomenda e por comunicar com os serviços de pagamento externos de forma a integrá-los com a aplicação.
- Serviço de Artigos de Arte - responsável pelo gestão de todos os artigos de arte vendidos na loja.
- Serviço de Artistas - responsável pelo gestão dos autores dos artigos vendidos na loja.
- Serviço de Encomendas - responsável pelo criação e gestão das encomendas da loja.

Em termos gerais, cada microserviço seria organizado em três camadas, sendo que a cada camada está associado um padrão de design: a primeira é a camada de controladores, que funciona como elemento de ligação entre a interface do utilizador e a lógica de negócio e que transmite o *input* do utilizador à camada de serviço. Segue-se então a camada de serviços, que é o *middleware* entre o controlador e o repositório e que recolhe a informação do controlador, efetua a sua validação e executa a lógica de negócio, recorrendo à camada de repositórios para concretizar a manipulação dos dados. Nos casos em que não existe esta manipulação de dados, deixa de existir a camada de repositórios.

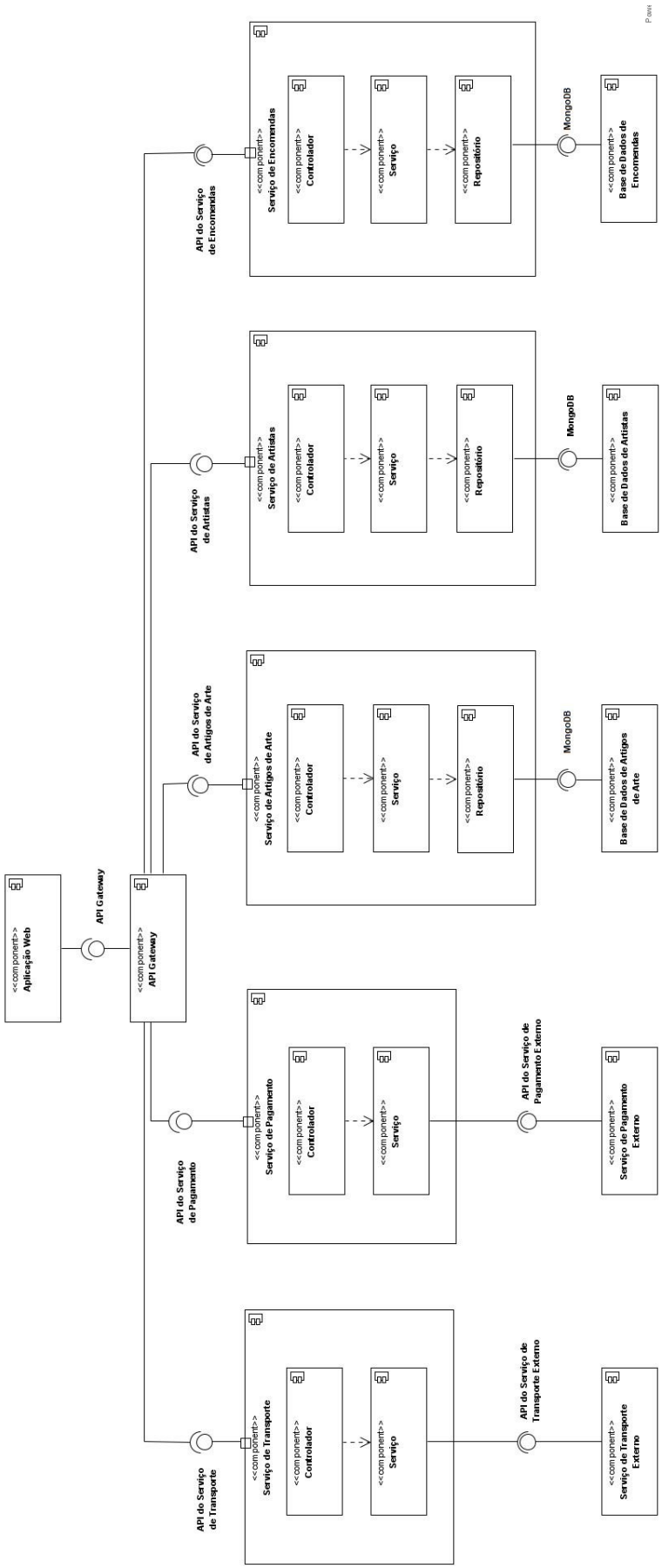

Figura 5.1: Arquitetura da abordagem orientada a microserviços

5.1.2 Abordagem serverless

Uma possível arquitetura serverless é a apresentada na figura 5.2. É composta por uma aplicação web que comunica com cada função de forma a realizar cada tarefa (de notar que outra alternativa poderia ser a existência de um API Gateway que gerisse esta logística de comunicação entre a aplicação web e cada função).

Esta arquitetura é composta por cinco serviços, que atuam como elementos agregadores de funções *serverless*:

- Serviço de Transporte - responsável pela lógica de transporte da encomenda até ao utilizador e que contém uma função responsável por obter os tipos de transporte disponíveis na localização do utilizador e uma segunda função responsável por confirmar o transporte da encomenda.
- Serviço de Pagamento - responsável pelo logística de pagamento da encomenda. Para gerir esta logística alberga duas funções, a primeira responsável por obter os métodos de pagamento disponíveis na localização do utilizador e a segunda responsável por concretizar o pagamento da encomenda.
- Serviço de Artigos de Arte - responsável pelo gestão de todos os artigos de arte vendidos na loja e que contém funções para a obtenção de todos os artigos de arte existentes na loja e para a criação de um novo artigo.
- Serviço de Artistas - responsável pelo gestão dos autores dos artigos vendidos na loja. Para esta gestão utiliza uma função responsável por obter todos os autores de artigos de arte vendidos na loja e outra função responsável por criar um novo artista.
- Serviço de Encomendas - responsável pela criação de encomendas da loja. Para este registo utiliza uma função responsável por criar as encomendas.

Cada função, dependendo do seu objetivo e do seu contexto, comunica com os serviços de transporte e pagamento externos ou com as bases de dados de artistas e artigos de arte.

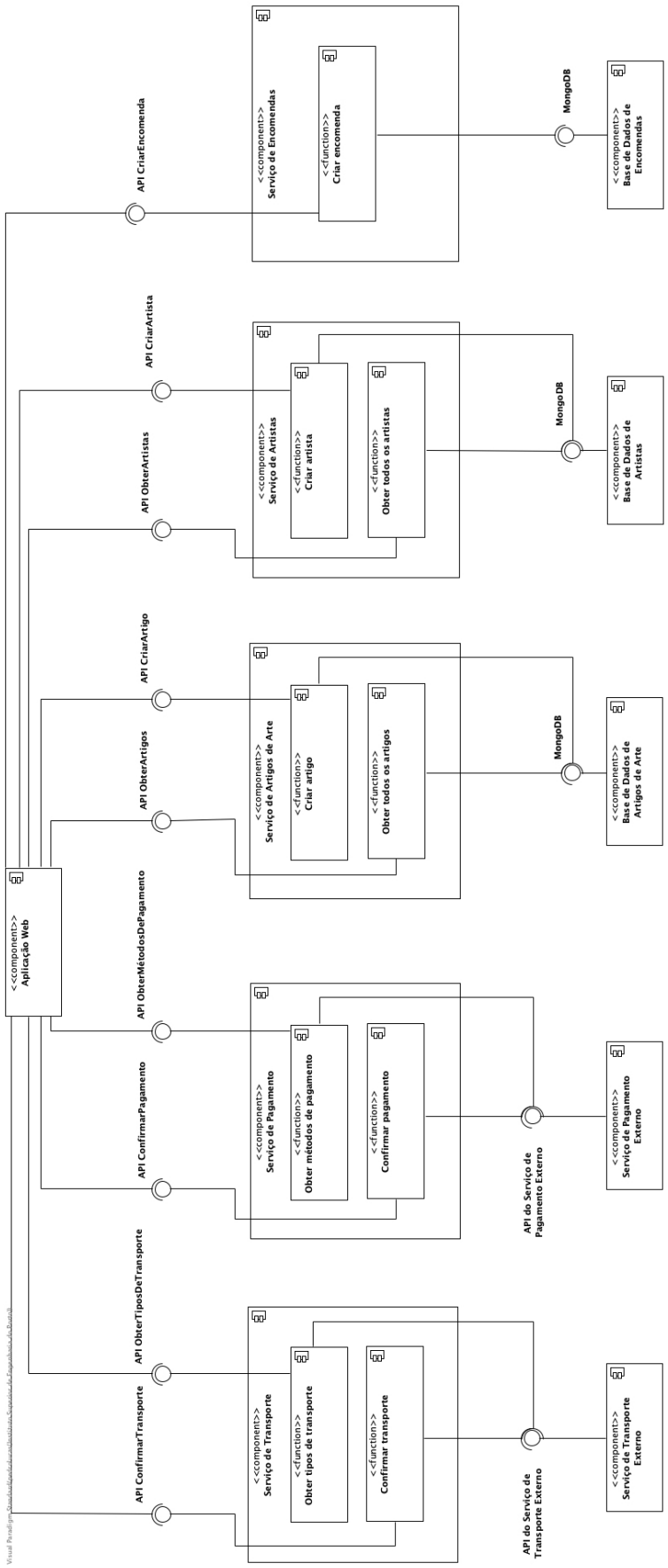

Figura 5.2: Arquitetura da abordagem *serverless*

5.1.3 Comparação das arquiteturas

Através da análise dos dois diagramas anteriores é possível identificar algumas semelhanças entre uma arquitetura orientada a microserviços e uma arquitetura *serverless*: a baixa granuralidade funcional de cada componente promove uma arquitetura com baixo nível de acoplamento e interdependência, alta escalabilidade e uma manutenção simplificada.

Por outro lado, é também possível identificar algumas diferenças entre estas duas abordagens. Enquanto que uma abordagem orientada a microserviços utiliza maioritariamente recursos Representational State Transfer (REST) [64], uma abordagem *serverless* faz uso de funções, sendo por isso importante compreender o que distingue estes dois tipos de componentes.

Um recurso REST é um objeto que possui um tipo, dados associados, relações com outros recursos e um conjunto de métodos que operam com estes dados. Por outro lado uma função *serverless* consiste num conjunto de código com um propósito único cuja configuração e gestão em termos de servidor é invisível para o utilizador final. Estes dois tipos de componentes diferem entre si na forma como são acedidos: os recursos REST são acedidos através de pedidos HTTP enquanto que as funções *serverless* são executadas através do despoletamento de eventos, que variam na sua forma desde pedidos HTTP e *websockets* a regras IoT e transmissões na plataforma Kinesis.

Para além disso, cada função *serverless* resulta do isolamento das responsabilidades já pequenas de cada microserviço em responsabilidades ainda mais pequenas, sendo quase equivalentes a nanoserviços. Ora, esta menor granularidade da responsabilidade de cada função pode aumentar a facilidade e o poder de escalabilidade do sistema como um todo. No entanto, é importante notar que por contrapartida a complexidade de gerir este isolamento também poderá aumentar.

5.2 Visualizar Lista de Artigos

Esta secção apresenta o design do caso de uso em que o utilizador acede a uma listagem de todos os artigos de arte.

5.2.1 Abordagem orientada a microserviços

Na abordagem orientada a microserviços, apresentada na figura 5.3, de forma a apresentar todos os artigos a aplicação web executa uma função, do serviço de artigos de arte, que é responsável por fazer um pedido HTTP ao microserviço de artigos de arte. Este pedido é redirecionado pela API Gateway para o controlador deste microserviço e este controlador recorre ao serviço e ao repositório do microserviço para obter a listagem de todos os artigos, que é depois retornada pelo controlador para a aplicação web.

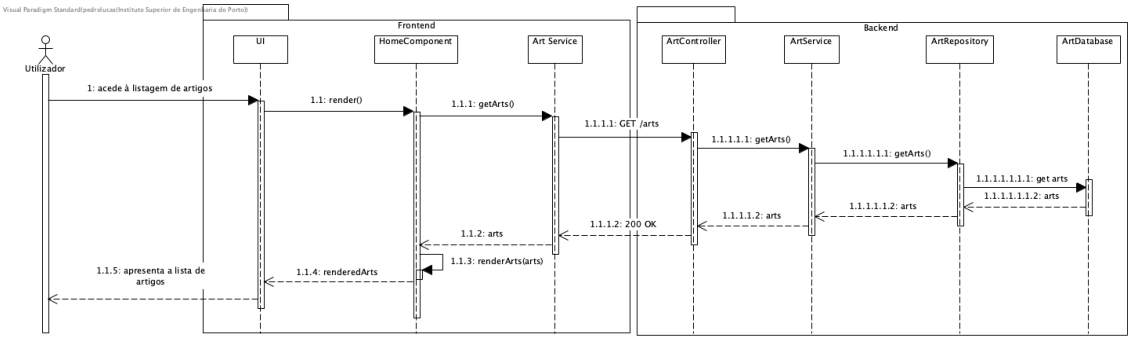

Figura 5.3: Visualizar Lista de Artigos - Abordagem orientada a microserviços

5.2.2 Abordagem serverless

Na abordagem *serverless*, a lógica da parte da aplicação web mantém-se sendo apenas alterada a lógica da componente de *backend*. O pedido HTTP feito pelo serviço de artigos de arte despoleta a execução da função GetArts que por sua vez comunica com a base de dados de forma a obter a listagem de todos os artigos de arte e, posteriormente, devolver esta listagem para a aplicação web.

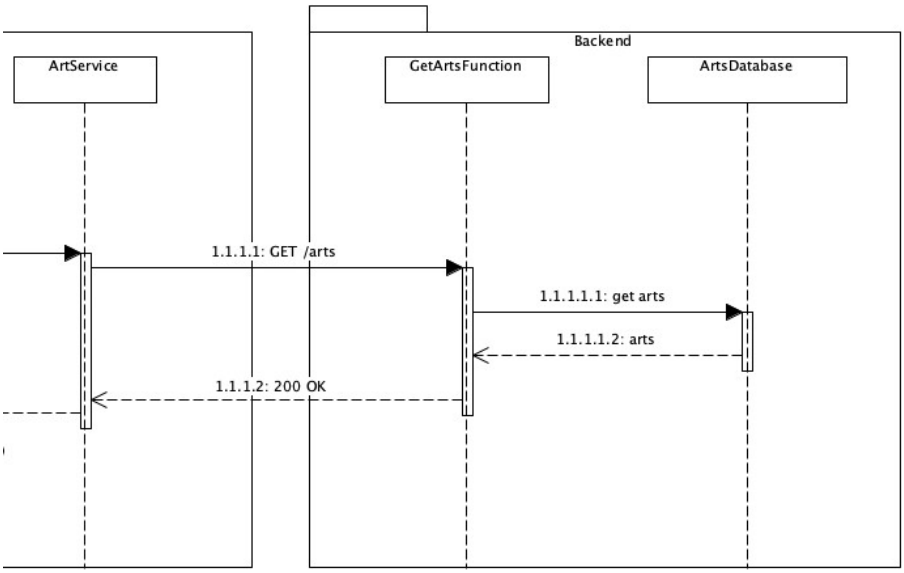

Figura 5.4: Excerto do diagrama de sequência da visualização da listagem de artigos de arte segundo a abordagem *serverless*

5.2.3 Comparação das abordagens

Neste caso de uso a principal diferença entre as duas abordagens reside no acesso à base de dados. Na abordagem orientada a microserviços o controlador delega a tarefa de obtenção esta informação a um método específico do serviço de artigos de arte, que comunica com o repositório e este último acede à listagem de artigos na base de dados. Já na abordagem *serverless* este acesso é realizado pela função *GetArts*.

5.3 Adicionar artigo ao carro de compras

Dado que esta funcionalidade não requer a comunicação com *backend*, o design das duas abordagens é igual pelo que não será feita uma comparação entre as duas para este caso de uso e por este motivo não é relevante para o resultado do caso de estudo, apenas para a lógica de sistema, pelo que o design deste caso de uso é apenas apresentado no apêndice A5.

Relativamente a este caso de uso importa notar que a informação relativa ao carro de compras será armazenada no próprio navegador através da *localStorage*, que consiste numa forma de armazenamento de dados sem expiração.

5.4 Remover artigo do carro de compras

À semelhança do caso de uso anterior, a remoção de artigos do carro de compras não requer comunicação com componentes de *backend*, pelo que, como o design para as duas abordagens é igual, não será feita uma comparação entre as duas no que diz respeito a esta funcionalidade.

Relativamente à funcionalidade, apresentada no apêndice A6, é possível compreender que a sua lógica é bastante parecida à de adição de artigos, comunicando com a *localStorage* para atualizar as quantidades de cada artigo presentes no carro de compras depois de realizada a remoção.

5.5 Fazer encomenda

Nesta secção é analisada a criação de uma encomenda, do ponto de vista do design, tendo em conta as duas abordagens anteriormente mencionadas.

5.5.1 Abordagem orientada a microserviços

Na abordagem orientada a microserviços é possível perceber que para a criação de uma encomenda são realizadas 4 etapas: inicialmente a aplicação web comunica com o microserviço de transporte, de forma a validar a morada do utilizador. Depois de validada a morada do utilizador, a aplicação web comunica novamente com o microserviço de transporte de forma a criar uma simulação de envio e obter todos os serviços de transporte disponíveis na morada validada anteriormente. Quer para realizar a validação da morada quer para obter os serviços de transporte disponíveis o microserviço de transporte comunica com a API do serviço de transporte externo, que fica assim responsável por concretizar estas tarefas.

De seguida, depois de o utilizador selecionar o serviço de transporte desejado é necessário concretizar o pagamento da encomenda. Para isso a aplicação web comunica com o microserviço de pagamento de forma a obter todos os métodos de pagamento disponíveis no país do utilizador. Posteriormente o utilizador concretiza este pagamento através de um dos métodos de pagamento disponíveis e é feita uma comunicação com o microserviço de encomendas de forma a registar a nova encomenda.

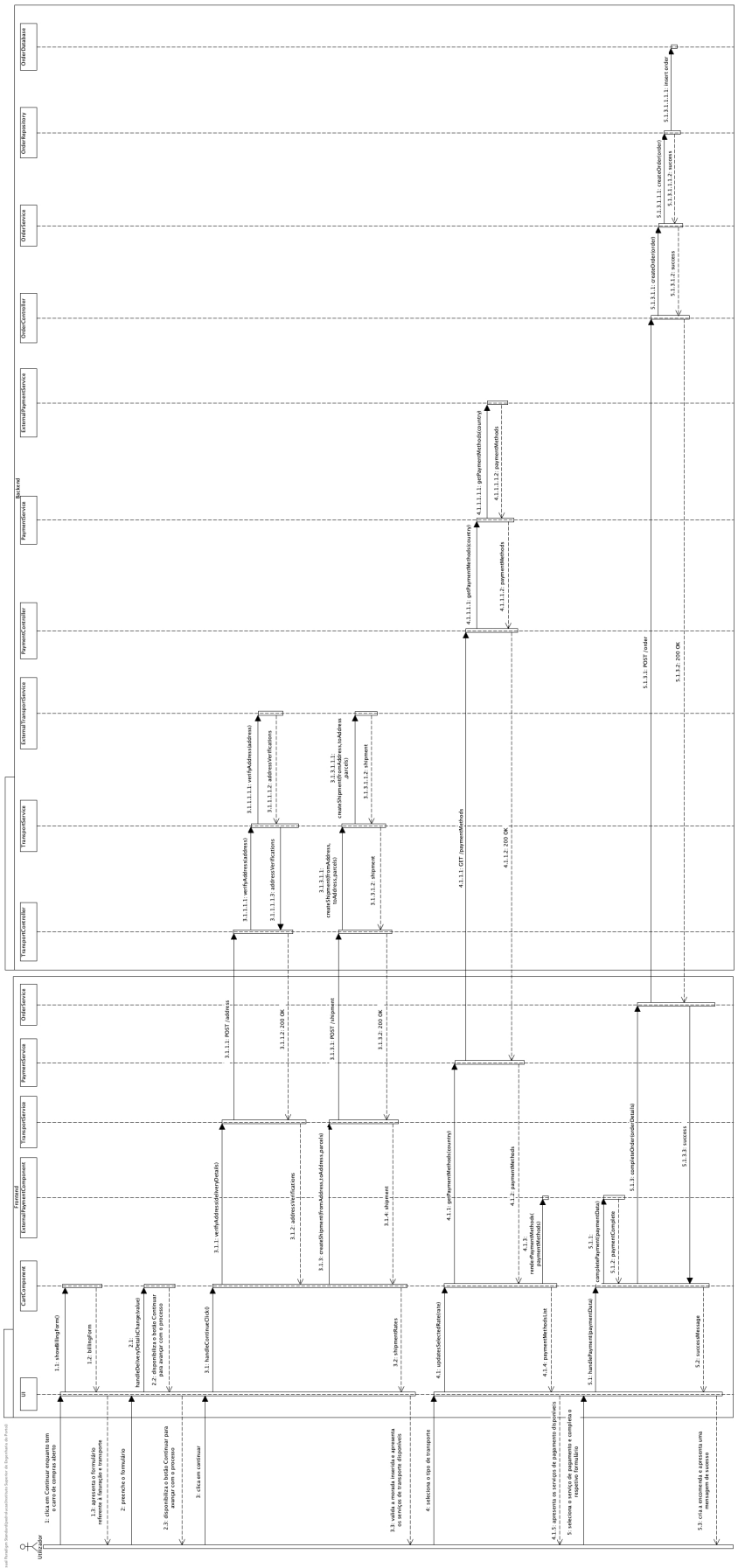

Figura 5.5: Fazer encomenda - Abordagem orientada a microserviços

5.5.2 Abordagem serverless

Na abordagem *serverless* é possível perceber que para a criação de uma encomenda também são realizadas 4 etapas: inicialmente a aplicação web comunica com a função ValidateAddress, de forma a validar a morada do utilizador. Esta função por sua vez comunica com o serviço de transporte externo de forma a concretizar a validação da morada do utilizador. Depois de validada a morada, a aplicação web despoleta a execução da função CreateShipment, de forma a criar uma simulação de envio e obter todos os serviços de transporte disponíveis na morada validada anteriormente. Para obter os serviços de transporte disponíveis esta função comunica com a API do serviço de transporte externo.

De seguida, depois de o utilizador selecionar o serviço de transporte desejado é necessário concretizar o pagamento da encomenda. Para isso a aplicação web comunica com a função GetPaymentMethods, que tem como principal finalidade a obtenção de todos os métodos de pagamento disponíveis no país do utilizador. Posteriormente o utilizador concretiza este pagamento através de um dos métodos de pagamento disponíveis e é feita uma comunicação com a função CreateOrder de forma a registar a nova encomenda.

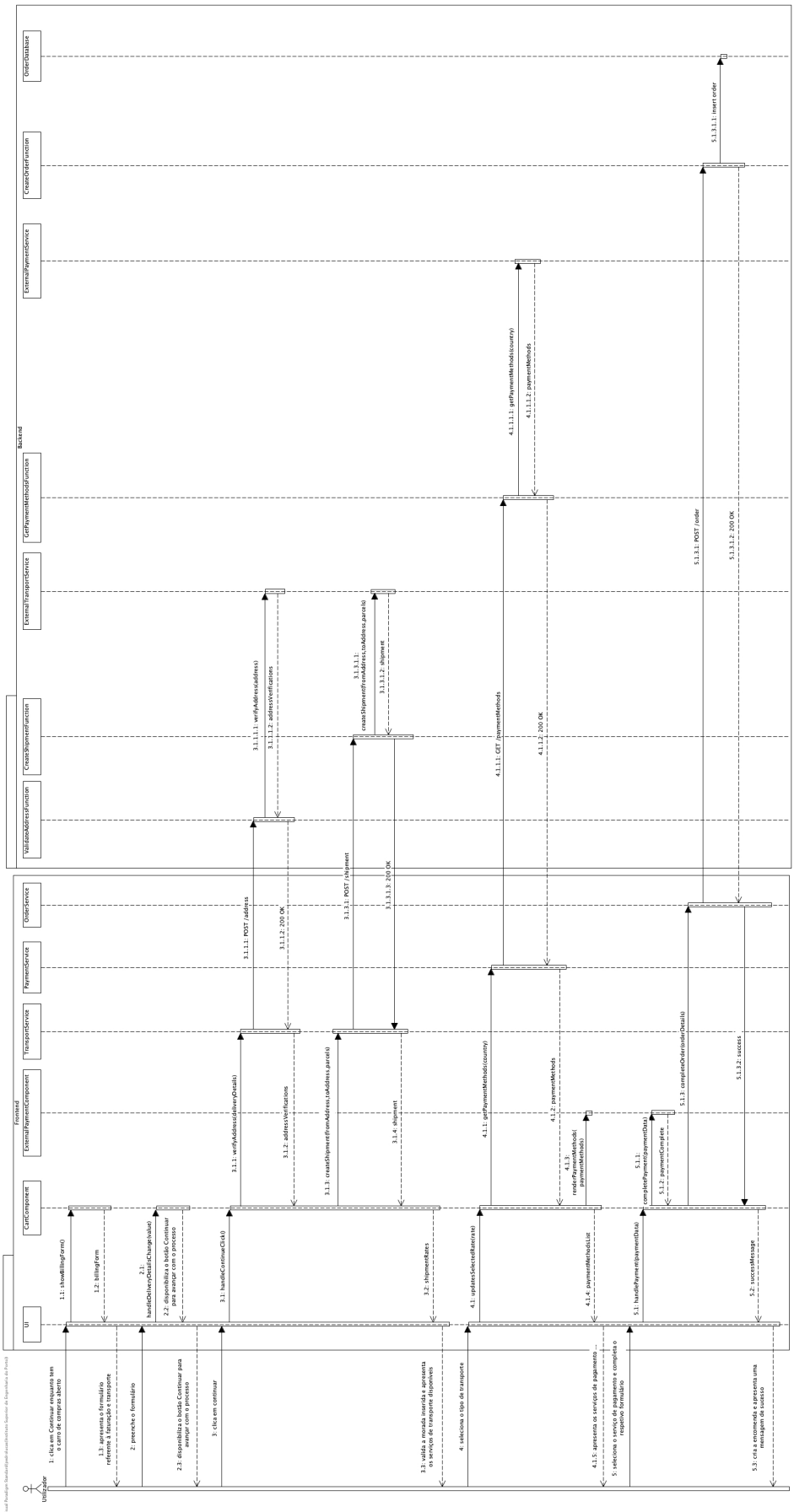

Figura 5.6: Fazer encomenda - Abordagem *serverless*

5.5.3 Comparação das abordagens

Analisando os dois diagramas é possível perceber, mais uma vez, que uma das principais diferenças entre as duas abordagens é a granularidade de cada recurso envolvido na lógica de *backend*. Na abordagem *serverless* o isolamento das responsabilidades já pequenas dos quatro serviços utilizados na abordagem orientada a microserviços em responsabilidades ainda mais pequenas leva a uma menor granularidade da responsabilidade de cada função *serverless*, o que pode levar a um aumento da escalabilidade do sistema nesta abordagem.

5.6 Acompanhar encomenda

Nesta secção é apresentado o design relativo caso de uso de acompanhamento da encomenda por parte do utilizador com base numa abordagem orientada a microserviços e numa abordagem *serverless*.

5.6.1 Abordagem orientada a microserviços

Nesta abordagem, apresentada na figura 5.7, para obter a informação relativa ao acompanhamento da encomenda a aplicação web comunica com o serviço de transporte de forma a gerar uma etiqueta de transporte, que contém a informação necessária para o transporte da encomenda desde o armazém até à morada final. Nesta informação está contido também um identificador do rastreador da encomenda. A partir deste identificador é feito mais um pedido ao serviço de transporte que retorna toda a informação do rastreador, onde se inclui um *url* do serviço de transporte externo que permite ao utilizador acompanhar o estado da sua encomenda.

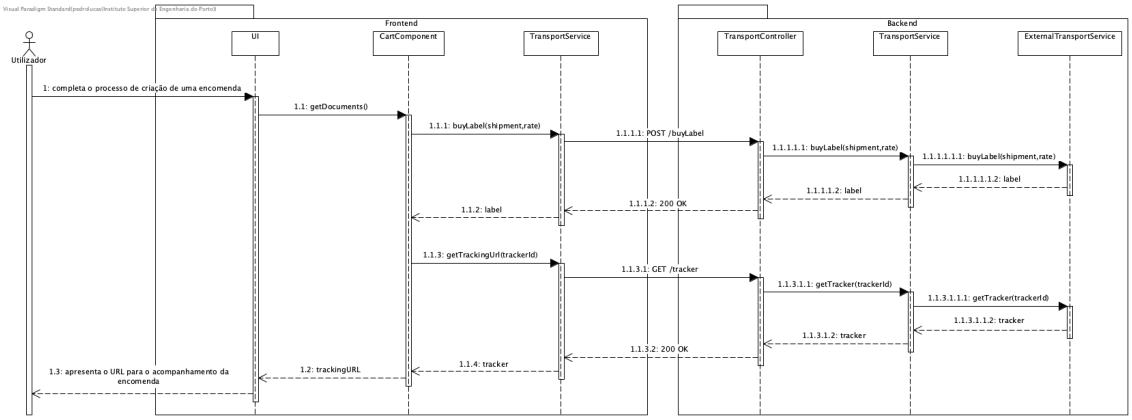

Figura 5.7: Acompanhar encomenda - Abordagem orientada a microserviços

5.6.2 Abordagem serverless

No que diz respeito à abordagem *serverless*, apresentada na figura 5.8, o processo é dividido em duas etapas: inicialmente a aplicação web despoleta a execução da função BuyLabel de forma a gerar a etiqueta de transporte, comunicando para isso com o serviço de transporte externo. Depois de gerada esta etiqueta é executada a função RetrieveTracker que recebe como parâmetro o identificador do rastreador da encomenda obtido na execução da função anterior e comunica com o serviço de transporte externo de forma a obter toda a informação

sobre o rastreador. Nesta informação obtida está incluído um *url* gerado pelo serviço de transporte externo que permite ao utilizador acompanhar o estado da sua encomenda.

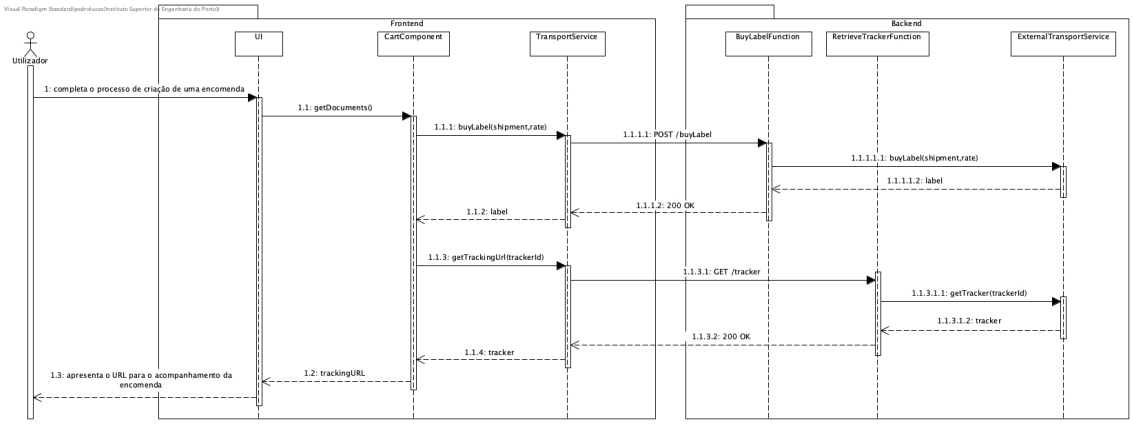

Figura 5.8: Acompanhar encomenda - Abordagem *serverless*

5.6.3 Comparação das abordagens

À semelhança do caso de uso anterior, uma das principais diferenças entre os dois exemplos de arquitetura apresentados é a granularidade de cada recurso envolvido na lógica de *backend*. Na abordagem orientada a microserviços toda a lógica de *backend* é gerida por um microserviço enquanto que na abordagem *serverless* é repartida por duas funções.

Capítulo 6

Implementação

Este capítulo documenta a implementação do caso de estudo previamente definido, uma loja *e-Commerce* de artigos de arte, para as duas abordagens desenhadas anteriormente. Inicialmente é apresentado o conjunto de tecnologias utilizado para a implementação de cada abordagem, seguido de uma introdução à implementação de funções *serverless* e por fim explanada a implementação de cada caso de uso.

6.1 Stack tecnológica

Nesta secção é apresentado o conjunto de tecnologias selecionado para implementar a loja *e-Commerce* de artigos de arte segundo cada abordagem.

6.1.1 Abordagem orientada a microserviços

Para a implementação de cada microserviço foi utilizada a *framework* Express, uma biblioteca Node.js orientada ao desenvolvimento de aplicações web e mobile e que oferece um conjunto de métodos utilitários que facilita o desenvolvimento de APIs. Esta escolha deveu-se ao facto de ser uma *framework* conceituada no que ao desenvolvimento deste tipo de componentes diz respeito e pelo grau de conhecimento que o autor da dissertação possuía, à partida, da mesma, diminuindo assim a curva de aprendizagem em relação a outras tecnologias. A base de dados, como apresentado anteriormente, é uma MongoDB, uma base de dados não relacional orientada a documentos e que foi escolhida pela integração simples com a framework *Express* que oferece. Por fim, a aplicação web foi desenvolvida em React, uma *framework* de JavaScript. A lista de todas as tecnologias utilizadas na implementação desta abordagem encontra-se apresentada na tabela 6.1.

Tabela 6.1: Tecnologias utilizadas na implementação do caso de estudo

Tecnologia	Descrição
Javascript	Linguagem de programação interpretada, normalmente utilizada no desenvolvimento de aplicações de *frontend* e *backend*
Node.js	Ambiente de execução JavaScript orientado a eventos projetado para o desenvolvimento de aplicações escaláveis
Express	*Framework* Node.js orientada ao desenvolvimento de aplicações web e mobile
MongoDB	Base de dados não relacional orientada a documentos
React	Biblioteca de JavaScript utilizada para construir interfaces baseadas em componentes

6.1.2 Abordagem serverless

A abordagem *serverless* foi criada a partir da ferramenta Serverless Framework, apresentada na secção 2.2.2, que permite a criação de múltiplos componentes *serverless*. Nestes componentes incluem-se, para este caso de estudo, as funções *serverless* e a aplicação web, sendo que as funções foram desenvolvidas em Node.js. A base de dados é uma MongoDB e a aplicação web foi desenvolvida em React, tal como na abordagem orientada a microserviços. Na tabela 6.1 são apresentadas todas as tecnologias utilizadas na implementação desta abordagem.

Tabela 6.2: Tecnologias utilizadas na implementação do caso de estudo

Tecnologia	Descrição
Javascript	Linguagem de programação interpretada, normalmente utilizada no desenvolvimento de aplicações de *frontend* e *backend*
Node.js	Ambiente de execução JavaScript orientado a eventos projetado para o desenvolvimento de aplicações escaláveis
MongoDB	Base de dados não relacional orientada a documentos
React	Biblioteca de JavaScript utilizada para construir interfaces baseadas em componentes

6.2 Serviços serverless

No desenvolvimento *serverless* a cada serviço está associado um conjunto de funções, de eventos que despoletam estas funções e de recursos que cada função necessita para desempenhar o comportamento esperado. Esta informação é definida num ficheiro chamado "serverless.yaml".

No excerto de código 6.1 é apresentado, a título de exemplo, o conteúdo do ficheiro serverless.yaml do serviço de artigos de arte.

```yaml
app: api

service: art

provider:
  name: aws
  stage: dev
  region: us-east-1
  iamRoleStatements:
    - Effect: Allow
      Action:
        - dynamodb:Query
        - dynamodb:Scan
        - dynamodb:GetItem
        - dynamodb:PutItem
      Resource: '*'

functions:
  getArts:
    handler: service.getAll
    events:
      - http: GET /art
  createArt:
    handler: service.create
    events:
      - http: POST /art
  getArtById:
    handler: service.getArtById
    events:
      - http: GET /art/{id}

resources:
  Resources:
    Art:
      Type: AWS::DynamoDB::Table
      Properties:
        TableName: Art
        AttributeDefinitions:
          - AttributeName: id
            AttributeType: S
          - AttributeName: image
            AttributeType: S
          - AttributeName: artist
            AttributeType: S
          - AttributeName: price
            AttributeType: N
        KeySchema:
          - AttributeName: id
            KeyType: HASH
        ProvisionedThroughput:
          ReadCapacityUnits: 1
          WriteCapacityUnits: 1
```

Código 6.1: Ficheiro serverless.yaml do serviço de artigos de arte

Ora, através da análise do conteúdo deste ficheiro, é possível perceber que existem cinco atributos com maior destaque no mesmo:

- *app* - o nome da aplicação que engloba todos os serviços.

- *service* - o nome do serviço .
- *provider* - os detalhes do fornecedor dos recursos, incluindo o nome, tipo de ambiente (desenvolvimento, teste ou produção), a região e o tipo de ações que o serviço pode executar nos recursos.
- *functions* - a lista de funções que o serviço contém, sendo definido para cada função o código que é executado quando esta é despoletada e o evento que a despoleta, neste caso pedidos HTTP.
- *resources* - o conjunto de recursos utilizado pelas funções no serviço, neste exemplo uma tabela da base de dados não relacional.

Desta forma, através deste ficheiro de configuração são definidas as funções contidas neste serviço bem como os eventos que as despoletam e, em caso disso, as bases de dados e tabelas utilizadas, sendo definido para cada tabela o seu nome, os atributos, as chaves primárias e a capacidade de leitura e escrita.

Depois de configurado este ficheiro resta apenas implementar as funções mencionadas no ficheiro.

6.3 Casos de uso

6.3.1 Visualizar lista de artigos

A visualização da lista de artigos de arte é o primeiro ponto de encontro do utilizador com a aplicação e é a partir desta visualização que todo o processo de negócio se desenrola. De seguida é a implementação deste caso de uso em cada abordagem.

Abordagem orientada a microserviços

Na abordagem orientada a microserviços, quando o utilizador acede à página inicial da aplicação web, é feito um pedido HTTP que é redirecionado pela API Gateway para o microserviço de artigos de arte. O controlador deste microserviço recebe este pedido e executa a função getAll da camada de serviço da aplicação, que por sua vez comunica com o repositório de forma a aceder à informação necessária. No excerto de código 6.1 é apresentada a função do controlador onde são devolvidos todos os artigos de arte e no excerto 6.2 a função do repositório executada pela camada de serviço de forma a atingir este fim.

```
controller.get('/', async function (req, res) {
  try {
    const arts = await service.getAll();
    return res.status(200).send({ arts });
  } catch (error) {
    return res.status(500).json({ message: error.message });
  }
});
```

Código 6.2: Função do controlador do microserviço de artigos de arte responsável por devolver todos os artigos de arte

```
getAll = function () {
  return ArtDB.find({});
};
```

Código 6.3: Função do repositório do microserviço de artigos de arte responsável por devolver todos os artigos de arte

Abordagem serverless

No que diz respeito à abordagem *serverless*, assim que o utilizador abre a aplicação, é feito um pedido GET ao *endpoint* /art pela aplicação web, pedido este que funciona como evento despoletador da execução da função getAll. Esta função comunica com a base de dados de artigos de arte e obtém o conjunto de todos os artigos, retornando para a aplicação web esta informação. No excerto de código 6.4 é apresentado o conteúdo da função *serverless* getAll.

```
getAll = async (event, context) => {
  const arts = ArtDB.find({});

  return {
    statusCode: 200,
    body: JSON.stringify({
      arts
    })
  };
};
};
```

Código 6.4: Função *serverless* getAll, utilizada para obter o conjunto de todos os artigos

6.3.2 Adicionar e remover artigos do carro de compras

Como mencionado na secção 5.3 e 5.4, a gestão da quantidade de artigos no carro compras é feita pela própria aplicação web, através do armazenamento do próprio navegador, pelo que para estes casos de uso a implementação da abordagem orientada a microserviços e da abordagem *serverless* é igual. Importa então perceber os tipos de armazenamento que os navegadores oferecem e qual o utilizado neste caso de estudo.

A primeira forma de armazenamento de informação no navegador foram as *cookies*. As *cookies* são normalmente utilizadas para fins de autenticação e persistência de informação do utilizador e, por este motivo, são enviadas do navegador para o servidor em cada pedido HTTP feito no domínio da aplicação. Devido ao tipo de informação que contêm possuem um limite de tamanho de 4kb e podem ser protegidas através da atribuição do valor *true* à propriedade *httpOnly* nos diversos pedidos.

Mais recentemente a *localStorage* e *sessionStorage*. As duas permitem o armazenamento de pares chave-valor no navegador e a principal diferença destas em relação às *cookies* é o limite de tamanho da informação armazenada, que varia entre 5MB e 10MB, o facto de não serem enviadas em cada pedido feito no domínio da aplicação e o de não possuirem, por defeito, qualquer protocolo de segurança. A *sessionStorage* e *localStorage* diferem entre si no período de expiração dos seus dados. Na *sessionStorage* os dados apenas são persistidos

enquanto a sessão estiver ativa, isto é, os dados são eliminados assim que a aba ou o navegador sejam fechados, enquanto que na *localStorage* os dados são persistidos até que sejam eliminados explicitamente.

Assim, e dado que a informação do carro de compras deve ser persistida até que o utilizador a deseje, de facto, apagar e que se espera que possa ter um tamanho considerável, optou-se por usar a *localStorage*. De forma a armazenar os dados na *localStorage* é usada a função *setItem* que recebe como parâmetro o par chave-valor a ser armazenado. De notar que a *localStorage* só armazena texto pelo que quando é necessário armazenar um objeto, este tem de ser convertido em texto. No excerto de código 6.5 é apresentada a função responsável pela atualização dos artigos armazenados na *localStorage* e, na figura 6.1, apresentado um exemplo da *localStorage* depois de serem armazenados artigos na mesma.

```
1  updateCartItems(cartItems) {
2    localStorage.setItem('cartItems', JSON.stringify(cartItems));
3  }
```

Código 6.5: Atualização dos artigos do carro de compras na *localStorage*

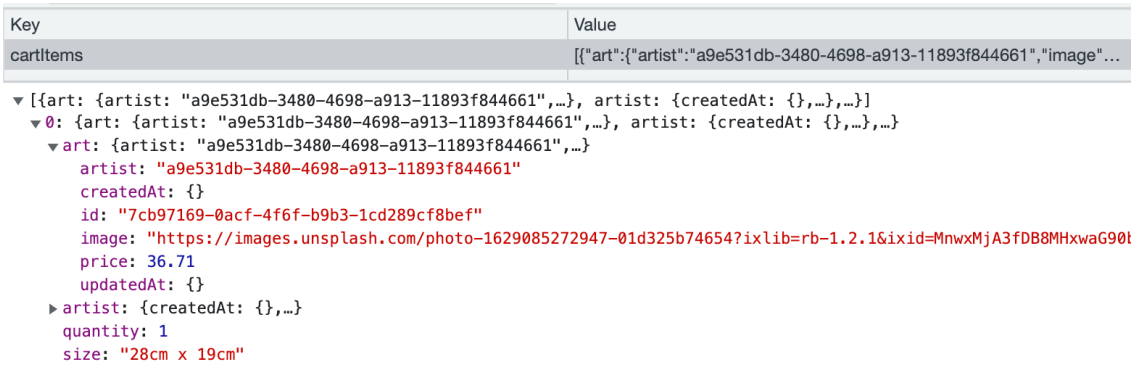

Figura 6.1: Artigos do carro de compras depois de armazenados na *localStorage*

6.3.3 Fazer encomenda

O processo de criação de uma encomenda, depois de serem adicionados artigos ao carro de compras, é composto por 4 etapas principais. São elas:

1. Validar morada do utilizador
2. Obter tipos de serviço de transporte disponíveis para a morada do utilizador
3. Obter métodos de pagamento
4. Criar encomenda

As duas primeiras etapas estão associadas a um serviço de transporte externo. Para este serviço fez-se um levantamento de opções disponíveis no mercado. Depois de alguma análise chegou-se à conclusão de que alguns dos serviços mais utilizados neste âmbito, como é o caso da Shippo e da UPS, requerem que seja criada uma conta de marca para a utilização da sua API, o que implica a obrigatoriedade de fazer no mínimo uma encomenda não fictícia de dois em dois em meses.

Ora, esta obrigatoriedade impossibilitou o uso destes serviços e por esse motivo optou-se por utilizar como serviço de transporte externo a EasyPost, que possui uma integração com o serviço postal dos Estados Unidos e, por esse motivo, a sua integração acaba por ser útil como caso de estudo.

Já a terceira etapa está associada a serviços de pagamento externos. Com o crescimento de mercado que o *e-Commerce* tem tido nos últimos anos, foram lançadas várias APIs de serviços de pagamento de forma a conseguir acompanhar este crescimento. Na tabela 6.3 é apresentado um resumo das características dos serviços mais conceituados.

Tabela 6.3: Serviços de pagamento. Fonte: [65]

Nome	**Custo**	**Cartões de Crédito**	**Pagamentos bancários**	**Criptomoeda**
Stripe	2.9% + 0.30$	Sim	Sim	Não
Noodlio	3.4% + 0.40$	Sim	Não	Não
Square	2.9% + 0.30$	Sim	Não	Não
Paypal	2.9% + 0.30$	Sim	Sim	Não
Adyen	2.9% + 0.30$	Sim	Sim	Sim
PayMill	2.95% + 0.28€	Sim	Não	Não

Depois de feito este levantamento, decidiu-se integrar a aplicação com 3 dos serviços: Stripe, por ser o mais conceituado e mais utilizado, Paypal, por ser uma carteira digital que tem ganho cada vez mais clientes, e finalmente Adyen, por suportar criptomoeda e por isso ser mais um exemplo de integração com um tipo de pagamento distinto.

Validar morada do utilizador

A primeira parte do processo de criação de uma encomenda é a inserção da informação de transporte por parte do utilizador e a validação desta mesma informação. Nesta secção é apresentada a forma como esta validação é feita de acordo com as duas abordagens anteriormente mencionadas.

Na abordagem orientada a microserviços, a aplicação web faz um pedido HTTP que é redirecionado pela API Gateway para o microserviço de transporte. O controlador deste microserviço recebe então este pedido, através da função apresentada no excerto de código 6.6, e solicita à camada de serviço que valide a morada do utilizador. Para concretizar essa validação, este serviço comunica com a API da EasyPost que tem a responsabilidade de validar a morada, tal como apresentado no excerto de código 6.7.

```
controller.post('/validateAddress', async function (req, res) {
  try {
    const data = req.body;
    const address = await service.validateAddress(data);
    return res.status(200).send({ address });
  } catch (error) {
    return res.status(500).json({ message: error.message });
  }
});
```

Código 6.6: Função do controlador do microserviço de transporte responsável por validar a morada do utilizador

```
validateAddress = async function (data) {
  const EasyPost = require('@easypost/api');
  const EasyPostAPI = new EasyPost(EASYPOST_TOKEN);
  return await new EasyPostAPI.Address(data).save();
};
```

Código 6.7: Código da função responsável por validar a morada do utilizador

Já na abordagem *serverless*, de forma a realizar esta validação foi criada uma função *serverless* que tem também como principal objetivo comunicar com o serviço da EasyPost e validar a morada do utilizador.

Assim, a aplicação web faz um pedido HTTP que funciona como evento despoletador da função de validação da morada. Esta função recebe então como parâmetros os detalhes da morada do utilizador, onde se inclui a rua, cidade, código postal, país, entre outros, e envia estes parâmetros para o serviço da EasyPost, simulando a criação de uma morada. A lógica desta função pode ser analisada no excerto de código 6.8.

```
validateAddress = async (event, context, callback) => {
  try {
    const data = JSON.parse(event.body);
    const EasyPost = require('@easypost/api');
    const EasyPostAPI = new EasyPost(EASYPOST_TOKEN);
    const address = await new EasyPostAPI.Address(data).save();

    return {
      statusCode: 200,
      body: JSON.stringify({
        address
      })
    };
  } catch (error) {
    return {
      statusCode: 500,
      body: JSON.stringify({
        message: error.message
      })
    };
  }
};
```

Código 6.8: Código da função *serverless* validateAddress

Desta criação de morada pode então ser retornada uma mensagem de sucesso ou de erro, como por exemplo no caso de a morada não ser encontrada. Na figura 6.2 é apresentado um exemplo deste caso de erro na validação de uma morada.

Figura 6.2: Exemplo de erro retornado na validação de morada

Obter tipos de serviço de transporte disponíveis para a morada do utilizador

Depois de validada a morada é necessário obter os tipos de serviço de transporte que os parceiros da EasyPost possuem para essa mesma morada. Para isso, é criado, através da API da EasyPost um envio fictício para a morada, sendo também definida a morada de partida, que para este caso de estudo é também fictícia, e os pacotes a enviar, sendo definida para estes a sua altura, largura e comprimento.

A aplicação web faz então um pedido HTTP que no caso da abordagem orientada a microserviços é redirecionado para o microserviço de transporte. Este pedido é recebido pelo controlador que executa uma função do serviço de a obter os métodos de transporte disponíveis. Para isso, a camada de serviço comunica com a API da EasyPost que devolve os métodos de transporte disponíveis para a morada do utilizador. Esta função é apresentada no excerto de código 6.9.

```
createShipment = async function (data) {
  const EasyPost = require('@easypost/api');
  const EasyPostAPI = new EasyPost(EASYPOST_TOKEN);
  return await new EasyPostAPI.Shipment(data).save();
};
```

Código 6.9: Código da função createShipment, responsável por obter os tipos de serviço de transporte disponíveis

Por outro lado, na abordagem *serverless*, este pedido HTTP despoleta a execução da função createShipment, apresentada no excerto de código 6.10, que faz então um pedido à API da EasyPost de forma a obter os métodos de transporte disponíveis.

```
createShipment = async (event, context, callback) => {
  try {
    const data = JSON.parse(event.body);
    const shipment = await new EasyPostAPI.Shipment(data).save();

    return {
      statusCode: 200,
      body: JSON.stringify({
        shipment,
      }),
    };
  }
};
```

Código 6.10: Código da função *serverless* createShipment

Da execução destas funções resulta uma lista de tipos de serviço de transporte disponíveis, sendo definido cada tipo de serviço o seu nome, transportador, preço e data estimada de entrega.

Obter métodos de pagamento

Numa fase posterior, é realizado o pagamento da encomenda. Quase toda a integração dos três serviços de pagamento é feita na aplicação *frontend* pois todos estes serviços de pagamento já possuem componentes próprios para serem integrados neste tipo de aplicações, no entanto há duas tarefas que é necessário realizar para que este processo aconteça: para inicializar o componente do serviço Adyen é necessário fazer um pedido à sua API para obter os métodos de pagamento disponíveis com base no total a pagar e no país do utilizador e o serviço Stripe requere que depois de completado o pagamento seja feito um pedido à sua API com os detalhes do mesmo.

Para a primeira tarefa, na abordagem orientada a microserviços, a lógica é semelhante ao realizado até aqui. A aplicação web faz um pedido HTTP que é reencaminhado pela API Gateway para o microserviço de pagamento. Neste microserviço, o controlador recebe o pedido e executa a função getAdyenPaymentMethods, da camada de serviço. Esta função, apresentada no excerto de código 6.11, recebe como parâmetros o país do utilizador, o total a pagar e o código da língua desse país, comunica com a API do serviço Adyen e obtém os métodos de pagamento desta ferramenta.

```
getAdyenPaymentMethods = async function (country, locale, amount) {
  if (!country || !locale || !amount) {
    throw Error("Couldn't retrieve adyen payment methods because of
    validation errors.");
  }

  const { Client, CheckoutAPI } = require('@adyen/api-library');
  const client = new Client(getAdyenConfig());
  client.setEnvironment(ADYEN_ENVIRONMENT);
  const checkout = new CheckoutAPI(client);

  return await checkout.paymentMethods({
    merchantAccount: config.merchantAccount,
    countryCode: country,
    shopperLocale: locale,
    amount: { currency: 'EUR', value: amount },
    channel: 'Web',
  });
};
```

Código 6.11: Código da função getAdyenPaymentMethods

Para a abordagem *serverless*, foi também criada a função getAdyenPaymentMethods, que valida a informação que recebe como parâmetros e transmite-a à API do serviço Adyen de forma a obter então os métodos de pagamento disponíveis no país do utilizador. Esta função encontra-se apresentada no excerto de código 6.12.

```
getAdyenPaymentMethods = async (event, context, callback) => {
  try {
    const { country, locale, amount } = JSON.parse(event.body);

    if (!country || !locale || !amount) {
      return {
        statusCode: 500,
        body: JSON.stringify({
          message: "It's not possible to retrieve adyen payment methods
    without an bill amount, locale or country.",
        }),
      };
    }

    const { Client, CheckoutAPI } = require('@adyen/api-library');
    const client = new Client(getAdyenConfig());
    client.setEnvironment(ADYEN_ENVIRONMENT);
    const checkout = new CheckoutAPI(client);

    const paymentsResponse = await checkout.paymentMethods({
      merchantAccount: config.merchantAccount,
      countryCode: country,
      shopperLocale: locale,
      amount: { currency: 'EUR', value: amount },
      channel: 'Web',
    });
```

```
30      return {
31        statusCode: 200,
32        body: JSON.stringify({
33          payments: paymentsResponse,
34        }),
35      };
36    } catch (error) {
37      return {
38        statusCode: 500,
39        body: JSON.stringify({
40          message: error.message,
41        }),
42      };
43    }
44  };
```

Código 6.12: Código da função *serverless* getAdyenPaymentMethods

Na figura 6.3 é apresentado um exemplo da informação retornada por cada uma destas funções.

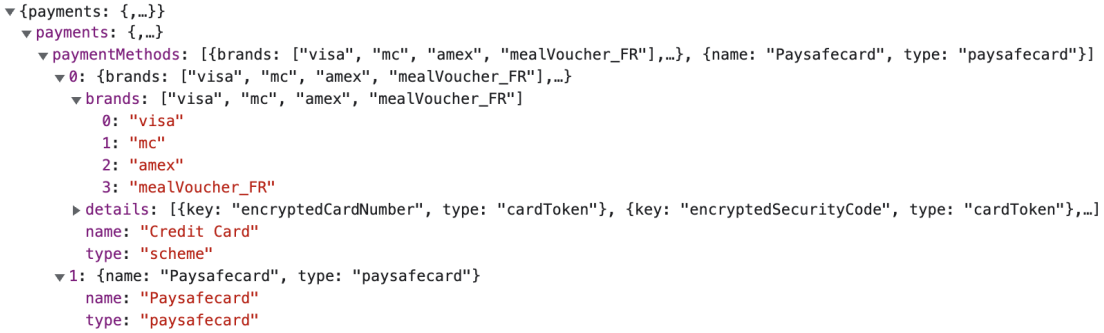

Figura 6.3: Exemplo da informação retornada pelas funções getAdyenPaymentMethods

Para a segunda tarefa foi também necessário implementar a lógica de criação da tentativa de pagamento no serviço Stripe. O código base para criar esta tentativa de pagamento neste serviço, apresentado no excerto 6.13, é igual para as duas plataformas. É recebida a quantidade a pagar por parte do utilizador e é enviado este valor para a API do serviço Stripe, que cria então a tentativa de pagamento e valida-o, sendo depois retornado o resultado desta validação para a aplicação web.

```
1  const stripe = require('stripe')(STRIPE_TOKEN);
2  const paymentIntent = await stripe.paymentIntents.create({
3      amount,
4      currency: 'eur',
5  });
```

Código 6.13: Criação de uma tentativa de pagamento no serviço Stripe

Criar encomenda

Por fim, depois de concretizado o pagamento, é criada a encomenda na base de dados. Na abordagem orientada a microserviços o pedido para esta criação é recebido pelo controlador

do microserviço de encomendas que de seguida executa a função createOrder da camada de serviço. Esta função recebe como parâmetros o nome do utilizador, os detalhes da morada e os artigos encomendados, parâmetros esses que depois de validados são enviados para o repositório deste microserviço de forma a criar a encomenda. O código desta função é apresentado no excerto 6.13.

```
createOrder = async function (order) {
  if (
    typeof order.name !== 'string' ||
    typeof order.address !== 'string' ||
    typeof order.postalCode !== 'string' ||
    typeof order.country !== 'string'
  ) {
    throw Error("Couldn't create the order because of validation errors.
    ");
  }

  return await orderRepository.createOrder(order);
};
```

Código 6.14: Função responsável pela logística da criação da encomenda

Para a abordagem *serverless*, esta criação é também feita por uma função denominada createOrder, que recebe então como parâmetros o nome do utilizador, os detalhes da sua morada e os artigos de arte encomendados, validando-os e inserindo esta informação na base de dados. Esta lógica é apresentada no excerto de código 6.14.

```
createOrder = async (event, context, callback) => {
  const requestBody = JSON.parse(event.body);
  const { name, address, postalCode, country, items } = requestBody;

  if (
    typeof name !== 'string' ||
    typeof address !== 'string' ||
    typeof postalCode !== 'string' ||
    typeof country !== 'string'
  ) {
    return {
      statusCode: 500,
      body: JSON.stringify({
        message: "Couldn't create the order because of validation errors
    ."
      })
    };
  }

  const order = {
    name,
    address,
    postalCode,
    country,
    items
  };

  try {
    const result = await OrderDB.create(order);
```

```
29
30      return {
31        statusCode: 200,
32        body: JSON.stringify({
33          result
34        })
35      };
36    } catch (error) {
37      return {
38        statusCode: 500,
39        body: JSON.stringify({
40          message: error.message
41        })
42      };
43    }
44  };
```

Código 6.15: Função responsável pela logística da criação da encomenda na abordagem *serverless*

6.3.4 Acompanhar encomenda

Depois de criada a encomenda é necessário que o utilizador consiga acompanhar a mesma e que tenha visibilidade sobre o seu estado. Esta capacidade é garantida pelo serviço da EasyPost num processo que envolve duas etapas.

A primeira consiste na criação de uma etiqueta de transporte, que contém toda a informação necessária para que o transportador consiga concretizar a encomenda, incluindo um identificador do rastreador da encomenda.

A lógica das funções responsáveis por esta tarefa nas duas abordagens é semelhante. Recebem como parâmetros a simulação de envio criada anteriormente e o tipo de serviço de transporte selecionado. Posteriormente, concretizam a compra do serviço de transporte e retornam para a aplicação web uma atualização deste envio, que inclui então o identificador do rastreador da encomenda. Esta lógica encontra-se apresentada no excerto de código 6.15 e na figura 6.4 é apresentado um exemplo de uma etiqueta gerada por estas funções.

```
1    const shipment = await EasyPostAPI.Shipment.retrieve(shipmentId);
2    const rate = shipment.rates.filter((r) => r.id === rateId)[0];
3    const { label } = await shipment.buy(rate, 0);
4    const udpatedShipment = await EasyPostAPI.Shipment.retrieve(
    shipmentId);
```

Código 6.16: Código da criação de uma etiqueta de transporte

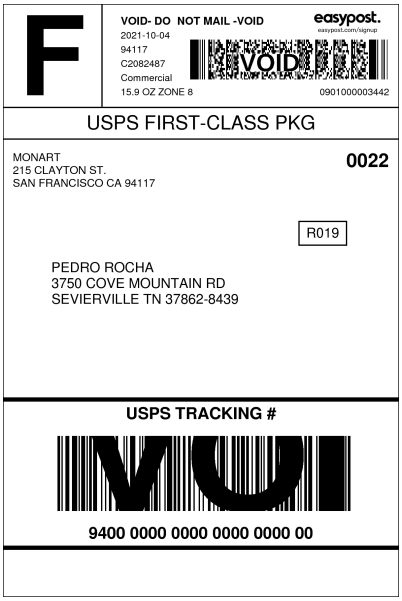

Figura 6.4: Exemplo de uma etiqueta de transporte gerada

A segunda etapa consiste em recolher toda a informação disponível sobre o identificador do rastreador obtido anteriormente, onde se inclui um *link* que permite ao utilizador acompanhar o estado da sua encomenda. Esta recolha é feita através da API da EasyPost, sendo enviado como parâmetro o identificador do rastreador. O código desta comunicação com a API da EasyPost é apresentado no excerto 6.15.

```
1    const tracker = await EasyPostAPI.Tracker.retrieve(trackerId);
```

Código 6.17: Código da obtenção da informação do rastreador através do identificador do mesmo

Capítulo 7

Avaliação

Neste capítulo são apresentadas as hipóteses em avaliação, a metodologia seguida, os resultados obtidos e é feita uma análise dos resultados obtidos.

7.1 Hipóteses

Um teste de hipóteses é um método que permite tomar uma decisão entre duas ou mais hipóteses (hipótese nula H0 ou hipótese alternativa H1), utilizando os dados observados de uma determinada experimentação. [66] Entende-se por hipótese nula a hipótese assumida como verdadeira para a construção do teste, ou seja, a alternativa que se está interessado em testar, enquanto que a hipótese alternativa, como o nome indica, é a alternativa à hipótese nula [66].

Para além disso existem dois conceitos de erros a reter [66]: erros do tipo I, que consiste na probabilidade de se rejeitar a hipótese nula quando ela é verdadeira, e erros do tipo II, que consiste na probabilidade de rejeitar a hipótese alternativa quando ela é verdadeira. Estes dois conceitos dão então origem à tabela abaixo

Tabela 7.1: Relação entre cada hipótese e os diferentes tipos de erro

	Hipótese nula é verdadeira	**Hipótese nula é falsa**
Hipótese nula é rejeitada	Erro do tipo I	Não há erro
Hipótese nula não é rejeitada	Não há erro	Erro do tipo II

Um dos principais objetivos desta dissertação consiste no estudo da adoção de tecnologia *serverless*, com vista a determinar se a adoção da mesma é viável, em que situações fará sentido adotá-la e quais as suas principais vantagens e desvantagens. Desta forma, neste contexto, as hipóteses em avaliação são então:

- Hipótese nula - O desempenho obtido nas duas abordagens implementadas não apresenta diferenças significativas
- Hipótese alternativa - O desempenho obtido numa das abordagens é significativamente distinto do desempenho da outra abordagem

7.2 Metodologia

Para a realização desta avaliação foi comparado o desempenho da aplicação desenvolvida com base numa arquitetura orientada a microserviços com a aplicação desenvolvida de acordo com uma arquitetura *serverless*.

Para testar a performance das duas aplicações desenvolvidas foram utilizadas duas métricas: o tempo de resposta (valor mínimo, valor máximo e mediana) e o rendimento. O tempo de resposta, como o nome indica, corresponde ao tempo que um cliente demora a obter uma resposta por parte da aplicação. Por outras palavras, o tempo de resposta corresponde ao tempo que decorre entre um pedido e uma resposta [67]. Já o rendimento corresponde ao número de pedidos que uma aplicação consegue suportar com sucesso por segundo, o que significa dividir o número total de pedidos processados com sucesso pelo tempo que foi necessário para processar estes pedidos.

Os resultados para cada métrica foram obtidos através da utilização da ferramenta Artillery [68], que permite simular pedidos a cada serviço e função. Para a avalição foi utilizado a título de exemplo a função *serverless* createOrder e, para a abordagem orientada a microserviços, o *endpoint* responsável pela mesma criação no microserviço de encomendas. De notar que os testes foram realizados em máquinas da nuvem com configurações semelhantes e da mesma região, de forma a que o teste fosse o mais justo possível.

7.3 Resultados

7.3.1 Cenário de Teste 1

O primeiro cenário de teste simula a submissão de três pedidos por segundo durante um minuto, o que equivale a um total de 180 pedidos. Na tabela 7.2 encontram-se apresentados os resultados deste cenário de teste para as duas abordagens.

Tabela 7.2: Resultados da execução do caso de teste 1

Métrica	**Orientada a microserviços**	**Serverless**
Número total de pedidos		
OK	180	180
Erro	0	0
Rendimento	2.97 pedidos/s	2.99 pedidos/s
Tempo de resposta		
Valor Mínimo	209 ms	188 ms
Valor Máximo	240 ms	215 ms
Mediana	219 ms	193 ms

7.3.2 Cenário de Teste 2

O segundo cenário de teste simula a submissão de três pedidos por segundo durante seis minutos, o que equivale a um total de 1080 pedidos. Na tabela 7.3 encontram-se apresentados os resultados deste cenário de teste para as duas abordagens.

Tabela 7.3: Resultados da execução do caso de teste 2

Métrica	**Orientada a microserviços**	**Serverless**
Número total de pedidos		
OK	1080	1080
Erro	0	0
Rendimento	2.98 pedidos/s	2.99 pedidos/s
Tempo de resposta		
Valor Mínimo	208 ms	187 ms
Valor Máximo	239 ms	230 ms
Mediana	218 ms	198 ms

7.3.3 Cenário de Teste 3

O terceiro cenário de teste simula a submissão de cinco pedidos por segundo durante seis minutos, o que equivale a um total de 1800 pedidos, e a um aumento do número de pedidos por segundo em comparação com o segundo cenário o que permite avaliar a escalabilidade das aplicações. Na tabela 7.4 encontram-se apresentados os resultados deste cenário de teste para as duas abordagens.

Tabela 7.4: Resultados da execução do caso de teste 3

Métrica	**Orientada a microserviços**	**Serverless**
Número total de pedidos		
OK	1800	1800
Erro	0	0
Rendimento	4.98 pedidos/s	5 pedidos/s
Tempo de resposta		
Valor Mínimo	205 ms	175 ms
Valor Máximo	641 ms	319 ms
Mediana	215 ms	191 ms

7.4 Análise de resultados

Para avaliar se as diferenças entre os desempenhos das duas aplicações nos cenários de teste realizados foram significativas, foi realizado o teste de hipóteses não paramétrico de Mann-Whitney [69]. Para a realização deste teste executou-se um *script* escrito em Python, que se encontra apresentado no excerto de código 7.1, tendo-se utilizado as medianas obtidas em cada cenário de teste para cada abordagem. Neste script utilizou-se o módulo *SciPy.stats*, que disponibiliza as funções necessárias para a realização deste tipo de testes.

```
1 from scipy.stats import mannwhitneyu
2
3 micro_services_results = [219,218,193]
4
5 serverless_results =  [193,198,191]
6
7 print(mannwhitneyu(micro_services_results,serverless_results))
```

Código 7.1: Script executado para a realização do teste não paramétrico de Mann-Whitney

Na realização deste teste obteve-se um *p-value* de 0.13 (utilizando um nível de significância de 0.05). Assim, como o *p-value* obtido é superior ao nível de significância utilizado não se pode rejeitar a hipótese nula e portanto, pode-se concluir que a diferença de desempenho nas abordagens, embora seja ligeiramente superior na abordagem *serverless*, não é significativa.

Capítulo 8

Conclusão

Neste capítulo são retiradas algumas conclusões sobre o trabalho desenvolvido, analisado o estado dos objetivos propostos à partida e descritas algumas limitações encontradas e o trabalho futuro a desenvolver.

O desenvolvimento *serverless* tem evoluído nos últimos anos, com o aparecimento de cada vez mais ferramentas com a finalidade de minimalizar os problemas apresentados por este tipo de desenvolvimento e a simplificar alguns dos processos, como é o caso da implantação das soluções.

No que diz respeito à arquitetura, também existiram mudanças nos últimos anos. Se antes as empresas optavam por arquiteturas monolíticas, com a adoção de metodologias de desenvolvimento ágil começaram a adotar também arquiteturas mais ágeis, como é o caso das arquiteturas de microserviços [70]. A arquitetura *serverless*, enquanto arquitetura ágil, vai também ela de encontro a esta mudança de paradigma, oferecendo às organizações escalabilidade e uma gestão simplificada das suas arquiteturas.

Ainda que o crescimento da utilização de arquiteturas *serverless* seja notório existem ainda algumas reticências na adoção deste tipo de desenvolvimento por parte das empresas e, por este motivo, era importante aferir a validade do conjunto de recomendações proposto com vista a eliminar ou minimizar as dificuldades sentidas na adoção do desenvolvimento *serverless*.

A primeira recomendação, relacionada com a identificação de problemas que a arquitetura atual não consegue resolver e que se espera que a arquitetura *serverless* resolva, revelou-se importante pois para além de ter ajudado no processo de escolha da stack tecnológica a utilizar na implementação, percebeu-se que existe um conjunto de nichos de mercado, como os sistemas bancários, governamentais, da área da saúde, entre outros, onde a arquitetura atual apresenta algumas limitações e para os quais a adoção de uma arquitetura *serverless* tende a ser mais útil.

Outra recomendação que se revelou importante foi a terceira, relacionada com o desenvolvimento da prova de conceito, pois este desenvolvimento e a sua avaliação permitiu perceber que, ao contrário do que se esperava e ainda que seja ligeiramente superior, o desempenho da aplicação *serverless* não é significativamente superior quando comparado ao desempenho de uma aplicação com uma arquitetura atual, como é o caso das arquiteturas orientadas a microserviços. No entanto, o facto de ter conseguido resultados ligeiramente superiores e os benefícios em termos de custo e implantação que oferece podem fazer com que, com a realização de algum trabalho, no futuro possa crescer ainda mais.

Por fim, as recomendação cujo trabalho desenvolvido não permitiu retirar conclusões significativas foram as recomendações R5 e R6, ainda que no que diz respeito à quinta recomendação, relacionada com a automação de testes, a utilização da framework Artillery se tenha revelado benéfica.

8.1 Objetivos concluídos

O trabalho desenvolvido permitiu obter uma percepção bastante maior sobre o que são as tecnologias *serverless*, qual o seu estado atual e que perspectivas para o futuro existem em relação às mesmas. Permitiu perceber também que quando, no desenvolvimento das funções *serverless*, são adotadas linguagens nas quais os desenvolvedores estão confortáveis, a curva de aprendizagem tende a ser combatida com maior facilidade.

Assim, o estado dos objetivos propostos à partida é o apresentado na tabela 6.1.

Tabela 8.1: Estado dos objetivos propostos inicialmente

Objetivo	**Estado**
T1 - Identificar e sistematizar em que consiste o paradigma *serverless* e práticas comuns de adoção	Atingido (2.1.3)
T2 - Identificar e sistematizar as principais dificuldades/razões que atualmente condicionam a adoção de *serverless* e a sua relevância dependendo do tipo de projeto/aplicação	Atingido (2.1.3)
T3 - Identificar, descrever e sintetizar características/qualidades que as aplicações/sistemas devem reunir de modo a que desenvolver as mesmas em/para *serverless* seja efetivamente útil	Atingido (2.1.3)
T4 - Identificar, descrever e sintetizar as abordagens, ferramentas e tecnologias que suportam o desenvolvimento *serverless*, fazendo uma distinção entre elas, por exemplo, ao nível do grau de (in)dependência da plataforma cloud utilizada	Atingido (2.2)
T5 - Avaliar com base na literatura e alguma experimentação a pertinência e adequação das abordagens, ferramentas e tecnologias que suportam o desenvolvimento *serverless*	Atingido (2.2)
T6 - Avaliar os efeitos/limitações/consequências de adoção de tecnologias dependentes da plataforma cloud quando comparadas com tecnologias independentes	Atingido (2.2.3)
T7 - Identificar e desenhar um conjunto de recomendações (e.g. ajustes) que devem ser adotadas no desenvolvimento de aplicações e respetivas arquiteturas com vista a eliminar e/ou minimizar as dificuldades/razões previamente identificadas que condicionam a adoção de *serverless*	Atingido (2.3)
T8 - Desenhar uma solução *serverless* para as aplicações usadas como casos de estudo	Atingido (5)
T9 - Construir a solução *serverless* anteriormente desenhada para as aplicações usadas como casos de estudo	Atingido (6)
T10 - Recolher informação, da solução desenvolvida,de alguns requisitos de qualidade como a performance, desempenho, disponibilidade e escalabilidade que permita fazer uma comparação com os valores normalmente obtidos em aplicações "no-serverless" semelhantes	Atingido (7)

Como é possível concluir pela análise da tabela, foram atingidos todos os objectivos propostos à partida.

8.2 Limitações e trabalho futuro

A principal limitação encontrada no desenvolvimento da dissertação foi o facto de o pensamento *serverless* ser relativamente recente, o que fez com que por vezes existisse alguma falta de informação, o que obrigou a que existisse um maior esforço na exploração e procura de informação. Para além disso, o facto de este tipo de tecnologia ser também ele novo para o autor da dissertação levou a que existisse um processo de auto-aprendizagem de forma a ficar a conhecer melhor o processo de desenvolvimento *serverless*.

Como referido anteriormente o desempenho de abordagem *serverless* não se mostrou significativamente superior ao desempenho das abordagens atuais, como é o caso da abordagem orientada a microserviços, e ainda que apresente outras vantagens ao nível da monetização e da implantação, espera-se que no futuro seja realizado algum trabalho de estudo e evolução da tecnologia para que se possa obter maior proveito daquilo que são as vantagens da mesma e para que a sua escalabilidade seja ainda maior.

Bibliografia

[1] Forrest Stroud. *On Premises*. url: https://www.webopedia.com/definitions/on-premises/.

[2] Mike Roberts e John Chapin. «What is Serverless? - Understanding the Latest Advances in Cloud and Service-Based Architecture». Em: (2017).

[3] Avi. *An Introduction to Cloud Service Models*. url: https://geekflare.com/cloud-service-models/.

[4] Gurudatt Kulkarni, Ramesh Sutar e Jayant Gambhir. «Cloud computing-Infrastructure as service-Amazon EC2». Em: (2012).

[5] Amazon Web Service. url: https://aws.amazon.com/pt/.

[6] App Engine. url: https://cloud.google.com/appengine.

[7] Heroku. url: https://dashboard.heroku.com.

[8] Kubernetes. url: https://kubernetes.io.

[9] *Amazon Lambda - AWS*. url: https://aws.amazon.com/pt/lambda/.

[10] Microsoft Office 365. url: https://www.microsoft.com/en-us/microsoft-365.

[11] Cisco. «Places in the Network: Secure Cloud». Em: (2019).

[12] Manoj Kumar. «Serverless Architectures Review, Future Trend and the Solutions to Open Problems». Em: (2019).

[13] Eurostat. *Households - type of connection to the internet*. url: https://ec.europa.eu/eurostat/databrowser/view/isoc_ci_it_h/default/line?lang=en.

[14] Sherweb. *Cost of server ownership: on-premise vs. IaaS*. url: https://www.sherweb.com/blog/cloud-server/total-cost-of-ownership-of-servers-iaas-vs-on-premise/.

[15] Eurostat. *Individuals - use of cloud services*. url: https://ec.europa.eu/eurostat/databrowser/view/isoc_cicci_use%5C$DV_514/default/line?lang=en.

[16] Eurostat. *Cloud computing services*. url: https://ec.europa.eu/eurostat/databrowser/view/isoc_cicce_use/default/line?lang=en.

[17] Flexera. *Cloud Computing Trends: 2020 State of the Cloud Report*. url: https://www.flexera.com/blog/industry-trends/trend-of-cloud-computing-2020/.

[18] IDG. *The 2020 IDG Cloud Computing Survey*. url: https://www.infoworld.com/article/3561269/the-2020-idg-cloud-computing-survey.html.

[19] Joe Hellerstein. «The State of the Serverless Art». Em: (2020).

[20] Chris Guzikowski Roger Magoulas. *O'Reilly serverless survey 2019: Concerns, what works, and what to expect*. url: https://www.oreilly.com/radar/oreilly-serverless-survey-2019-concerns-what-works-and-what-to-expect/.

[21] Grand View Research. *Serverless Architecture Market Size, Share & Trends Analysis Report By Organization (SME, Large Enterprises), By Vertical, By Service (Automation & Integration), And Segment Forecasts 2018 - 2025*. 2018. url: https://www.grandviewresearch.com/industry-analysis/serverless-architecture-market.

[22] *Azure Functions serverless compute*. url: https://azure.microsoft.com/en-gb/services/functions/.

[23] *Google Cloud Functions*. url: https://cloud.google.com/functions.
[24] Rafal Gancarz. *An essential guide to the serverless ecosystem*. url: https://techbeacon.com/enterprise-it/essential-guide-serverless-ecosystem.
[25] *Amazon DynamoDB*. url: https://aws.amazon.com/pt/dynamodb/.
[26] *Amazon Kinesis*. url: https://aws.amazon.com/pt/kinesis/.
[27] Dilip Geevarghese. *What Azure Functions are and How You Can Use Them to Run Small Applications Effortlessly*. url: https://www.cabotsolutions.com/what-azure-functions-are-and-how-you-can-use-them-to-run-small-applications-effortlessly.
[28] Google. *Google Cloud Pub/Sub*. url: https://cloud.google.com/pubsub.
[29] Google. *Google Cloud Storage*. url: https://cloud.google.com/storage.
[30] Sulav Malla e Ken Christensen. «HPC in the cloud: Performance comparison of function as a service (FaaS) vs infrastructure as a service (IaaS)». Em: (2019).
[31] Google. *Pacote de operações do Google Cloud*. url: https://cloud.google.com/products/operations.
[32] César Rodríguez, Fernando Alvarez e Martín Evgeniev. «Serverless». Em: (2019).
[33] *Serverless Framework*. url: https://github.com/serverless/serverless.
[34] *Architect*. url: https://arc.codes/docs/en/guides/get-started/why-architect.
[35] Michael Wittig, Andreas Wittig, Ben Whaley e outros. *Amazon web services in action*. 2018.
[36] *AWS CloudFormation*. url: https://aws.amazon.com/pt/cloudformation/.
[37] Serkan Özal. *Introduction to Architect*. url: https://blog.thundra.io/introduction-to-the-architect-framework.
[38] *Up*. url: https://apex.sh/up/.
[39] *Pulumi*. url: https://pulumi.com.
[40] Chris Munns. *Announcing Ruby Support for AWS Lambda*. url: https://aws.amazon.com/pt/blogs/compute/announcing-ruby-support-for-aws-lambda/.
[41] *Jets: The Ruby Serverless Framework*. url: https://rubyonjets.com.
[42] *Ruby on Rails*. url: https://rubyonrails.org.
[43] *Middy*. url: https://github.com/middyjs/middy.
[44] *Sigma*. url: https://www.slappforge.com/sigma.
[45] Chandan Kumar. *5 Best Serverless Security Platform for Your Applications*. url: https://geekflare.com/serverless-application-security/.
[46] Andrea Passwater. *The State of Serverless Multi-cloud*. url: https://www.serverless.com/blog/state-of-serverless-multi-cloud.
[47] Sebastien Goasguen. *Serverless and multi-cloud: Hype or reality?* url: https://techbeacon.com/enterprise-it/serverless-multi-cloud-hype-or-reality.
[48] AVI Networks. *Multi-Cloud*. url: https://avinetworks.com/glossary/multi-cloud/.
[49] Marcia Villalba. *6 Things to Know Before Migrating An Existing Service to Serverless*. url: https://www.serverless.com/blog/6-things-to-know-before-migrating-an-existing-service-to-serverless.
[50] P. Koen, G. Ajamian, R. Burkart, A. Clamen, J. Davidson, R. D'Amore, C. Elkins, K. Herald, M. Incorvia, A. Johnson, R. Karol, R. Seibert, A. Slavejkov e K. Wagner. «Providing Clarity and A Common Language to the "Fuzzy Front End"». Em: (2001).
[51] P. Koen, G. Ajamian, S. Boyce, A. Clamen, E. Fisher, S. Fountoulakis, A. Johnson, P. Puri e R. Seibert. «Fuzzy front end: effective methods, tools, and techniques». Em: (2002).

[52] Atte Martikainen. «Front End of Innovation in Industrial Organization». Tese de doutoramento. Nov. de 2017.
[53] Valarie Zeithaml. «Consumer Perceptions of Price, Quality and Value: A Means-End Model and Synthesis of Evidence». Em: (1988).
[54] G.A. Churchill e J.P. Peter. *Marketing: Creating Value for Customers*. 1998.
[55] David Aaker. *Construindo Marcas Fortes*. 2007.
[56] Alexander Osterwalder e Yves Pigneur. «An eBusiness Model Ontology for Modeling eBusiness». Em: (2002).
[57] Alexander Osterwalder e Yves Pigneur. «Business Model Generation: A Handbook for Visionaries, Game Changers, and Challengers». Em: (2010).
[58] Verna Allee. «Value network analysis and value conversion of tangible and intangible assets». Em: (2008).
[59] Thomas Saaty. «The Analytic Hierarchy Process: Planning, Priority Setting, Resource Allocation». Em: (1980).
[60] Gerald Kotonya e Ian Sommerville. *Requirements engineering : processes and techniques*. 1998.
[61] Deborah Caswell e Robert Grady. «Software Metrics: Establishing a Company-wide Program». Em: (1987).
[62] Peter Eeles. «Capturing architectural requirements». Em: (2005).
[63] Eric Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. 2014.
[64] Mark Masse. *REST API Design Rulebook*. 2011.
[65] RapidAPI Staff. «The Top 9 Payment APIs To Manage Your Payments». Em: (). url: `https://rapidapi.com/blog/top-payment-apis-manage-your-payments/`.
[66] Franklin A Graybill, Hariharan K Iyer e Richard K Burdick. *Applied statistics: A first course in inference*. 1998.
[67] Omar Al-Debagy. «A Comparative Review of Microservices and Monolithic Architectures». Em: (2018), pp. 000149–000154.
[68] Artillery. url: `https://artillery.io/`.
[69] Henry B Mann e Donald R Whitney. «On a test of whether one of two random variables is stochastically larger than the other». Em: *The annals of mathematical statistics* (1947), pp. 50–60.
[70] Mike Loukides e Steve Swoyer. *Microservices Adoption in 2020*. url: `https://www.oreilly.com/radar/microservices-adoption-in-2020`.

Apêndice A

Design - Diagramas

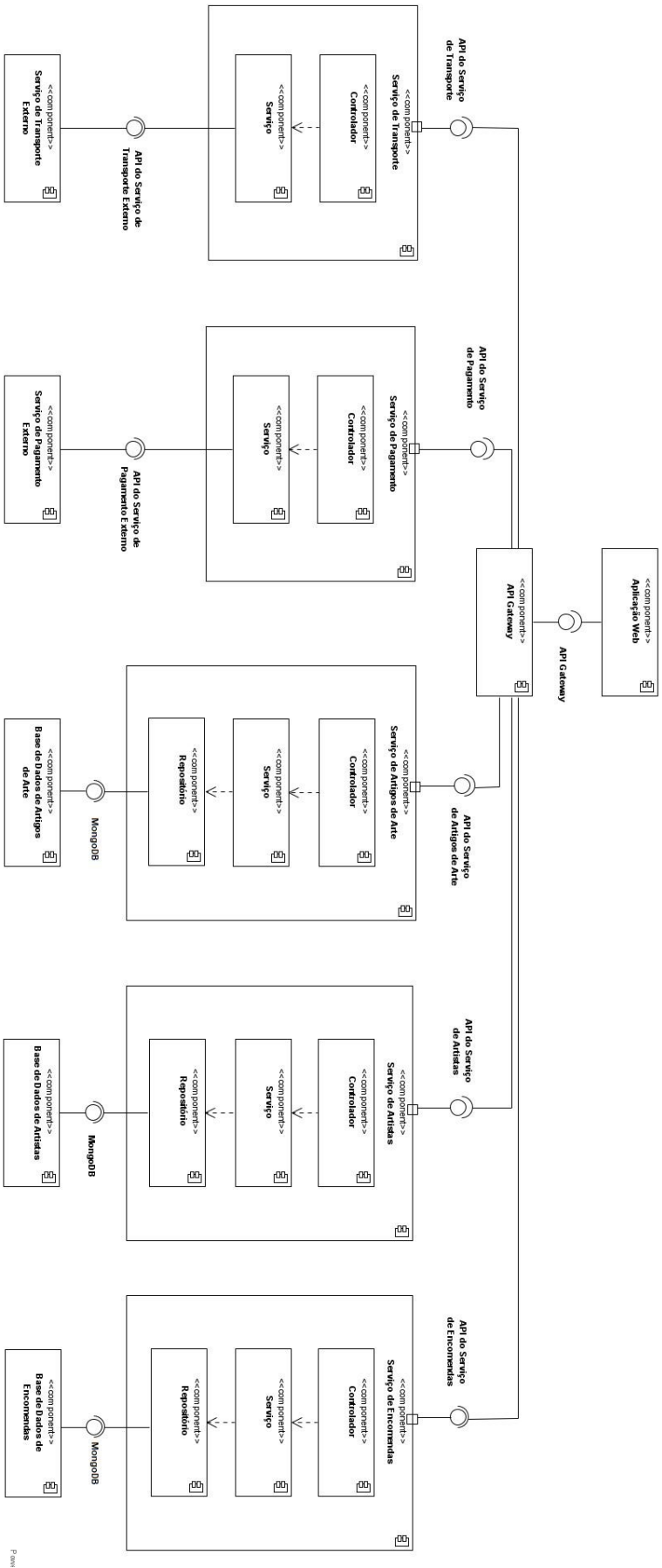

Figura A.1: Arquitetura da abordagem orientada a microserviços

Figura A.2: Arquitetura da abordagem *serverless*

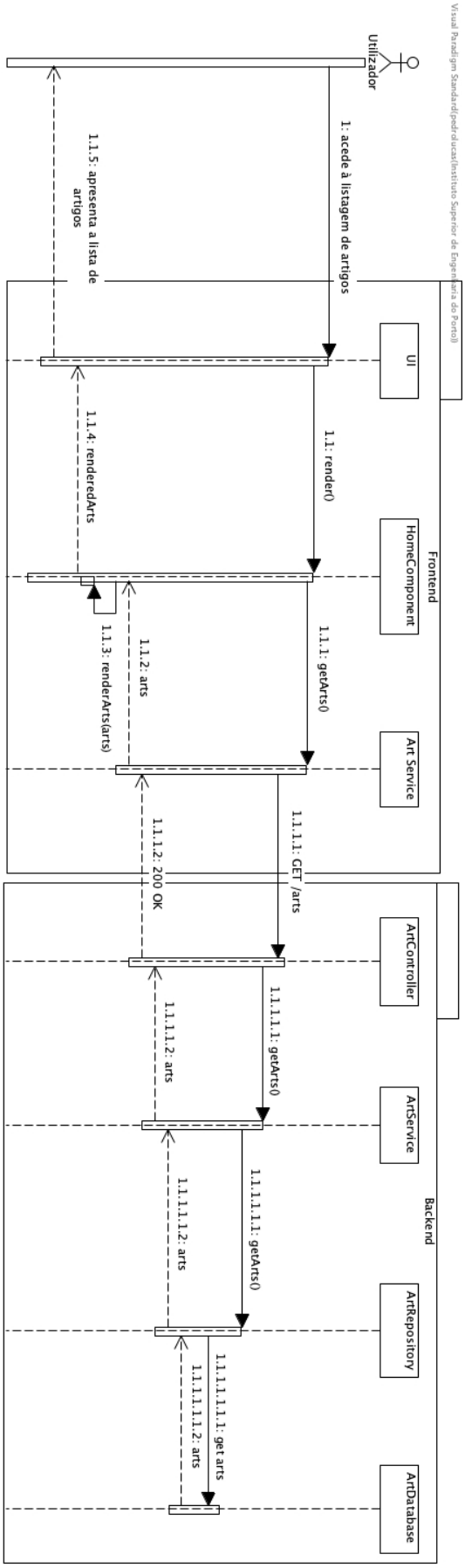

Figura A.3: Visualizar Lista de Artigos - Abordagem orientada a microserviços

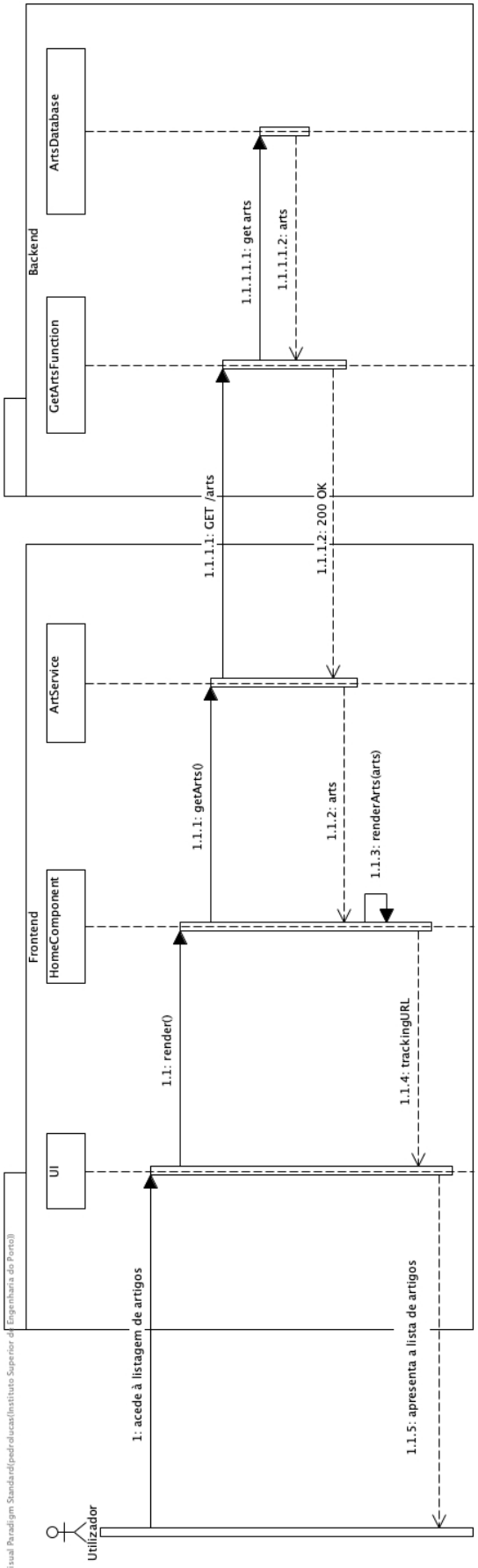

Figura A.4: Visualizar Lista de Artigos - Abordagem *serverless*

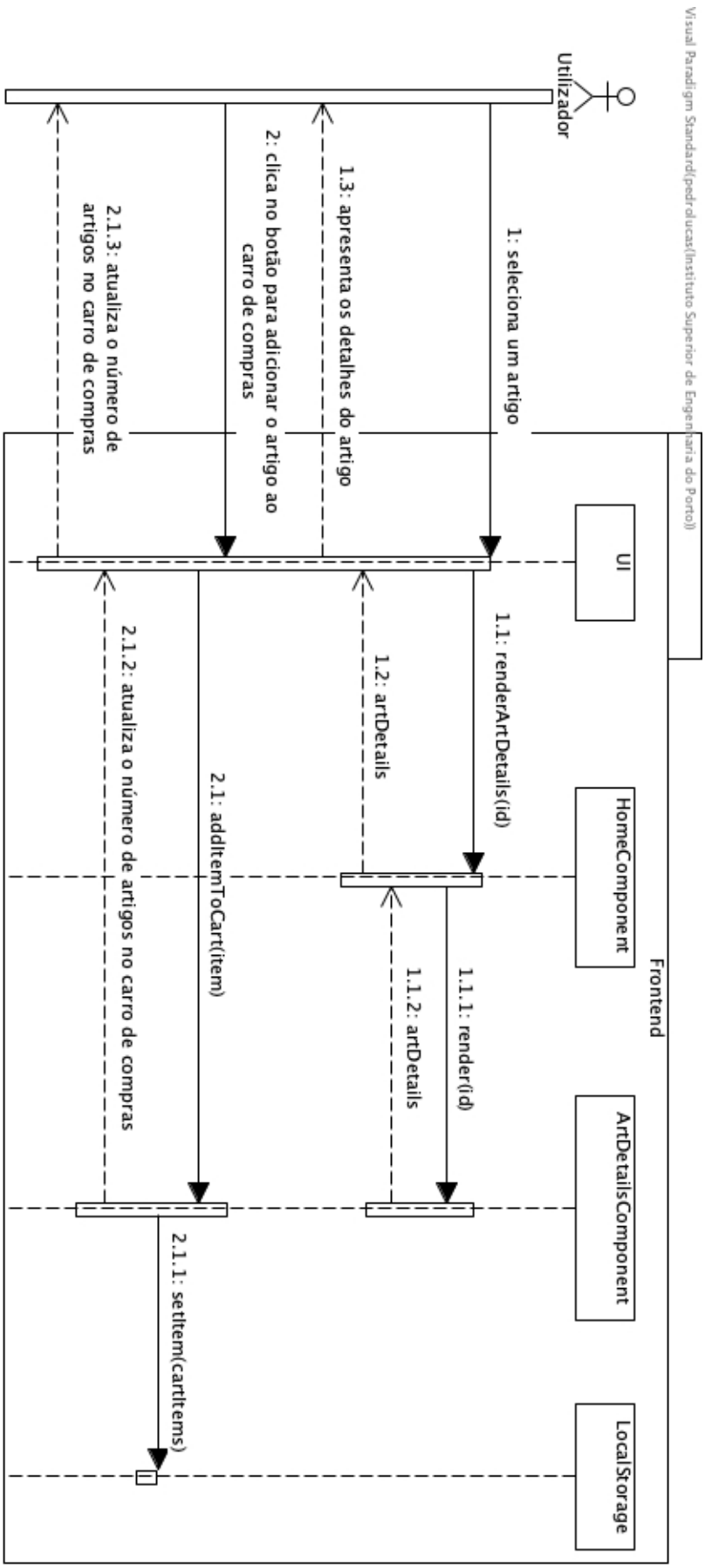

Figura A.5: Diagrama de sequência da adição de artigos ao carro de compras

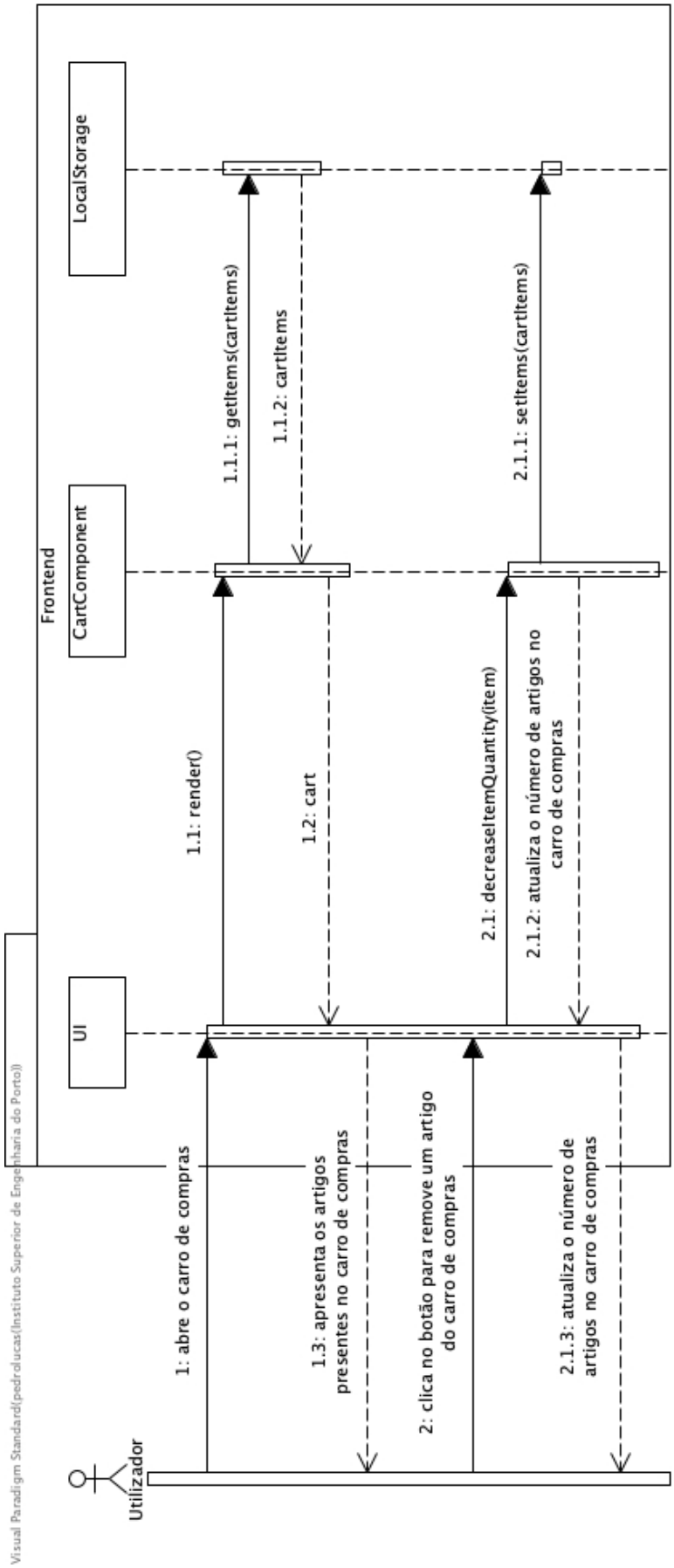

Figura A.6: Diagrama de sequência da remoção de artigos do carro de compras

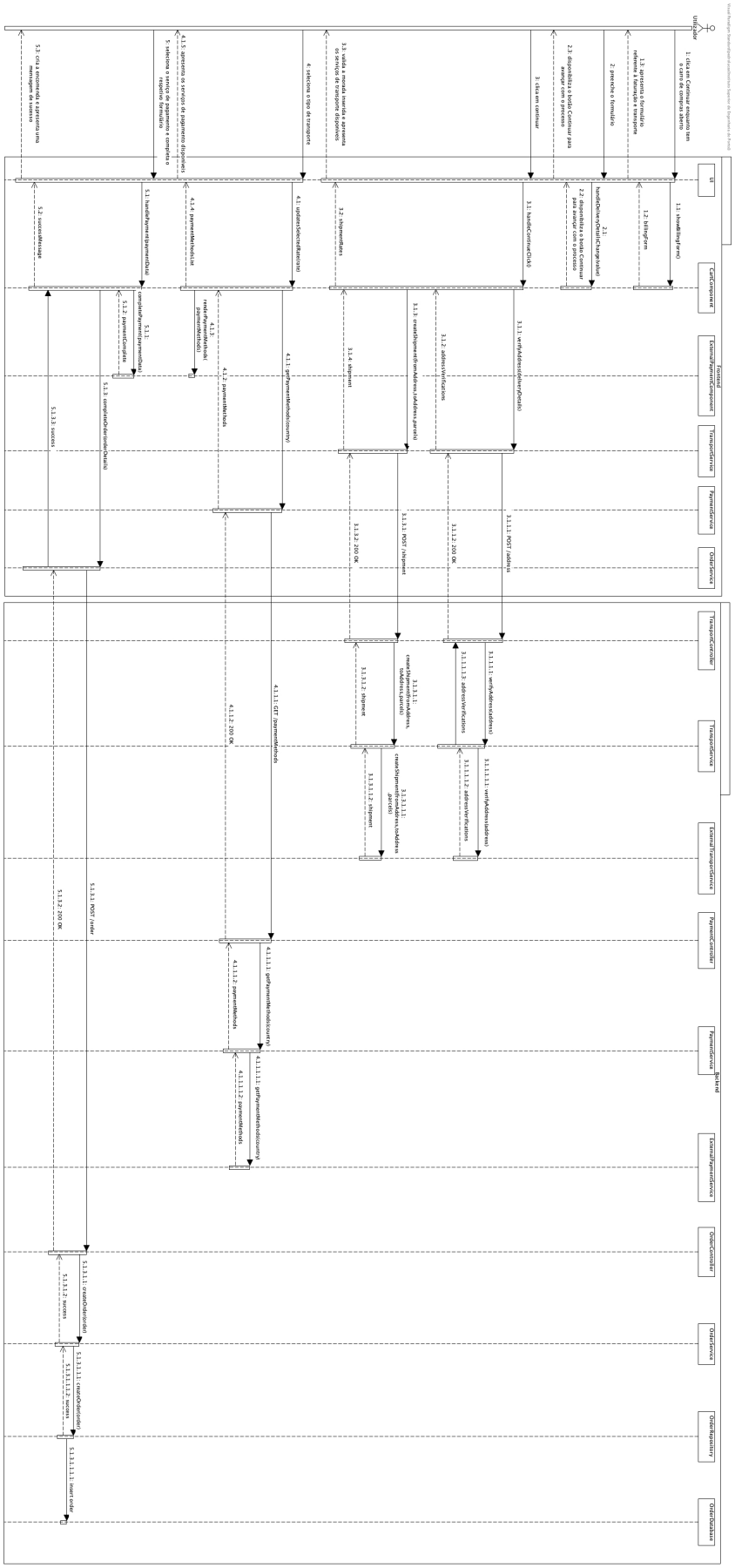

Figura A.7: Fazer encomenda - Abordagem orientada a microserviços

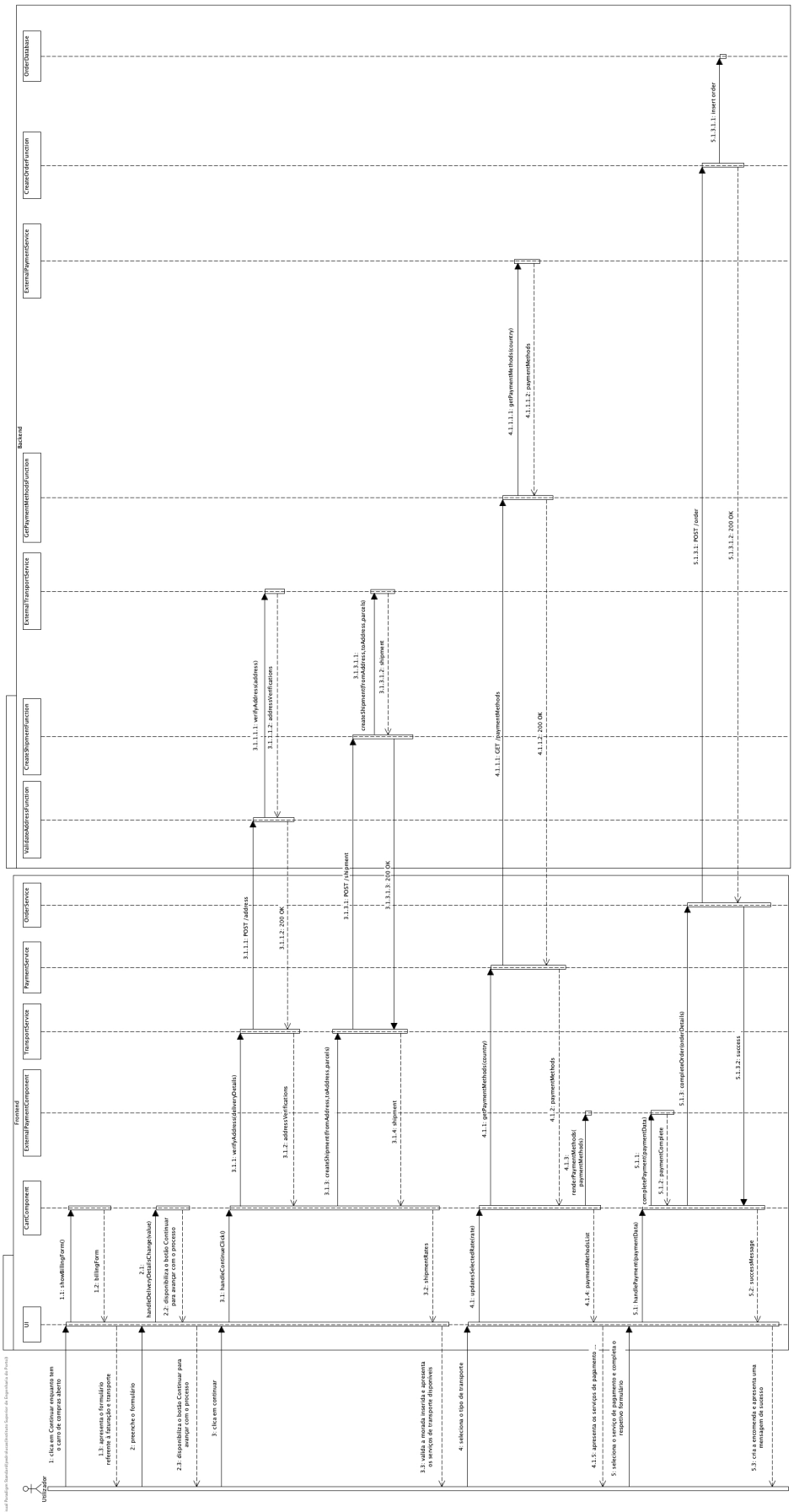

Figura A.8: Fazer encomenda - Abordagem *serverless*

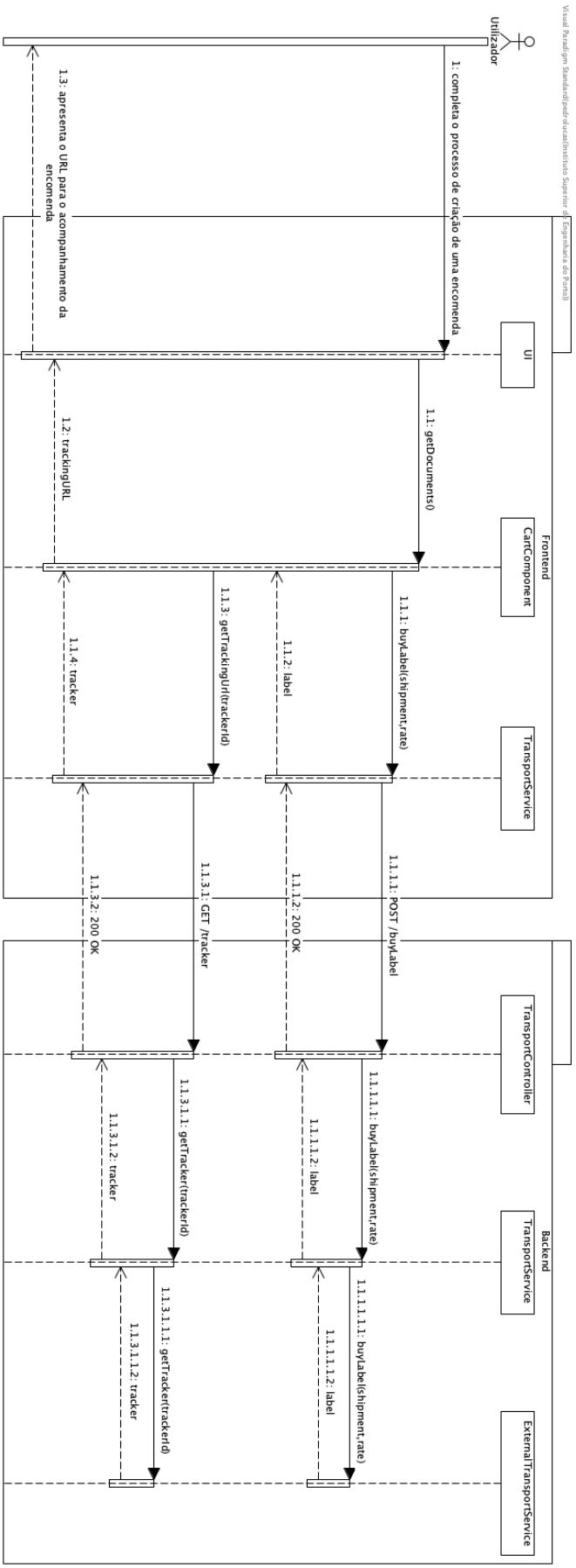

Figura A.9: Acompanhar encomenda - Abordagem orientada a microserviços

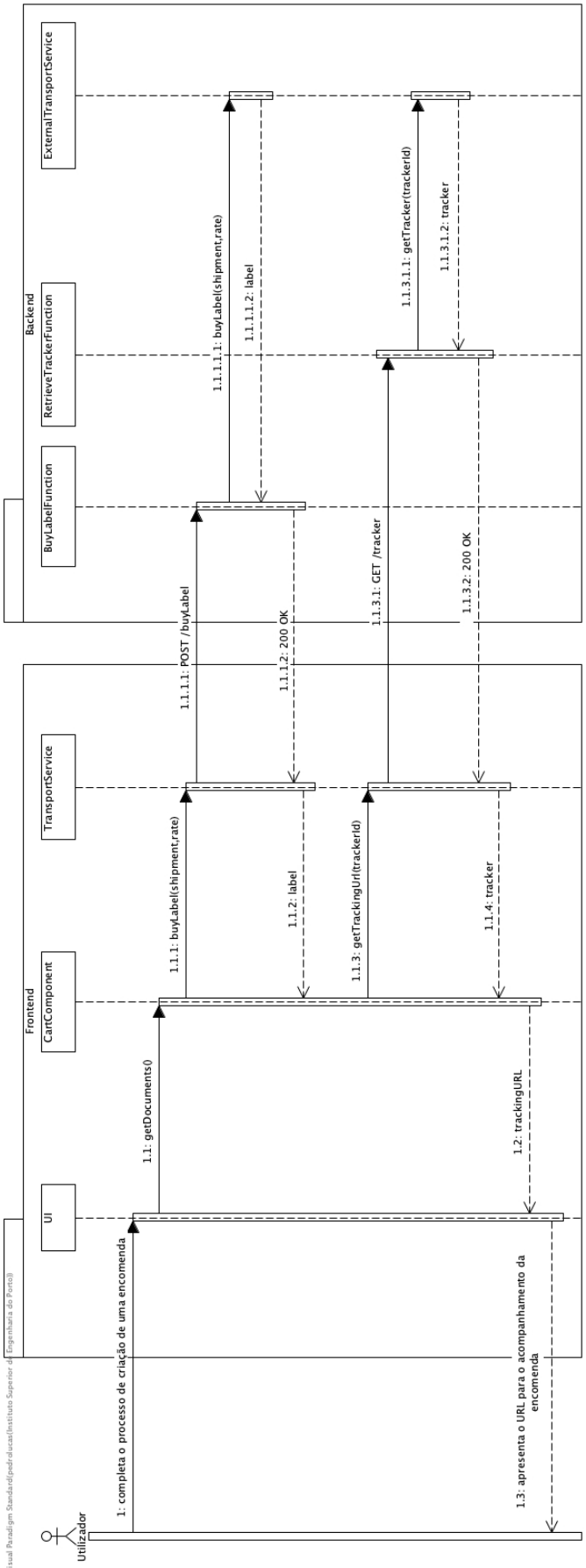

Figura A.10: Acompanhar encomenda - Abordagem *serverless*