Faculdade de Ciências e Tecnologia
Departamento de Engenharia Informática

PLATAFORMAS DE COMPUTAÇÃO *SERVERLESS*:

Estudo e *Benchmark*

Horácio José Morgado Martins

Dissertação no âmbito do Mestrado em Engenharia Informática, especialização em Engenharia de Software orientada pelo Professor Doutor Paulo Rupino da Cunha e pelo Professor Doutor Filipe Araújo e apresentada à Faculdade de Ciências e Tecnologia / Departamento de Engenharia Informática.

Janeiro de 2019

Esta página foi propositadamente deixada em branco.

Resumo

Existem várias plataformas que suportam o paradigma de computação *serverless* na *cloud*. Perceber se estas plataformas são suficientemente maduras e oferecem uma performance que as permita aplicar a contextos de utilização real, é essencial para que se possa optar por uma delas. Estudos de performance e *benchmarks* existentes na área revelam-se lacunares e não suficientemente abrangentes, pelo que surgiu a necessidade de abordar esta temática.

No decorrer deste trabalho, é realizada a identificação e estudo das plataformas *serverless* existentes, sendo estas comparadas em termos de funcionalidades e características, através da utilização de um conjunto de parâmetros apresentado. É desenhado e proposto um grupo de testes para um *benchmark* de utilização genérica e com vista à automatização da sua execução, é apresentada uma arquitetura de um sistema.

Procedeu-se à implementação do sistema, que foi utilizado para a execução do conjunto de testes que constitui o *benchmark*. Foram testadas os serviços de computação *serverless AWS Lambda*, *Azure Functions*, *Google Cloud Functions* e *OpenWhisk*. Com os resultados obtidos foi efetuada uma observação de como estas se comportam em diferentes cenários, tendo também sido efetuada uma comparação entre elas.

Por último, procedeu-se à criação de uma demonstração da utilização da *OpenWhisk* para a formação de um serviço de mobilidade que permite a otimização de rotas de um fluxo de veículos.

Pensa-se que o conjunto de testes proposto e o sistema de execução do *benchmark* consistirá num bom auxiliar para pessoas que pretendam realizar testes de performance às plataformas de computação *serverless*, de forma a tomarem as melhores decisões, conforme as necessidades, no momento de utilizarem uma delas.

Palavras-Chave

Computação *serverless*, *Workflows serverless*, Plataformas *serverless*, *Function-as-a-Service*, *Benchmark*.

Esta página foi propositadamente deixada em branco.

Abstract

There are several platforms that supports the paradigm of serverless cloud computing. Realizing if these platforms are sufficiently mature and deliver performance that allows them to be applied to real-life contexts is essential so that we can chose one. Existing performance studies and benchmarks in the area presented a gap and are not very comprehensive, and therefore the need to address this issue has arisen.

In the course of this internship, an identification and a study of the existing serverless platforms is carried out, being these compared in terms of functionalities and characteristics, through the use of a presented set of parameters. It is also designed and proposed a test suite for a benchmark of generic us and in order to automate its execution, an architecture of a system is presented.

The implementation of the system was performed, which was used to run the test suite that constitutes the benchmark. Serverless computing platforms *AWS Lambda*, *Azure Functions*, *Google Cloud Functions* and *OpenWhisk* were tested. With the obtained results, an observation was made of how the platforms behave in different scenarios and a comparison was also made between them.

Finally, a demonstration of the use of OpenWhisk for the creation of a mobility service that allows the optimization of routes of a flow of vehicles has been created.

It is thought that the proposed test suite and the benchmark execution system will be a good helper for people who want to perform performance tests on serverless computing platforms in order to make the best decisions, according to the needs, when is the moment using one of them.

Keywords

Serverless computing, Serverless Workflows, Serverless platforms, Function-as-a-Service, Benchmark.

Esta página foi propositadamente deixada em branco.

Agradecimentos

O trabalho apresentado nesta tese foi parcialmente realizado no âmbito do projeto MobiWise: *From mobile sensing to mobility advising* (P2020 SAICTPAC/0011/2015), cofinanciado pelo COMPETE 2020, Portugal 2020 - Programa Operacional de Competitividade e Internacionalização (POCI), do FEDER (Fundo Europeu de Desenvolvimento Regional) da União Europeia e da Fundação para a Ciência e Tecnologia (FCT).

Esta página foi propositadamente deixada em branco.

Conteúdo

1 Introdução **1**

1.1 Enquadramento e objetivos . . . 1
1.2 Contribuições . . . 2
1.3 Estrutura do documento . . . 3

2 Planeamento **5**

2.1 Cronograma planeado . . . 5
2.2 Cronograma executado . . . 9

3 Conceitos e tecnologias base **13**

3.1 Computação em *Cloud* . . . 13
3.1.1 Características de uma *cloud* . . . 14
3.1.2 Modelos de *deploy* de *cloud* . . . 15
3.1.3 Arquitetura de *cloud* . . . 15
3.1.4 Modelo de *delivery* de serviços da *cloud* . . . 16
3.2 *Software Containers* . . . 18
3.2.1 *Docker* . . . 21
3.2.2 *Docker Swarm* . . . 22
3.2.3 *Kubernets* . . . 23
3.3 Computação *Serverless* . . . 24
3.4 *Workflows Serverless* . . . 25

4 Plataformas *Serverless* **27**

4.1 *AWS Lambda* . . . 27
4.2 *Microsoft Azure Functions* . . . 31
4.3 *Google Cloud Functions* . . . 33
4.4 *Apache OpenWhisk/IBM Cloud Functions* . . . 35
4.5 *Picasso* . . . 40
4.6 *Qinling* . . . 40
4.7 *OpenFaaS* . . . 41
4.8 *Fission* . . . 42

5 Comparação das plataformas *Serverless* **45**

6 *Benchmarking* de plataformas *Serverless* **53**

6.1 *Benchmarkings* existentes . . . 53
6.2 Proposta de *benchmark* de plataformas *serverless* . . . 62

6.2.1 Propostas de testes para o *benchmark* 62
6.2.2 Sistema de execução do *Benchmark* 67

7 Implementação e configuração do sistema de execução do *Benchmark* 71

7.1 Ferramenta auxiliar *Serverless Framework* 71
7.1.1 Instalação da *Serverless Framework* 72
7.1.2 Configuração da *AWS Lambda* 72
7.1.3 Configuração da *Google Cloud Functions* 73
7.1.4 Configuração da *Azure Functions* 74
7.1.5 Configuração da *OpenWhisk/IBM Cloud Functions* 74
7.1.6 Criação, configuração e *deploy* de funções 75
7.2 *JMeter* 77
7.2.1 Instalação e utilização do *JMeter* 77
7.2.2 Criação do plano de teste padrão para execução de testes no *JMeter* 78
7.2.3 Métricas obtidas da execução de testes no *JMeter* 78
7.3 Interface de linha de comandos e aplicação de gestão de testes 80
7.3.1 Ficheiro de configuração da plataforma 81
7.3.2 Interface de linha de comandos 82
7.3.3 Controlador de *deploy* 86
7.3.4 Controlador de testes 86
7.3.5 Controlador de resultados 87
7.4 Resumo do Capítulo 88

8 Execução do *Benchmark* e Resultados 89

8.1 Termos de serviço e licenças para a realização de *benchmarks* e divulgação de resultados 90
8.2 *T1 - Teste de overhead das plataformas serverless* 90
8.3 T2 - Teste de carga concorrente 92
8.4 T3 - Teste de reutilização de *containers* 94
8.5 T4 - Teste de impacto do tamanho do *payload* 95
8.6 T5 - Teste de impacto da linguagem de programação na performance 96
8.7 T6 - Teste de performance em função da memória alocada 99
8.8 T7 - Teste de performance para execução de funções computacionalmente pesadas 103
8.9 Comparação de performance entre uma instalação local da *OpenWhisk* e a *OpenWhisk* da IBM 105
8.10 Conclusões finais da execução do *benchmark* 108

9 Demonstração da utilização da plataforma *serverless* no contexto do MobiWise 111

10 Conclusão 117

10.1 Trabalho futuro 118

Referências 119

Acrónimos

AWS Amazon Web Services. xi, 17, 18, 27, 29, 30, 72, 90

EBS Amazon Elastic Block Store. 17

EC2 Amazon Elastic Compute Cloud. 17

FaaS Function-as-a-Service. 18, 24, 25, 40, 42, 117

IaaS Infrastructure-as-a-Service. 16, 17

IDE Ambiente de Desenvolvimento Integrado. 28

IoT Internet of Things. 2, 25, 31, 38

NIST Instituto Nacional de Standards e Tecnologia do Departamento de Comércio dos Estados Unidos. 13

PaaS Platform-as-a-Service. 16–18, 24

SaaS Software-as-a-Service. 16, 18, 24

USD Dólar dos Estados Unidos. 30, 34

VM Máquina Virtual. xi, 19–21

VMs Máquinas Virtuais. xiii, 18, 20, 31, 39

Esta página foi propositadamente deixada em branco.

Lista de Figuras

2.1 Diagrama de *Gantt* do trabalho planeado para o primeiro semestre 6
2.2 Diagrama de *Gantt* do trabalho planeado, no início do estágio, para o segundo semestre 7
2.3 Diagrama de *Gantt* do trabalho planeado no início do segundo semestre para o mesmo 8
2.4 Diagrama de *Gantt* do trabalho executado no primeiro semestre 10
2.5 Diagrama de *Gantt* do trabalho executado no segundo semestre 11

3.1 Arquitetura por camadas de computação em *cloud* e modelos de serviços (Adaptado de [1]) 17
3.2 Arquitetura de virtualização baseada em Máquina Virtual (VM) (Adaptado de [2]) 19
3.3 Arquitetura de virtualização baseada em *containers* (Adaptado de [2]) . . . 19
3.4 Arquitetura do *Docker* (Retirada de [3]) 21
3.5 Fluxo de funcionamento do *Docker* (Retirada de [4]) 22
3.6 Arquitetura do *Kubernetes* (Retirada de [5]) 23
3.7 Representação visual de um exemplo de *workflow* 26

4.1 Processo de execução de uma função na *Amazon Web Services (AWS) Lambda* (Adaptado de [6]) 29
4.2 *Warm* e *Cold containers* na *AWS Lambda* (Adaptado de [7]) 30
4.3 Arquitetura de alto nível da *OpenWhisk* [8] 36
4.4 Arquitetura e processo de execução de uma função na *OpenWhisk* (Adaptado de [8]) 37
4.5 *Workflow serverless* criado recorrendo à *OpenWhisk* e à *IBM Composer* . . 39

6.1 Diagrama de contexto do sistema de *Benchmarking* a implementar 68
6.2 Diagrama de *container* do sistema de *Benchmarking* a implementar 68
6.3 Diagrama de componentes do sistema de *Benchmarking* a implementar . . . 69

7.1 Exemplo de função escrita em *Node.js* para *deploy* na *AWS Lambda* 76
7.2 Exemplo de ficheiro "serverless.yml", utilizado para definição de um serviço 76
7.3 Criação do grupo de *threads* do plano de teste padrão do *JMeter* 79
7.4 Criação do pedido HTTP do plano de teste padrão do *JMeter* 79
7.5 Resultado da execução do comando de ajuda no sistema de execução de *benchmark* 83

8.1 Resultado da execução do teste de latência T1 por plataforma de computação *serverless*, durante 20 segundos: Média (círculo a laranja), desvio padrão (retângulo cinzento) e valores máximos e mínimos (barra a vermelho) 91
8.2 Resultado da execução do teste T2: *Throughput* por segundo e latência média por invocação em função do número de pedidos concorrentes efetuados durante 20 segundos 92

8.3 Resultado da execução do teste T3: Latência da invocação em função do tempo ocorrido desde a ultima invocação 94
8.4 Resultado da execução do teste T4: Latência média em função do tamanho do *payload* 96
8.5 Resultado da execução do teste T5 na *AWS Lambda*, durante 20 segundos: Média (círculo a laranja), desvio padrão (retângulo cinzento) e valores máximos e mínimos (barra a vermelho) de latência em função da linguagem de programação 97
8.6 Resultado da execução do teste T5 na *OpenWhisk*: Média (círculo a laranja), desvio padrão (retângulo cinzento) e valores máximos e mínimos (barra a vermelho) de latência em função da linguagem de programação . . 98
8.7 Resultado da execução do teste T6 na *OpenWhisk*, durante 10 segundos, utilizando uma função leve computacionalmente: Latência por invocação para cada valor de memória alocada em MB 100
8.8 Resultado da execução do teste T6 na *AWS Lambda*, durante 10 segundos, utilizando uma função leve computacionalmente: Latência por invocação para cada valor de memória alocada em MB 100
8.9 Resultado da execução do teste T6 na *Google Cloud Functions*, durante 10 segundos, utilizando uma função leve computacionalmente: Latência por invocação para cada valor de memória alocada em MB 101
8.10 Resultado da execução do teste T6 na *OpenWhisk*, durante 10 segundos, utilizando a função de Fibonacci com n=35: Latência por invocação para cada valor de memória alocada em MB 101
8.11 Resultado da execução do teste T6 na *AWS Lambda*, durante 10 segundos, utilizando a função de Fibonacci com n=35: Latência por invocação para cada valor de memória alocada em MB 102
8.12 Resultado da execução do teste T6 na *Google Cloud Functions*, durante 10 segundos, utilizando a função de Fibonacci com n=35: Latência por invocação para cada valor de memória alocada em MB 103
8.13 Resultado da execução do teste T7: Média da latência por invocação em função do N da sequência de *Fibonacci* utilizado 104
8.14 *Containers* ativos de uma função e pré alocados para a linguagem de programação na instalação local da *OpenWhisk* 105
8.15 Resultado da execução do teste T1 na *OpenWhisk* da IBM e na local, durante 20 segundos: Média (círculo a laranja), desvio padrão (retângulo cinzento) e valores máximos e mínimos (barra a vermelho) de latência . . . 106
8.16 Resultado da execução do teste T2 na instalação local da *OpenWhisk*: *Throughput* por segundo e latência média por invocação em função do número de pedidos concorrentes efetuados durante 20 segundos 107
8.17 Resultado da execução do teste T2 na *OpenWhisk* da IBM: *Throughput* por segundo e latência média por invocação em função do número de pedidos concorrentes efetuados durante 20 segundos 107

9.1 Exemplo de rede rodoviária criada no *software* SUMO 112
9.2 *Workflow serveless* do serviço, criado com a utilização da *Openwhisk* e da sua ferramenta de *Workflows Composer* 113
9.3 Exemplo de função *serverless* da *OpenWhisk* utilizada no serviço 114
9.4 Execução da simulação com as rotas otimizadas pelo serviço, de acordo com a distância (à esquerda) e com o tempo (à direita). Veículos amarelos possuem percursos fixos e os vermelhos seguem as rotas otimizadas pelo serviço 114

9.5 Comparação dos dados de distância, tempo e poluentes obtidos das simulações das rotas geradas pela otimização em função da distância total e do tempo total gasto em viagem . . . 115

Lista de Tabelas

3.1 Comparação de virtualização através de Máquinas Virtuais (VMs) e de *containers* [9] . . . 20

4.1 Preço da *AWS Lambda* por nível de memória . . . 30
4.2 Preço da *Azure Functions* por nível de memória . . . 32
4.3 Preço de computação por 100ms da *Google Cloud Functions* por nível de memória e CPU . . . 34
4.4 Preço da *IBM Cloud Functions* por nível de memória . . . 38

5.1 Comparação de plataformas *Serverless* . . . 50

6.1 Resumo dos estudos de performance e *benchmarkings* apresentados . . . 61
6.2 Definição do teste de *overhead* das plataformas *serverless* . . . 63
6.3 Definição do teste de carga concorrente . . . 63
6.4 Definição do teste de reutilização de *containers* . . . 64
6.5 Definição do teste do impacto do tamanho do *payload* . . . 64
6.6 Definição do teste de impacto da linguagem de programação na performance 65
6.7 Definição do teste de performance em função da memória alocada . . . 65
6.8 Definição do teste de performance para execução de funções computacionalmente pesadas . . . 66

8.1 Resultado da execução do teste T1 por plataforma de computação *serverless*, durante 20 segundos . . . 91
8.2 Resultado da execução do teste T2 por plataforma de computação *serverless*, durante 20 segundos com 15 pedidos concorrentes . . . 93
8.3 Resultado da execução do teste T5 na *AWS Lambda*, durante 20 segundos . 97
8.4 Resultado da execução do teste T5 na *OpenWhisk*, durante 20 segundos . . 98
8.5 Resultado da execução do teste T1 na *OpenWhisk* da IBM e na local, durante 20 segundos . . . 105

Esta página foi propositadamente deixada em branco.

Capítulo 1

Introdução

O presente documento tem como objetivo descrever o trabalho realizado durante a Dissertação/Estágio do Mestrado em Engenharia Informática, do Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra, relativo ao modelo plurianual do ano letivo 2017/2018 e 2018/2019.

O estágio foi desenvolvido sob orientação do Professor Doutor Paulo Rupino da Cunha e do Professor Doutor Filipe Araújo, decorrendo no Departamento de Engenharia Informática.

Nas secções seguintes serão apresentados o enquadramento e objetivos do estágio, as contribuições dadas através do trabalho desenvolvido e a estrutura do presente documento.

1.1 Enquadramento e objetivos

A computação *serverless*, também conhecida por *Function-as-a-Service*, é um paradigma altamente escalável de computação em *cloud* para o desenvolvimento e execução de aplicações em resposta a eventos, em que os recursos computacionais são abstraídos. Seguindo a *AWS Lambda*, que foi a pioneira na oferta deste tipo de serviço, várias outras plataformas deste tipo surgiram.

O objetivo original da proposta do presente estágio era o *design* e implementação de uma plataforma de *Function-as-a-Service*, e de *workflow serverless*, que funcionasse sobre *Openstack*. No entanto, no tempo decorrido entre a proposta e o arranque dos trabalhos, vários projetos *open source* foram iniciados mundialmente, deixando de fazer sentido prosseguir o objetivo inicial. Neste contexto, tentou-se perceber se as plataformas em causa eram suficientemente maduras para aplicação em contextos reais e quais as suas lacunas. Para atingir este novo objetivo orientou-se o estudo para sistemas de teste e *benchmarking*, os quais se revelaram lacunares, tendo-se decidido desenhar e propor um conjunto de testes de *benchmarking* para utilização genérica que ultrapasse essas limitações e uma arquitetura de um sistema de automatização da execução do *benchmark*. Num segundo momento procedeu-se a implementação do sistema seguindo a arquitetura proposta. A ferramenta

implementada foi utilizado no *benchmarking* de algumas plataformas de computação *serverless*, tendo sido realizada uma análise comparativa relativamente aos resultados obtidos em cada um dos testes.

Aplicou-se ainda uma das plataformas de computação e *workflows serverless open source* existentes, a *OpenWhisk*, a uma das tarefas do projeto *Mobiwise*[1,2], através da demonstração da criação de um serviço, que fornece uma solução de mobilidade baseada na otimização de rotas de um fluxo de veículos, recorrendo-se a este tipo de plataformas.

O projeto *MobiWise* tem como objetivo principal a construção de uma plataforma 5G, que engloba uma infraestrutura preenchida com sensores, pessoas e veículos, e uma plataforma de serviços de suporte para uma implantação real de Internet of Things (IoT) numa cidade inteligente para melhorar a mobilidade, quer para viajantes habituais, quer para turistas, através da análise de múltiplos indicadores, obtidos de uma análise de dados massivos, de forma a propor rotas mais ecológicas . Alguns dos serviços incluirão o eco-encaminhamento urbano, o qual requer a obtenção de dados de *smartphones*, sensores e transportes para escolher os melhores caminhos na cidade. O MobiWise irá estudar e integrar diferentes tipos de tecnologias e redes numa mesma infraestrutura com suporte a mobilidade, serviços e aplicações disponibilizadas através de uma *cloud*.

1.2 Contribuições

Com o trabalho realizado durante o estágio e apresentado neste relatório, considera-se ter dado as seguintes contribuições:

- Estudo e apresentação de algumas das plataformas de computação *serverless*.
- Proposta de parâmetros para a comparação de alto nível, em termos de funcionalidades e características, das plataformas de computação *serverless*.
- Estudo do estado da arte em métodos e sistemas para o teste de plataformas *serverless*.
- Proposta de um conjunto de testes para o *benchmark* de plataformas de computação *serverless*.
- Desenho e implementação de uma ferramenta para automatização da execução do conjunto de testes.
- Aplicação da plataforma *serverless OpenWhisk* e da sua ferramenta de *workflows Composer* a uma demonstração de criação de serviço de mobilidade para *smart cities*, no contexto do projeto MobiWise.

[1] `http://mobiwise.av.it.pt/`
[2] `https://www.it.pt/Projects/Index/4493`

1.3 Estrutura do documento

Neste capítulo foi realizada a apresentação do enquadramento e os objetivos do estágio. É ainda feita uma breve introdução ao projeto *MobiWise*.

No capítulo dois, descreve-se o planeamento de trabalho do estágio e a sua calendarização, comparando-se a que foi idealizada inicialmente, com a que foi realmente executada.

No terceiro capítulo são apresentados e descritos alguns conceitos e tecnologias de referência ao estágio. Numa primeira fase é exposta a computação em *cloud*, referindo-se a sua definição, as suas características, os modelos de *deploy*, a sua arquitetura e os modelos de *delivery* de serviços. De seguida, é apresentada a tecnologia de *containers* e os seus principais *softwares*. Por fim, é abordada a computação *serverless*, que é o tema central do estágio, e o modelo de *workflows serverless*.

O quarto capítulo é dedicado à análise de algumas das plataformas de computação *serverless* existentes, sendo estas apresentadas de forma detalhada.

No capítulo cinco, é proposto um conjunto de parâmetros que é utilizado na comparação, também aqui apresentada, em termos de funcionalidades e características das várias plataformas *serverless*, abordadas no capítulo quatro.

No capítulo seis procedeu-se à exposição de alguns estudos e *benchmarkings* de plataformas *serverless* existentes na literatura, referindo-se as suas limitações. Com base nas lacunas encontradas é proposto um novo *benchmarking*, sendo definidos os testes que o constituem, assim como uma arquitetura de uma aplicação de automatização do processo de execução do conjunto de testes.

No sétimo capítulo são apresentados alguns detalhes do processo de implementação do sistema de execução do *benchmkar*, sendo descritos os passos necessários para a sua configuração e utilização.

No capítulo 8 são descritos os testes do *benchmark* efetuados, sendo descrito como foram realizados e apresentando-se e analisando-se os respetivos resultados obtidos.

No nono capítulo é abordada a demonstração realizada no contexto do *MobiWise*, sendo apresentado o serviço criado recorrendo-se à *OpenWhisk*.

Por fim, no capítulo dez são apresentadas as conclusões resultantes do trabalho realizado durante o estágio e sugeridas algumas direções de trabalho futuro.

Esta página foi propositadamente deixada em branco.

Capítulo 2

Planeamento

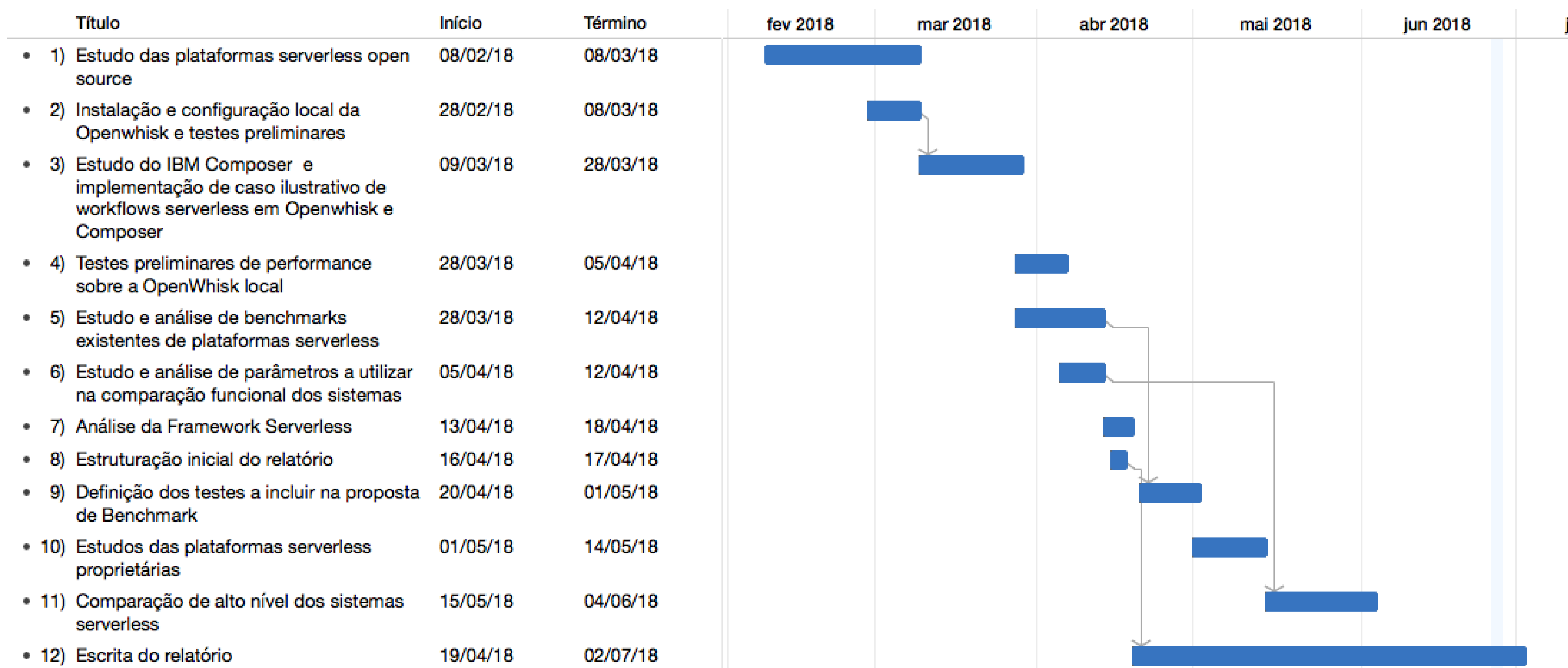
Neste capítulo será apresentado o planeamento do trabalho que irá ser executado ao longo deste estágio. Será também efetuada uma comparação do que foi proposto inicialmente com o que foi realmente executado.

2.1 Cronograma planeado

O planeamento do estágio está dividido em dois blocos, que são os dois semestres que compreendem a sua duração. Neste primeiro semestre, o trabalho foi realizado a tempo parcial (16 horas semanais), tendo-se iniciado a 8 de fevereiro de 2018 e findado a 4 de junho de 2017, sendo esta data estendida, para elaboração do presente documento, até 2 de Julho de 2018. O segundo semestre foi realizado a tempo integral (40 horas semanais) e durou de 5 de Setembro de 2018 a 23 de Janeiro de 2019.

Na figura 2.1 é apresentado o diagrama de *Gantt* da calendarização das tarefas propostas inicialmente, isto é, aquelas que foram planeadas no início dos trabalhos do estágio. De destacar como principais atividades realizadas a identificação e estudo das plataformas *serverless* existentes, a comparação funcional destas, a identificação de *benchmarks* existentes, a proposta de um *benchmark*, para sistemas *serverless*, e definição de uma arquitetura de um sistema de automatização de execução dos testes que compõem o *benchmark*.

O cronograma inicialmente planeado para o segundo semestre é exposto no diagrama de *Gantt* da figura 2.2, e dele faziam parte essencialmente o desenvolvimento do sistema de *benchmark*, a execução do mesmo nas diversas plataformas *serverless*, a análise de resultados obtidos e, por fim, a aplicação de umas das plataformas mais maduras a uma das tarefas do projeto *MobiWise*. No início do semestre devido à necessidade de se realizar uma demonstração da utilização da plataforma de computação *serverless*, numa das tarefas do projeto *MobiWise*, para ser apresentada no dia 19 de Outubro de 2018, procedeu-se a redefinição do planeamento para este semestre, sendo o respetivo diagrama de *Gantt* apresentado na figura 2.3.



	Título	Início	Término
1)	Estudo das plataformas serverless open source	08/02/18	08/03/18
2)	Instalação e configuração local da Openwhisk e testes preliminares	28/02/18	08/03/18
3)	Estudo do IBM Composer e implementação de caso ilustrativo de workflows serverless em Openwhisk e Composer	09/03/18	28/03/18
4)	Testes preliminares de performance sobre a OpenWhisk local	28/03/18	05/04/18
5)	Estudo e análise de benchmarks existentes de plataformas serverless	28/03/18	12/04/18
6)	Estudo e análise de parâmetros a utilizar na comparação funcional dos sistemas	05/04/18	12/04/18
7)	Análise da Framework Serverless	13/04/18	18/04/18
8)	Estruturação inicial do relatório	16/04/18	17/04/18
9)	Definição dos testes a incluir na proposta de Benchmark	20/04/18	01/05/18
10)	Estudos das plataformas serverless proprietárias	01/05/18	14/05/18
11)	Comparação de alto nível dos sistemas serverless	15/05/18	04/06/18
12)	Escrita do relatório	19/04/18	02/07/18

Figura 2.1: Diagrama de *Gantt* do trabalho planeado para o primeiro semestre

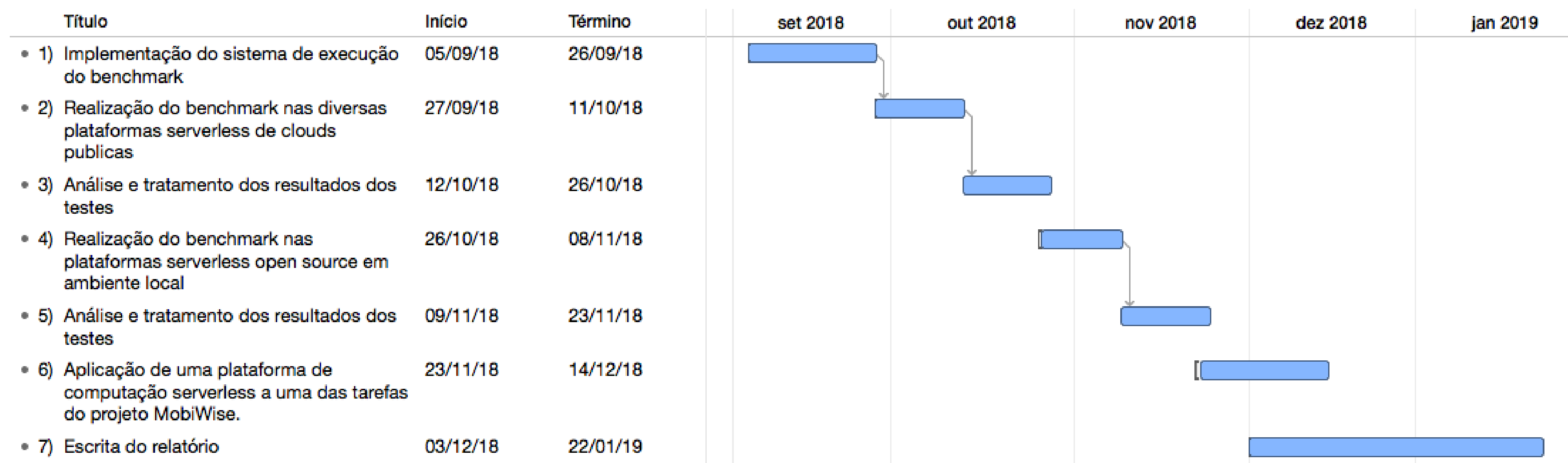


Figura 2.2: Diagrama de *Gantt* do trabalho planeado, no início do estágio, para o segundo semestre

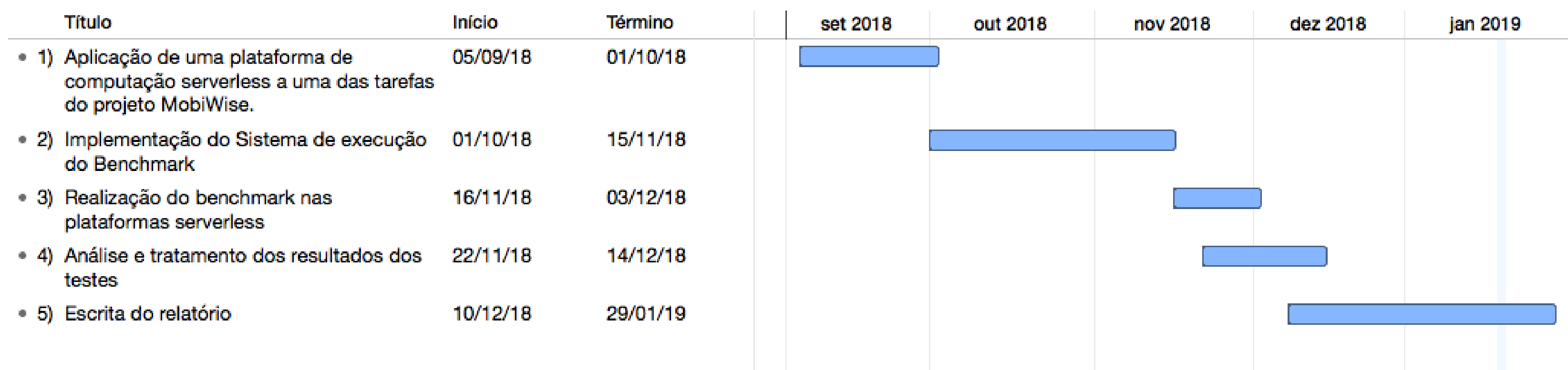


Figura 2.3: Diagrama de *Gantt* do trabalho planeado no início do segundo semestre para o mesmo

2.2 Cronograma executado

Nesta secção será apresentada a execução real das tarefas, fazendo-se uma comparação com o que foi inicialmente planeado e estimado.

No diagrama de *Gantt* da figura 2.4, é apresentado o cronograma real de execução das tarefas do primeiro semestre. Pela sua análise, é possível verificar, que todas as atividades planeadas inicialmente foram cumpridas. Foram adicionadas duas novas tarefas, que não tinham sido consideradas inicialmente, nomeadamente, a definição da arquitetura do sistema de execução de *benchmark* e o tratamento e análise dos resultados dos testes preliminares (marcadas a laranja na figura). Este facto, levou a um atraso de uma semana no início da execução do estudo das plataformas *serverless* proprietárias, da comparação de alto nível dos sistemas *serverless* e da escrita do relatório. Duas tarefas levaram mais tempo na sua realização do que o que tinha sido estimado inicialmente. O estudo das plataformas *serverless* proprietárias levou mais 3 dias para ser realizado (marcado a vermelho no diagrama de *Gantt*), assim como a comparação de alto nível destes sistemas. Por fim, a escrita do relatório foi feita em menos tempo do que o planeado inicialmente, pois, como já foi visto, o inicio do seu desenvolvimento teve um atraso de uma semana.

O cronograma real do segundo semestre pode ser visualizado no diagrama de *Gantt* da figura 2.5. É de salientar que as tarefas propostas no início do semestre foram cumpridas, apesar da existência de atrasos significativos em algumas das tarefas, que podem ser vistos a vermelho no diagrama de *Gantt*. A tarefa de aplicação de uma plataforma de computação *serverless* a uma das tarefas do projeto *MobiWise* teve um atraso na sua realização de cerca de 18 dias, devido essencialmente à demora na definição do caso de uso a utilizar e a complicações de interligação com outros membros do projeto, nomeadamente nos responsáveis pela implementação do algoritmo de otimização. Este atraso levou a que todas as tarefas posteriores começassem também com algum atraso. A implementação do sistema de execução do *Benchmark* começou 18 dias depois do previsto, tendo demorado cinco dias a mais que o previsto, que são essencialmente explicados por algumas dificuldades na configuração dos provedores na *Serverless Framework*. A realização do *benchmark* nas plataformas *serverless*, devido ao atraso das tarefas anteriores, iniciou-se com 23 dias de atraso tendo demorado mais 3 dias que o planeado. A tarefa de análise e tratamento dos resultados dos testes iniciou-se com 20 dias de atraso, contudo demorou menos 4 dias a estar completa em relação ao que tinha sido planeado. Por fim, a escrita do relatório teve de ser realizada em menos tempo do que o planeado inicialmente, devido ao atraso no inicio da sua escrita de cerca de 15 dias.

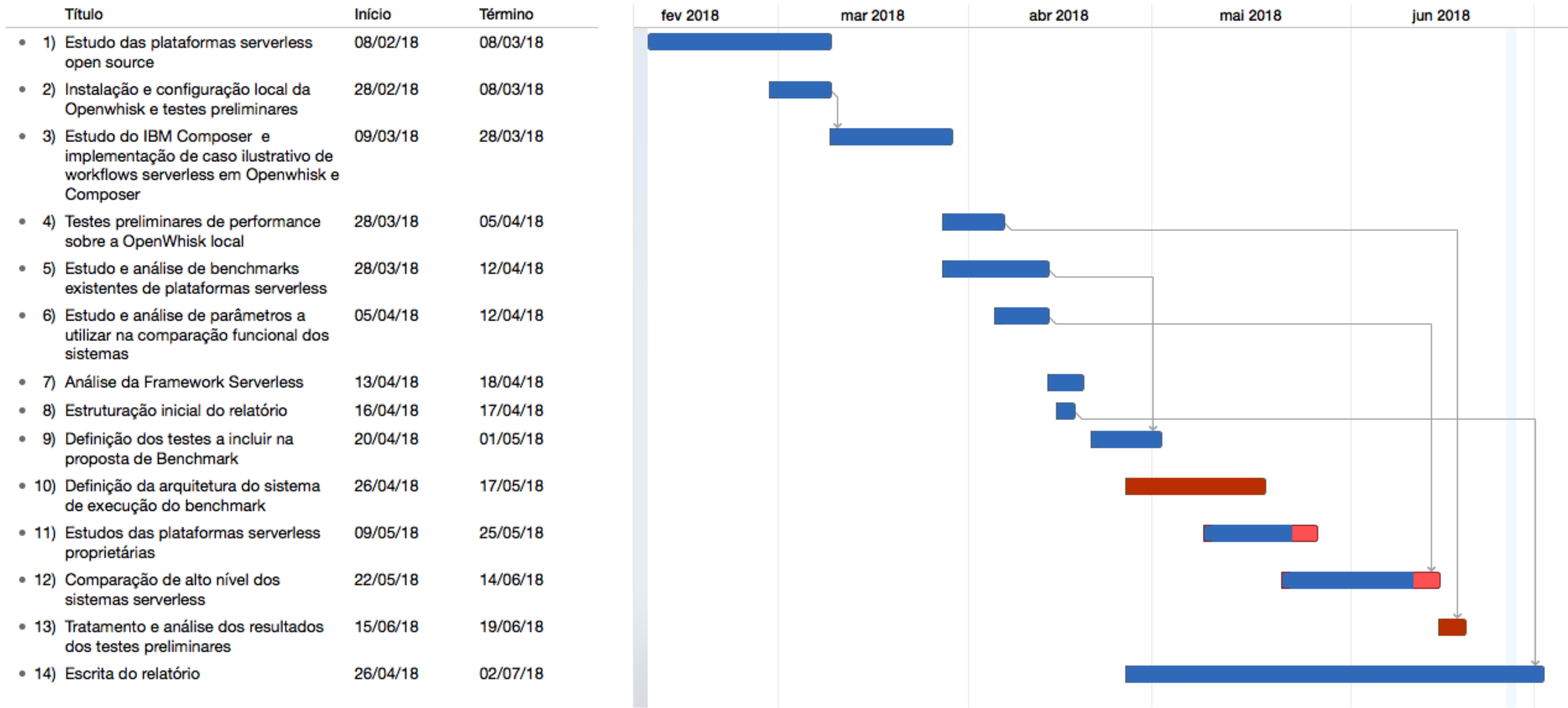


Figura 2.4: Diagrama de *Gantt* do trabalho executado no primeiro semestre

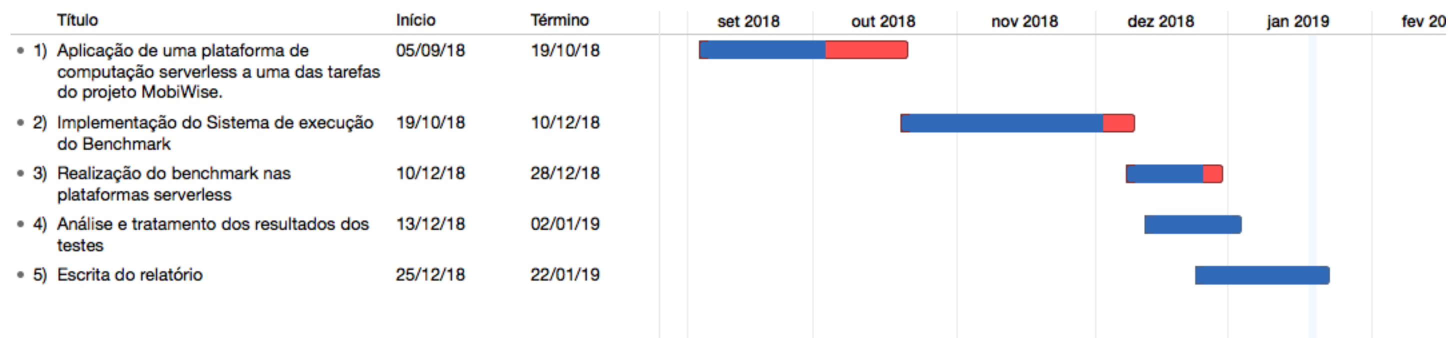


Figura 2.5: Diagrama de *Gantt* do trabalho executado no segundo semestre

Esta página foi propositadamente deixada em branco.

Capítulo 3

Conceitos e tecnologias base

Computação *serverless* é um paradigma de computação em *cloud*, para o desenvolvimento de aplicações, em que os utilizadores podem executar o seu código sem que se tenham de preocupar com o aprovisionamento e configuração dos servidores e da infraestrutura que a suporta. Neste capítulo são definidos alguns conceitos e apresentadas algumas tecnologias que permitiram o surgimento da computação *serverless*. Numa primeira fase é dada uma visão geral sobre a *cloud* e sobre os modelos de serviço existentes. Numa segunda fase é abordada a tecnologia de *containers* que permitiu grandes avanços na área da virtualização e que serve de base para a computação *serverless*, sendo de seguida apresentado este mesmo conceito com algumas das duas características. Por fim é apresentado o modelo de *workflow serverless*, usado para a criação de aplicações *serverless*, recorrendo à orquestração de funções e outras componentes, na *cloud*.

3.1 Computação em *Cloud*

Computação em *cloud*, ou simplesmente *cloud*, pode ser entendida como sendo uma tecnologia, ou um paradigma, que permite o fornecimento e utilização de serviços, de recursos, de aplicações, de programas (*software*) e de outros recursos tecnológicos, que podem ser próprios ou de terceiros, através da rede (*internet/cloud*), sendo cobrado um valor pelos recursos utilizados. Na literatura é possível encontrar várias definições para estes termos, onde de seguida se apresentarão algumas delas, por representarem várias vertentes.

Zhang *et al.*, em [1], definem computação em *cloud* como sendo um paradigma para o alojamento, gestão e distribuição de serviços e recursos tecnológicos escaláveis através da Internet, que segue um modelo de negócio em que o provedor de *cloud*, ou fornecedor de serviço, disponibiliza recursos sob procura do cliente à medida das suas necessidades mediante pagamento.

De acordo com o Instituto Nacional de Standards e Tecnologia do Departamento de Comércio dos Estados Unidos (NIST), em [10], a computação em *cloud* é um modelo que permite o acesso ubíquo, conveniente, sob procura e através da rede a um conjunto

partilhado de recursos computacionais configuráveis (por exemplo, redes, servidores, armazenamento, aplicações e serviços) que podem ser provisionados, disponibilizados e libertados com o mínimo de esforço de gestão ou iteração com o provedor do serviço.

Para Kaur e Chana [11], computação em *cloud* é descrita como um paradigma de computação distribuída que permite que aplicações virtualizadas, software, plataformas, computação e armazenamento sejam disponibilizados ao cliente através do uso de serviços auto-gerenciados que são distribuídos pela Internet e que são pagos de acordo com a utilização. Armbrust *et al.* [12], referem que, do ponto de vista do aprovisionamento de *hardware* e de preço, a computação em *cloud* apresenta a aparência de possuir recursos computacionais infinitos disponíveis sob procura, capazes de se adaptarem rapidamente a picos de carga, eliminando a necessidade de os utilizadores planearem com antecedência a capacidade que possam vir a necessitar. A *cloud* permite também que os utilizadores adaptem os recursos que utilizam à medida que as suas necessidades vão aumentando. A capacidade de se pagar pelo que se utiliza a curto prazo permite a diminuição de despesas, pois os recursos podem ser libertados quando já não são necessários, sendo desta forma pagos na medida em que foram utilizados, como por exemplo, pelo tempo de utilização de processador.

3.1.1 Características de uma *cloud*

Há algumas características importantes que um sistema de computação em *cloud* devem incluir e apresentar [10] [1]:

- Disponibilização de recursos e serviços sob procura: O cliente deve ser capaz de obter e gerir os recursos de que necessita de forma unilateral e automaticamente.
- *Pooling* de recursos: A *cloud* oferece uma *pool* de recursos computacionais que podem ser dinamicamente atribuídos a múltiplos clientes.
- Ubiquidade: Os recursos devem poder ser acedidos de forma fácil e rápida de qualquer lugar que tenha acesso a Internet.
- Escalabilidade: A *cloud* deve manter o desempenho e performance à medida que a carga sobre ela aumenta.
- Elasticidade: Os recursos computacionais devem ser aumentados ou diminuídos de forma automática de acordo com a carga a que estes estão sujeitos, ou de acordo das exigências do utilizador.
- Monitorização e controlo: A *cloud* deve permitir que os utilizadores tenham acesso a um sistema de controlo que permita monitorizar e gerir os recursos e serviços computacionais que estão ao seu dispor.
- Pagamento pelo que se utiliza: A *cloud* aplica um modelo de pagamento em que os utilizadores são cobrados pelos recursos que utilizam.

3.1.2 Modelos de *deploy* de *cloud*

As *clouds* podem ser implementadas e classificadas de acordo com a forma como são *deployed* pelos seus provedores para que sejam aprovisionadas aos seus clientes, que escolhe o modelo de acordo com os seus requisitos de negócio, operacionais e técnicos [13]. Tipicamente, as *clouds* são classificadas em *cloud* pública, *cloud* privada, *cloud* comunitária ou *cloud* híbrida [10].

- *Cloud* pública: Modelo em que os provedores de serviços disponibilizam os seus recursos como serviços ao público em geral. A infraestrutura é detida, gerida e operada por uma organização comercial, académica, governamental ou por uma combinação destas. Normalmente, neste tipo de *clouds* os utilizadores pagam pelos recursos que utilizam. O custo inicial para os clientes é nulo pois a infraestrutura é do fornecedor assim como o risco da sua administração e manutenção. Contudo, neste tipo de *cloud*, o controlo por parte do cliente sobre a infraestrutura pode ser quase nulo e podem existir mais riscos relacionados com segurança e privacidade [10] [1].

- *Cloud* privada: *Cloud* em que a infraestrutura é disponibilizada para uso exclusivo de uma organização tendo esta um controlo total sobre a sua performance, fiabilidade e segurança. Normalmente é detida, gerida e operada pela organização e está implementada nas suas instalações. Este tipo de *cloud* apresenta um grande custo inicial, pois é necessário comprar, instalar, configurar e manter toda a infraestrutura e todos os custos e riscos associados [10] [1].

- *Cloud* comunitária: *Cloud* em que a infraestrutura é disponibilizada para uso exclusivo por parte de uma comunidade de clientes ou organizações com objetivos e preocupações comuns. Em comparação com a *cloud* privada, esta possui as mesmas vantagens, sendo que os custos são repartidos pelas partes que fazem parte da comunidade. A gestão da infraestrutura pode ser realizada por uma ou várias partes da comunidade ou por uma entidade exterior e pode estar implantada dentro de uma das organizações ou no exterior [10] [1].

- *Cloud* híbrida: A infraestrutura deste tipo de *cloud* é composta por uma mistura de pelo menos duas das apresentadas anteriormente. Geralmente, uma parte da infraestrutura é fornecida por uma *cloud* privada e a outra parte por uma pública. Este modelo permite ter um maior controlo e segurança sobre a *cloud* quando comparado com o modelo público, e mantém a capacidade de expandir os serviços sob procura mantidos pela infraestrutura com maior capacidade [10] [1].

3.1.3 Arquitetura de *cloud*

Uma infraestrutura de *cloud* possui uma arquitetura modular que permita o suporte a um elevado número de recursos mantendo o *overhead* de manutenção e gestão baixo. A arquitetura pode ser dividida em quatro camadas, como visto na figura 3.1: A camada de

hardware, a camada de infraestrutura, a camada de plataforma e a camada de aplicação [1].

A camada de *hardware* é geralmente implementada nos *data centers* e é responsável por gerir todo o *hardware* sobre o qual a *cloud* assenta, nomeadamente os servidores, os dispositivos de rede e todos os outros componentes necessários. A camada de infraestrutura utiliza os recursos físicos da camada *hardware* para que, com a utilização de tecnologias de virtualização, sejam criados um conjunto de recursos de computação, máquinas virtuais, de armazenamento e de comunicação. A camada de plataforma assenta sobre a camada de infraestrutura, e consiste essencialmente em *frameworks* aplicacionais que facilitam a forma como é feito o *deploy* de aplicações nas máquinas virtuais, como por exemplo oferecendo API's que permitem implementar e usar de forma fácil uma base de dados, ou lógica de negócio de uma aplicação *web*. A camada de aplicação consiste nas aplicações *cloud* que apresentam um grande nível de escalabilidade automática de forma a atingirem um grande nível de performance e disponibilidade, como por exemplo, o *Facebook*, o *Netflix* e o *Youtube* [1].

Cada uma das camadas apresentadas pode ser implementada como serviço para as que lhe são superiores visto que as últimas assentam e são implementadas sob as camadas inferiores.

3.1.4 Modelo de *delivery* de serviços da *cloud*

Os recursos e serviços oferecidos sob procura pelas *clouds* podem ser classificados essencialmente segundo três modelos: *Infrastructure-as-a-Service (IaaS)*, *Platform-as-a-Service (PaaS)* e *Software-as-a-Service (SaaS)* [10] [1] [12]. Na figura 3.1 é possível vê-los assim como as camadas de arquitetura que fazem parte de cada um deles. Os modelos são distinguidos essencialmente pelo tipo de serviço que oferecem e pelo nível de abstração e de controlo que oferecem sobre os recursos que disponibilizam. Quanto maior for o nível de controlo menor será nível de abstração e vice-versa [13]. Enquanto que *IaaS* apresenta um maior nível de controlo sobre a infraestrutura e menor nível de abstração, *SaaS* apresenta um menor nível de controlo e um maior nível de abstração e *PaaS* um nível intermédio para ambas as características.

O modelo *IaaS* consiste na disponibilização de recursos, tais como computação, armazenamento e redes de comunicação, geralmente sobre a forma de máquinas virtuais. Neste tipo de modelo o cliente pode construir sistemas complexos e fazer o *deploy* e execução das suas aplicações, sendo responsável por gerir a arquitetura que pretende que o seu sistema apresente, como por exemplo a replicação de máquinas para tolerância a falhas e arquitetura de escalabilidade. Como se pode verificar, o nível de controlo é alto visto, que é o cliente que instala, configura, mantém e gere a infraestrutura que demandou, as características computacionais que pretende para as máquinas virtuais, o sistema operativo das máquinas, o *software* que necessita, como por exemplo linguagens de programação, servidores *web* e bases de dados, e as aplicações que pretende executar. Neste tipo de

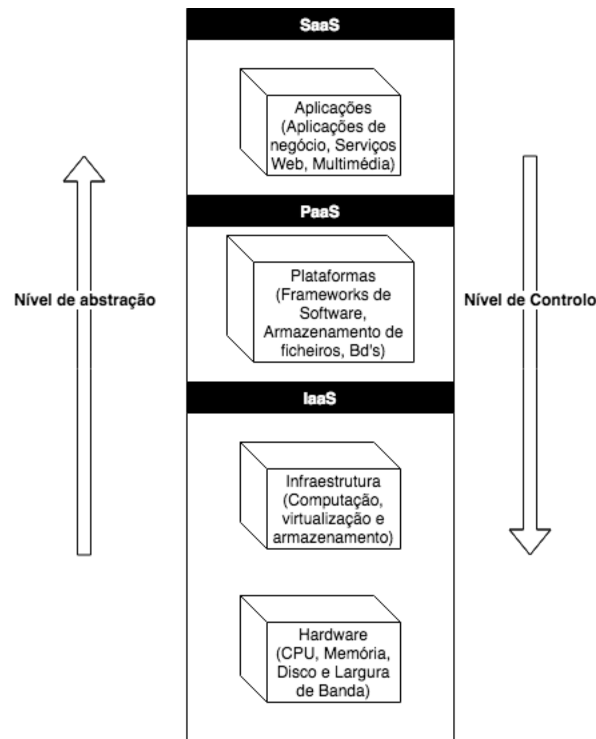


Figura 3.1: Arquitetura por camadas de computação em *cloud* e modelos de serviços (Adaptado de [1])

modelo os clientes são cobrados pelo tempo de utilização das máquinas virtuais e pela capacidade computacional destas, pelo espaço de armazenamento que estão a ocupar, e pelo volume de dados que é transferido pela rede. Alguns dos provedores de IaaS são a *Amazon Web Services (AWS)* essencialmente com o *Amazon Elastic Compute Cloud (EC2)*[1] para computação e *Amazon Elastic Block Store (EBS)*[2] para armazenamento, a *Google Cloud* com o serviço *Compute Engine*[3] e a *OpenStack* com o *Nova*[4].

O modelo de *PaaS* refere-se à disponibilização de recursos prontos a utilizar, como bases de dados e sistemas de armazenamento, e outras plataformas computacionais escaláveis, tais como ferramentas de desenvolvimento e *deploy* em que os ambientes de desenvolvimento já estão instalados, que já incluem sistema operativo, ambientes de execução de linguagens de programação, servidores *web*, bibliotecas, serviços, ferramentas e *frameworks* de *deploy* de aplicações que permitem ao utilizador fazer *deploy* na *cloud* das suas aplicações. *PaaS* fornece assim uma camada de abstração relativamente a *IaaS* para o *deploy* de aplicações e serviços de uma forma mais fácil e rápida. O provedor de *PaaS* é responsável por disponibilizar e gerir toda a infraestrutura que suporta estes ambientes de desenvolvimento e por manter os mesmos. O cliente de *PaaS*, para além de gerir as aplicações de que fez *deploy*, pode definir e alterar algumas configurações do ambiente de desenvolvimento, como por exemplo, o seu tipo, as linguagens de programação, políticas de escalabilidade, bases de dados e poder computacional das máquinas. Este modelo comparado com *IaaS*

[1] `https://aws.amazon.com/pt/ec2/`
[2] `https://aws.amazon.com/pt/ebs/`
[3] `https://cloud.google.com/compute/?hl=pt-PT`
[4] `https://wiki.openstack.org/wiki/Nova`

apresenta uma maior abstração relativamente a infraestrutura que o suporta e sob a qual o cliente ainda tem ainda algum nível de controlo. Algumas das *PaaS* mais conhecidas são: *AWS Elastic Beanstalk*[5], *Microsoft Azure App Service*[6], *Heroku*[7] e *Google App Engine*[8].

Finalmente, o modelo *SaaS* disponibiliza aos clientes, sob procura, aplicações, que correm na infraestrutura do provedor. Estas podem ser acedidas e utilizadas através da Internet de uma forma simples, quer através de um *browser*, quer, por exemplo, através de aplicações de *smartphones* e *smart tv's*. O utilizador não tem qualquer tipo de controlo sobre a infraestrutura em que assentam estão aplicações, já que é provisionada, gerida, e mantida totalmente pelo provedor. O utilizador pode não ter qualquer nível de controlo sobre as aplicações. Apenas as utiliza, salvo algumas exceções, em que poderá ser possível definir algumas configurações. Normalmente, este modelo é cobrado sob a forma de uma taxa de subscrição ou utilização, em que o utilizador paga pela utilização das aplicações, ao contrario dos modelos anteriores em que eram cobrados os recursos computacionais utilizados. Alguns exemplos de *SaaS* são o *Facebook*[9], o *Gmail*[10], a *Microsoft Office 365*[11], os *Google Docs*[12], a Salesforce[13] e a *Dropbox*[14].

Recentemente surgiu um novo modelo de serviço de *cloud* denominado de *Function-as-a-Service (FaaS)*, também conhecido por computação *Serverless*, e que irá ser abordado na secção 3.3.

3.2 *Software Containers*

Software containers, assim como *Máquinas Virtuais (VMs)*, são tecnologias de virtualização que podem ser aplicadas a diferentes cenários [2].

As VMs permitem a virtualização através de alocação e gestão de *hardware*, ainda que virtualizado, oferendo máquinas que se comportam como servidores/computadores reais e que apresentam as mesmas características, em que é instalado um sistema operativo completo e tudo o que for necessário para correr as aplicações [2]. A virtualização por VMs segue uma arquitetura que pode ser vista na figura 3.2, e dela fazem parte a máquina *host* (que é um computador ou servidor ou outro dispositivo que contem o *hardware* que irá permitir a virtualização), um hipervisor (que é um *software* responsável por criar e executar máquinas virtuais) e as várias instâncias de máquinas virtuais, também conhecidas por *guests*. Cada uma tem o seu próprio sistema operativo, as suas bibliotecas e as aplicações que executa [9].

[5] `https://aws.amazon.com/pt/elasticbeanstalk/`
[6] `https://azure.microsoft.com/services/app-service/`
[7] `https://www.heroku.com/`
[8] `https://cloud.google.com/appengine`
[9] `https://www.facebook.com/`
[10] `https://www.google.com/gmail/`
[11] `www.microsoft.com/Office/365`
[12] `https://www.google.com/docs/`
[13] `https://www.salesforce.com/eu/?ir=1`
[14] `http://www.dropbox.com`

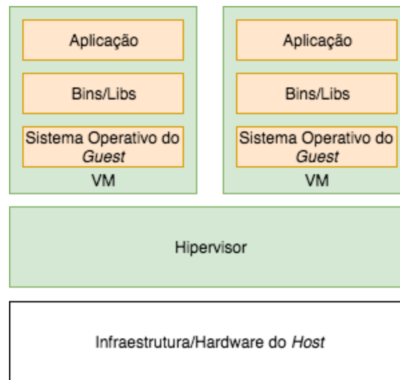


Figura 3.2: Arquitetura de virtualização baseada em Máquina Virtual (VM) (Adaptado de [2])

Cada instância de uma VM armazena no seu *host* um ficheiro isolado de grande dimensão, que é utilizado para armazenar todo o sistema de ficheiros da máquina *guest*, e tipicamente é executada num único processo de elevada carga computacional, tanto a nível de utilização de processador, como de memória. O nível de isolamento entre cada VM é alto, visto que cada uma corre num processo distinto, têm o seu próprio sistema operativo e o seu próprio sistema de ficheiros. Tipicamente, cada VM precisa de espaço de disco, na ordem dos GBs, e de memória considerável na máquina *host* e pode ter um tempo de inicialização de alguns minutos até que uma aplicação possa ser executada [2].

Os *software containers* podem ser entendidos de uma forma muito simples como máquinas virtuais mais leves, sem sistema operativo próprio, contendo tudo o que é necessário para que uma aplicação seja executada: O código da aplicação, o ambiente de execução, dependências, bibliotecas e ficheiros de configuração. Permitem que se empacotem aplicações juntamente com as bibliotecas e fornecem um ambiente isolado para a execução das aplicações [9].

A arquitetura de *containers* encontra-se representada na figura 3.3. É constituída pela máquina *host*, pelo seu sistema operativo, e pelo conjunto de *containers* que estão a ser executados no *host*. Cada um destes *containers* contém as aplicações e as bibliotecas necessárias para que as aplicações possam ser executadas.

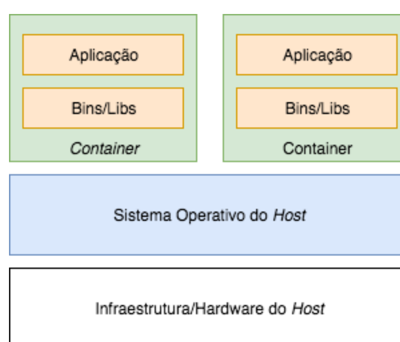


Figura 3.3: Arquitetura de virtualização baseada em *containers* (Adaptado de [2])

Como se pode verificar pela arquitetura, os *containers* usam a virtualização ao nível do

sistema operativo do *host*, e correm diretamente sobre o seu *kernel* num processo isolado. O espaço ocupado em disco por cada *container* é bastante mais baixo do que o requerido por cada VM, na ordem dos MBs, assim como o tempo de instanciação que é quase imediato, na ordem de poucos segundos. Como se pode verificar, tendo em conta estes aspetos, os *containers* são mais leves, tem um menor tempo de iniciação, e necessitam de muito menos recursos que as VMs, tanto a nível de espaço em disco, como de memória e de processamento, mantendo o isolamento. O facto de não haver virtualização de *hardware* permite obter também uma melhor performance computacional [9].

A capacidade de instanciação de *containers* de forma quase imediata permite que estes sejam criados quando são necessários e destruídos quando deixam de o ser, libertando assim recursos computacionais. Os *containers* são altamente portáveis, pois é possível criar imagens destes e executá-las em qualquer sistema que suporte a tecnologia. A mais conhecida destas é o *Docker* que irá ser abordado na subsecção seguinte assim como o *Docker Swarm* e o *Kubernetes*, que são sistemas de orquestração de *cluster* de *containers* [9]. Na tabela 3.1 são apresentadas as principais diferenças entre os dois tipos de virtualização.

	VM	***Container***
Nível de virtualização	Virtualização de *hardware* em que cada VM tem o seu	Virtualização de sistema operativo
Sistema operativo do *guest*	Cada VM tem o seu sistema operativo e *kernel*	Todos os *containers*, que corram no mesmo *host*, partilham o mesmo sistema operativo
Modo de execução	Corre sob um hipervisor instalado na máquina *host*	Cada *container* corre diretamente sob o sistema operativo do *host*
Performance	Pior performance pois depende da emulação de *hardware* e as instruções têm de ser traduzidas do sistema operativo do *guest* para o *host*	Performance semelhante à conseguida pelo sistema operativo do *host*, já que as instruções correm diretamente no seu CPU
Tempo de arranque	As VMs demoram alguns minutos para arrancarem	Os *containers* arrancam em alguns segundos
Recursos computacionais	Necessitam de memória e espaço de armazenamento na ordem dos GBs e de um poder de CPU maior	Necessitam apenas de alguns MBs de armazenamento e memória e consomem menos CPU

Tabela 3.1: Comparação de virtualização através de VMs e de *containers* [9]

3.2.1 *Docker*

O *Docker*[15] é uma solução, *open souce*, de virtualização ao nível de sistema operativo, focada no empacotamento, distribuição e execução de aplicações distribuídas através de *containers Docker*. Estes são baseados nos *Linux*, sendo criados usando funcionalidades do *kernel Linux* como os *namespaces*, que fornecem isolamento entre *containers* e entre estes e o sistema, e os *control groups*, que permitem que se acedam apenas aos recursos computacionais de que necessitam [14]. Apesar de correr de forma nativa apenas em *Linux* pode correr também em ambientes *Windows* ou *Mac* recorrendo para isso à utilização de uma VM minimalista que corre uma distribuição de sistema operativo *Linux* leve [4].

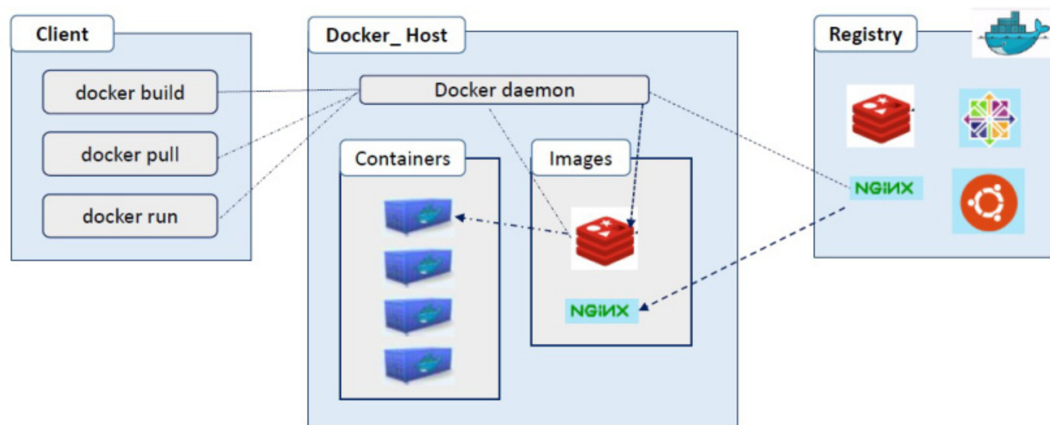


Figura 3.4: Arquitetura do *Docker* (Retirada de [3])

Docker segue uma arquitetura cliente-servidor, que pode ser vista na figura 3.4, e dela fazem parte os seguintes componentes [4] [14]:

- *Docker Engine*: É um ambiente de execução leve e uma ferramenta que permite gerir os *containers* e as imagens, sendo composto pelo *Docker Daemon*, pelo *Docker Client* e pela *Docker API*.
- *Docker Host*: É uma máquina virtual ou física que corre o *Docker daemon* e suporta a execução dos *containers* criados a partir de imagens que armazena em *cache*, *Docker Images*.
- *Docker Daemon*: Aplicação que corre em *background* no *host* e que executa os comandos que permitem criar, executar e distribuir os *containers*.
- *Docker Container*: *Container* criado através do *template* contido na *Docker Image* de origem.
- *Docker Images*: São templates criados a partir de *Dockerfiles*, que incluem os passos para a instalação e execução da aplicação, que permitem a criação de *Docker Containers*.

[15]`https://www.docker.com/`

- *Docker API*: Uma *API REST* que permite a iteração de forma remota com o *Docker Daemon*.
- *Docker Client*: É uma interface, que pode ser de linha de comandos, que permite a comunicação com o *Docker Daemon* através da utilização da *Docker API*.
- *Docker Registry*: É um repositório de *Docker Images* que permite que estas sejam partilhadas. Um dos mais populares é o *Docker Hub*[16].
- *Dockerfile*: Contém as instruções e comandos de instalação e configuração de *software*, que permitem criar uma *Docker Image*, como por exemplo, instruções *linux* que permitem instalar pacotes de *software*, definições de variáveis de ambiente, e comandos para a execução de código.

No *Docker* existem uma sequência de passos, representados na figura 3.5, que permitem a criação e distribuição de imagens e a execução de *containers*.

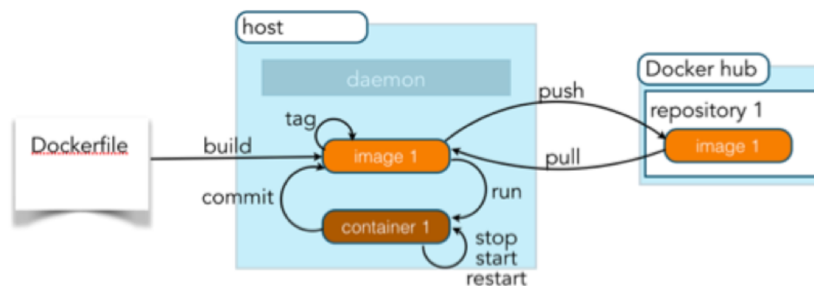


Figura 3.5: Fluxo de funcionamento do *Docker* (Retirada de [4])

Uma imagem é criada a partir de um *Dockerfile*, ou pode ser descarregada de um *Docker Registry*, e a partir dela é possível criar um *container* que age como um ambiente de execução de aplicações e que contém todas as configurações descritas na imagem. Os *containers* podem ser parados, iniciados ou reiniciados e caso sejam efetuadas alterações diretamente nas suas configurações podem ser geradas novas imagens que refletem as modificações e que podem ser usadas para correr novos *containers*. Caso se pretenda partilhar as imagens apenas terão que se carregar para um repositório do *Docker Registry* [4].

3.2.2 *Docker Swarm*

Docker Swarm, ou modo *swarm*, é uma funcionalidade nativa do *Docker*, embutida no seu *Engine*, que permite a orquestração e gestão de *clusters Docker*. Consiste em múltiplos *Docker Hosts*, contendo *Docker Engine*, denominados de nós, que correm no modo *swarm* e que agem como gestores (Managers) e como *Workers*, assentando nos conceitos de serviços e tarefas [15].

[16]`https://hub.docker.com`

Um serviço é a definição de uma, ou mais, tarefas que serão executadas nos nós *Worker*, que inclui a especificação da imagem do *container* que irá ser usada e os comandos que serão executados no *container*. Quando é criado é também definido o seu estado ótimo, que inclui o número de réplicas, os recursos disponíveis para esse serviço e algumas configurações. Uma tarefa é um *container* executável que faz parte do serviço e é gerido pelos nós *Manager* [15].

Para que uma aplicação seja executada no *Swarm*, um dos nós *Manager* através da definição do serviço, envia tarefas para os nós *Worker*, e estes encarregam-se de as executar. Os nós *Manager* garantem que o estado ótimo definido para cada serviço é alcançado, como por exemplo, garantindo a replicação das tarefas de forma a atingir o número pretendido e implementando mecanismos de tolerância a falhas e de balanceamento de carga [15].

3.2.3 *Kubernets*

O *Kubernetes*[17], ou *k8s* ou *kube*, é um sistema de orquestração *open source* para gestão de aplicações que corram em *containers*, distribuídos por vários *hosts*, sendo suportados os *containers Docker*. Possui mecanismos para o *deploy*, manutenção e escalabilidade das aplicações. É especificado o estado que se pretende para o *cluster*, por exemplo, o número de *containers* de uma certa aplicação que queremos ativos, e o *Kubernetes* encarrega-se de o garantir [5].

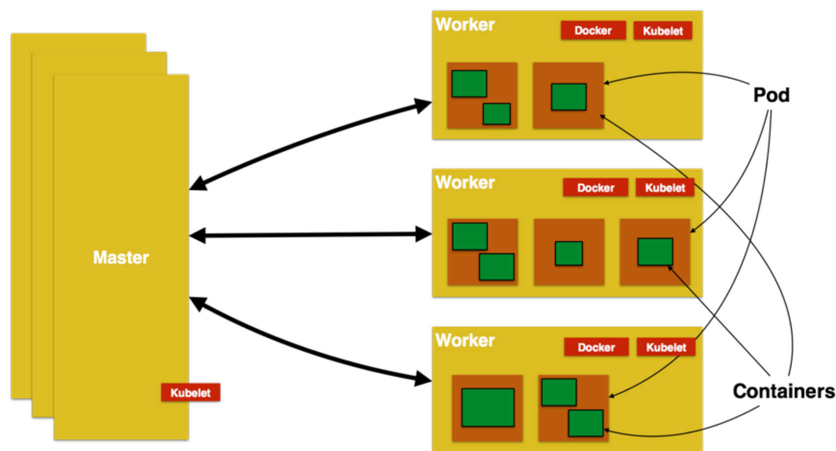


Figura 3.6: Arquitetura do *Kubernetes* (Retirada de [5])

A arquitetura do *Kubernets* e algumas das suas componentes estão representadas na figura 3.6. Um *cluster Kubernetes* é constituído por um conjunto de máquinas, físicas ou virtuais, que são usadas para a execução dos *containers*, que contêm as aplicações, e que são chamadas de nós (*Nodes*). Os *Master Nodes* são responsáveis por gerir o *cluster* e os *Workers* encarregam-se de correr os *containers* utilizando para isso o *Docker*. O *Pod* é a unidade mais pequena que pode ser criada, escalonada e gerida pelo *Kubernetes*, e dele fazem parte um conjunto de *containers* que possam pertencer a uma aplicação. O *Kubelet*

[17] `https://kubernetes.io/`

é um serviço, gerido pelos *Masters*, que corre em cada nó, que gere os *containers* e garante que os que tiverem sido definidos nos *Pods* foram iniciados e se encontram em execução [5].

3.3 Computação *Serverless*

A computação *Serverless*, também conhecida por *FaaS*, é um modelo de serviço de computação em *cloud*, altamente escálavel, que permite a execução de aplicações, sobre a forma de funções sem estado, sem que o utilizador tenha de provisionar ou gerir o servidor e os recursos computacionais necessários. O mesmo apenas tem de criar e fazer o *deploy* na plataforma das funções, que contêm a lógica das suas aplicações, e o provedor de *serverless* encarrega-se de tudo o resto. O termo *serverless* pode dar ideia que não existem servidores, contudo esta ideia está errada. Continua a existir uma infraestrutura e um ambiente de execução que suporta o modelo, e que permite o armazenamento e a execução das funções, só que esta é totalmente provisionada, gerida e mantida pelo provedor de *FaaS* [16]. Este deve garantir que a plataforma é tolerante a falhas e que consegue escalar de forma automática.

As funções não podem ter estado, pois existe uma abstração onde a computação está desconectada de onde esta vai ser executada. Duas execuções consecutivas de uma mesma função podem ocorrer em locais distintos e, por isso, não é possível armazenar um estado na função, permitindo também que estas possam ser reutilizadas em diferentes contextos e em diferentes aplicações.

Considerando os modelos de serviço da *cloud* abordados na secção 3.1, e considerando o nível de abstração e de controlo que *FaaS* oferece ao utilizador, podemos colocá-lo como estando entre *PaaS* e *SaaS*. *FaaS* tem o mesmo nível de controlo e abstração que *SaaS* sobre a infraestutura, pois ambos correm e usam uma sobre o qual o utilizador não tem qualquer capacidade de gerir ou modificar. Ao nível das aplicações o utilizador tem um controlo igual ao que teria em *PaaS* em que é o utilizador que cria e gere as suas aplicações [16].

McGrath [17], refere que o *FaaS* deve seguir uma abordagem guiada por eventos, em que a sua ocorrência, dentro ou no exterior da *cloud*, deve desencadear a execução de uma função *serverless*. Uma aplicação *serverless* é então constituída por um conjunto de funções, também chamadas de ações, e pelos eventos que desencadeiam a execução das ações e que devem também ser definidos pelo utilizador. Cada uma destas funções pode escalar de forma independente de acordo com o pedido de execuções, o que faz com que a aplicação apresente o mesmo comportamento.

As plataformas *serverless* tiram proveito dos *containers* para implementarem o seu modelo de computação. Os mesmos são usados como ambiente de execução de rápido aprovisionamento, onde as funções são colocadas e executadas, mantendo o isolamento entre elas. A utilização de *containers* permite que não sejam alocados recursos até que sejam efeti-

vamente necessários, o que permite aos provedores obterem uma melhor otimização dos recursos. Quando é recebido um pedido de invocação de uma função, um *container* é provisionado e preparado para que a função possa ser executada, consumindo recursos computacionais necessário para a operação, sendo estes libertados após a conclusão da mesma. Apesar de trazer uma grande otimização na gestão de recursos, o tempo de aprovisionamento do *container* acrescenta algum *overhead* no tempo de execução, que é referido em [16] como sendo um "cold start". Algumas plataformas aplicam algumas técnicas de forma a minimizar este "cold start"e que irão ser abordadas no capítulo 4.

FaaS simplifica a criação de aplicações em *cloud*, por abstrair dos recursos e plataformas subjacentes, permitindo o rápido *deploy* de funções que respondem a eventos. O cliente apenas paga pelas invocações das suas funções e pelo tempo de computação das mesmas, não pagando nenhum custo associado diretamente aos recursos computacionais utilizados. Quando não está a ser executada qualquer função de um utilizador, este não terá qualquer custo associado, o que é uma grande vantagem, pois só se paga efetivamente pelo que se usou.

Alguns dos casos de uso para a computação *serverless* são: Processamento de eventos, composição de API's e aplicações de Internet of Things (IoT) [16]. Mais casos de uso serão apresentados no capítulo 4.

3.4 *Workflows Serverless*

Workflows serverless, ou fluxos de trabalho *serverless*, são um modelo de programação que permite, através da coordenação e orquestração de funções *serverless*, criar aplicações mais complexas e mais poderosas, usando para isso a definição de um fluxo de controlo que define a forma como as funções vão interagir entre si e a forma como os dados vão fluir entre elas. Apesar da aplicação ser produzida através da utilização e orquestração de componentes *serverless* sem estado, durante a execução do *workflow* passa a existir um estado, que é atualizado após a execução de cada uma das funções que compõe a aplicação e que é passado à função seguinte, de acordo com o fluxo definido.

Geralmente, os *workflows* criados são apresentados de forma visual, por exemplo, sob a forma de um diagrama ou grafo, o que permite uma fácil leitura e interpretação da aplicação criada e uma fácil visualização das interações entre funções. Alterar um *workflow*, também é, de forma geral, um processo rápido e intuitivo, que passa por adicionar, remover ou editar uma das suas componentes.

Algumas ferramentas de construção e implementação de *workflows serverless* permitem, para além da orquestração das funções, ter componentes com algum tipo de lógica, nomeadamente para tratamento de exceções, controlo de fluxos com instruções condicionais e de ciclos.

Na figura 3.7 está representado, sob a forma de um diagrama, um exemplo de um *work-*

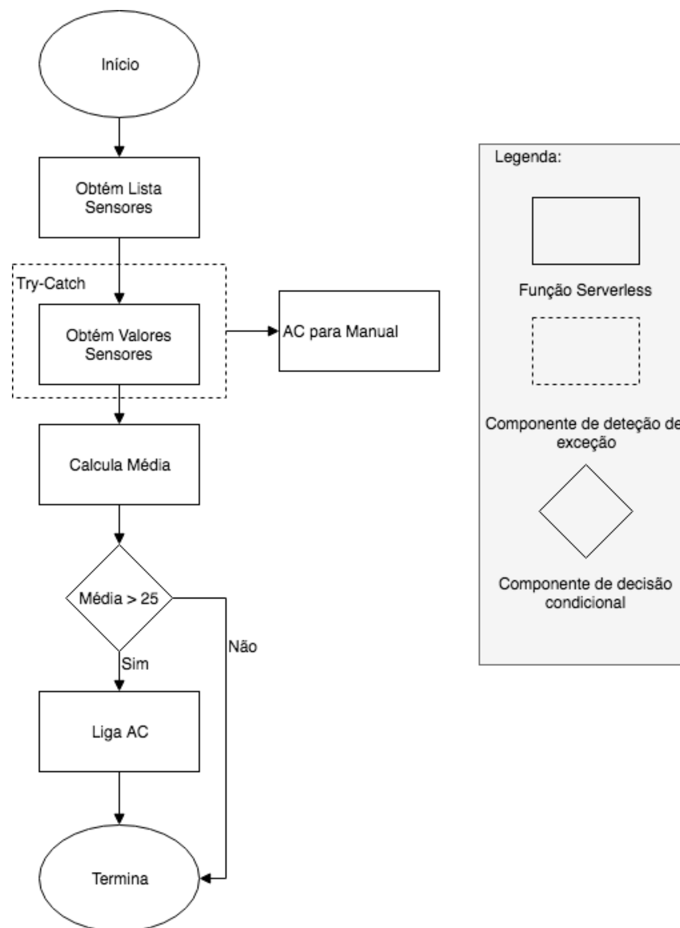


Figura 3.7: Representação visual de um exemplo de *workflow*

flow de um sistema de ar condicionado de uma sala, que implementa uma aplicação que gere o seu funcionamento. São orquestradas um conjunto de funções *serverless* e incluídas componentes de lógica. A aplicação inicia a sua execução invocando uma função *serverless* que permite obter a lista de sensores instalados numa sala. Após obter a lista, é executada uma nova função, passando-lhe a lista de sensores, que obtém os valores de temperatura recolhidos por eles. Caso seja detetada uma exceção/erro na execução da função, é invocada uma outra que torna o ar condicionado controlado de forma manual. Caso não ocorra nenhuma exceção são obtidos os valores de temperatura e é chamada uma função que calcula a média e retorna-a. Com a média e através de uma componente condicional é verificado se o valor é superior a 25. Em caso afirmativo é realizada uma execução que permite ligar o ar condicionado.

Capítulo 4

Plataformas *Serverless*

Neste capítulo são apresentadas algumas das plataformas *serverless* que existem atualmente. Inicialmente, são apresentadas quatro alternativas comerciais, que podem ser usadas sem qualquer tipo de configuração, disponibilizadas como serviço por alguns dos maiores provedores de *cloud*, sendo elas a *AWS Lambda*, a *Microsoft Azure Functions*, a *Google Cloud Functions* e a *IBM Cloud Functions*, que é baseada na *open source Apache OpenWhisk*. De seguida são apresentadas duas opções *open source*, especialmente desenhadas para correrem na infraestrutura de uma *cloud* privada, criada através de *OpenStack*, que é um dos sistemas de operação de *clouds* privadas mais utilizado. Por fim, são apresentadas duas plataformas, também *open source*, que podem ser instaladas e executadas em diferentes infraestruturas e até em computadores pessoais. Existem outras plataformas como a *OpenLambda*[1], a *kubeless*[2] e a *IronFunctions*[3] que não serão abordadas por apresentarem falta de documentação, levando a que o seu estudo fosse difícil.

4.1 *AWS Lambda*

A *Amazon Web Services (AWS) Lambda*[4] é uma plataforma de computação *serverless*, da *AWS* lançada em Dezembro de 2014. É considerada a responsável pelo surgimento do modelo *serverless* na *cloud*, que executa automaticamente funções em resposta a eventos, sem que seja necessário o utilizador requerer ou gerir os recursos computacionais necessários, tendo este apenas de escrever o código, chamado de função *Lambda*, carregá-lo para a plataforma e definir os eventos que permitem acionar a execução [18].

A *AWS Lambda* garante a escalabilidade das funções de forma automática. Quando é recebido um pedido de execução de uma função, a plataforma localiza e disponibiliza capacidade computacional de forma a responder ao pedido. Como as funções não têm estado (são *stateless*), existe a capacidade de serem iniciadas de forma rápida quantas

[1] `https://github.com/open-lambda/open-lambda`
[2] `https://github.com/kubeless/kubeless`
[3] `https://github.com/iron-io/functions`
[4] `https://aws.amazon.com/lambda/`

cópias forem necessárias, alocando dinamicamente recursos que permitem atender todos os pedidos de invocação. Caso o utilizador pretenda armazenar algum estado, devem ser usados serviços de armazenamento como o *S3*[5], a *Amazon DynamoDB*[6] ou um outro que esteja disponível.

Para garantir uma alta disponibilidade é usada replicação. A *AWS Lambda* mantém capacidade computacional, destinada à plataforma, distribuída em várias zonas de disponibilidade em cada região, mantendo também cópias das funções nessas diversas zonas.

Ao criar uma nova função na plataforma, o utilizador pode especificar o nome que lhe será atribuído, uma descrição , o ambiente de execução (qual a linguagem de programação do código), a memória desejada, o tempo máximo que poderá usar em cada execução, as regras de gestão de identidade e de acesso, o código da função e o método da função que será iniciado quando a execução desta arrancar, chamado de *handler*. É suportado código escrito em *Node.js, Python, Java, C#* e *Go*. A memória alocada para cada função pode ser definida entre 128MB e 3GB em incrementos de 64MB, e ao ser definida mediante o que se pretende para a função, a capacidade de processamento e de outros recursos computacionais, como a largura de banda, são também alterados. Por exemplo, se o utilizador escolher 256 MB de memória para uma função é alocada aproximadamente duas vezes mais potência de *CPU* do que para uma que tenha sido definida com 128 MB. Cada função recebe 512MB de espaço não persistente temporário em disco, que permite, por exemplo, escrever ficheiros que serão acessíveis enquanto o código estiver a ser executado. Por omissão, existe um limite de tempo de três segundos, que pode ser alterado por um valor entre um e trezentos segundos, para cada execução. Caso esta não seja terminada dentro do tempo definido é interrompida. O código de uma função pode ser escrito diretamente no editor de código da consola da *AWS Lambda*, que permite também o teste das funções e a visualização dos resultados das suas execuções num ambiente semelhante ao de um Ambiente de Desenvolvimento Integrado (IDE). O utilizador pode ainda escrever o código localmente (no seu computador), comprimi-lo (juntando quaisquer bibliotecas dependentes) num arquivo *ZIP* e carregá-lo usando a consola da *AWS Lambda*. Outra opção é enviar o código para a *Amazon S3* e especificar a sua localização no momento da criação da função. Os arquivos para *upload* não devem ter mais de 50 MB depois de comprimidos. É possível ainda usar o *plugin AWS Eclipse* para criar e fazer o *deploy* de funções *Lambda* em Java ou o *plugin* do *Visual Studio* para funções em *C#* e *Node.js*. Para cada função, o utilizador pode ainda definir e associar um conjunto de variáveis de ambiente que permitem passar informações ou definições a uma função ou às bibliotecas que esta utiliza, sem que seja necessário alterar o código.

O código que é carregado para a *AWS Lambda* é encriptado e armazenado na *AWS S3*. Quando é feito um pedido para a execução de uma função é desencadeado o processo que pode ser visto na figura 4.1. Ao ser realizada uma invocação, a plataforma pede à *AWS S3* a função. Após a sua receção cria um novo *container linux* com as definições

[5] `https://aws.amazon.com/s3/`
[6] `https://aws.amazon.com/dynamodb/`

associadas (ambiente de execução, memória e tempo de execução), faz o *deploy* do código nesse *container*, que após ser criado é colocado numa instância da *AWS EC2* que será responsável por lançá-lo. Este por sua vez executará o código. Quando a execução termina, o resultado é retornado para a plataforma que o envia para o cliente.

As funções *Lambda* podem ser executadas de forma síncrona, seguindo o processo descrito acima, mas também de forma assíncrona, em resposta a eventos, em que o cliente não fica bloqueado à espera de uma resposta.

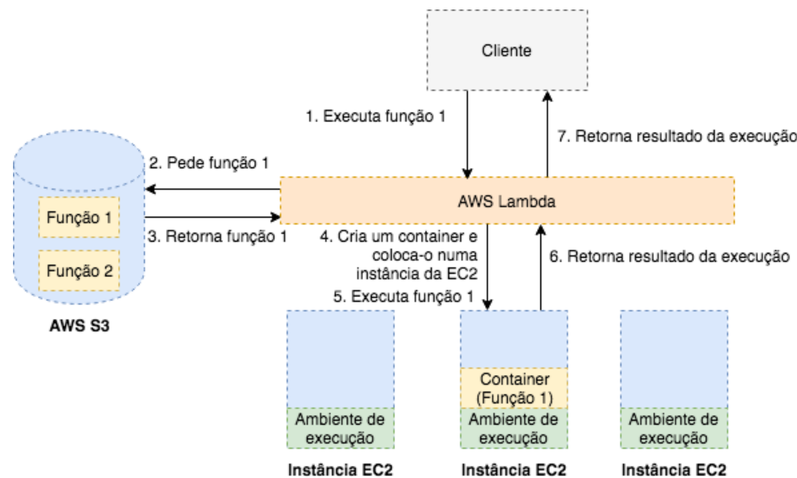


Figura 4.1: Processo de execução de uma função na *AWS Lambda* (Adaptado de [6])

Para melhorar o desempenho, a *AWS Lambda* pode decidir reter uma instância de um *container* de uma função durante alguns minutos e reutilizá-la para atender a uma solicitação posterior, em vez de criar uma nova cópia. Quando é atendido um pedido de execução de uma função com uma instância que já estava criada, utiliza-se o que se chama de *warm container*, que permite melhorar o desempenho da execução de uma função, pois o tempo de criação de um container e de *deploy* da função no mesmo é eliminado. Quando é criado de novo um *container* para uma função diz-se ser um *cold container*. Ao processo e tempo de *overhead* de criação e *deploy* deste dá-se o nome de *cold start*. O processo de utilização de *contaires* pode ser visto na figura 4.2. A plataforma pode ainda criar instâncias de funções previamente, baseando-se em padrões de uso, permitindo assim que futuros pedidos de execução sejam processados em *warm containers* evitando assim o *cold start* em alguns momentos [7].

As funções *Lambda* podem ser invocadas através de algumas fontes de eventos, ou *triggers*, que podem ter origem em vários serviços da *AWS* ou em proveniências externas. A *API Invoke*[7] permite a chamada direta a uma função e normalmente é usada para testes. Existem dois modelos que permitem a invocação de funções: o *pull* e o *push*. No primeiro, a *AWS Lambda* pesquisa numa fonte de dados e caso ocorra uma alteração nesta é despoletada uma função. Exemplos de *triggers* do modelo *pull* são a alteração de uma tabela da *Amazon DynamoDB*[8] ou a alteração de uma *stream* de dados na *Amazon*

[7] `https://docs.aws.amazon.com/lambda/latest/dg/API_Invoke.html`
[8] `https://aws.amazon.com/dynamodb/`

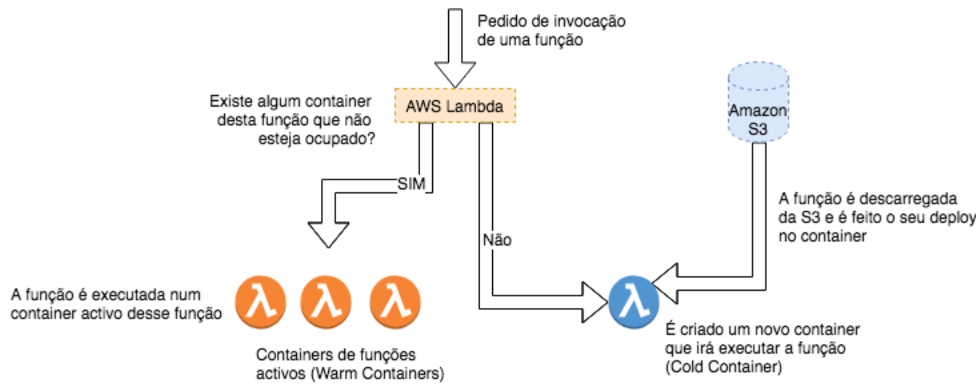


Figura 4.2: *Warm* e *Cold containers* na *AWS Lambda* (Adaptado de [7])

Kinesis Data Streams[9]. No modelo *push*, sempre que ocorre um evento particular é executada uma função. Exemplos de eventos *push* são a adição/remoção de um ficheiro na *Amazon S3*, um pedido *HTTP* realizado a um método criado através da *API Gateway*[10], uma nova mensagem num tópico da *Amazon SNS*[11] ou um evento ocorrido na *Amazon CloudWatch*[12].

O custo da *AWS Lambda* é baseado na utilização, sendo os utilizadores cobrados pelo número de execuções, a duração das mesmas e pela quantidade de memória alocada a cada função que tenha sido ativada. A duração é calculada a partir do momento em que o código começa a ser executado até que termina, arredondando-se aos 100ms mais próximos. A *AWS* oferece um nível gratuito, que inclui 1 milhão de execuções gratuitas por mês e 400.000 GB segundo, até 3,2 milhões de segundos, de tempo de computação por mês. Após este nível ser usado o serviço é taxado a 0,20 Dólar dos Estados Unidos (USD) por cada milhão de execuções e 0,00001667 USD por cada GB segundo. Para diferentes níveis de memória alocada o preço por cada 100ms de execução e o tempo de execução gratuito são apresentados na tabela 4.1. Apenas é exposto o preço para alguns níveis de memória alocada, sendo possível ter uma visão completa da tabela de preços no *site* da plataforma[13].

Memória (MB)	Preço por 100ms (em USD)	Tempo de execução gratuito (em s)
128	0,000000208	3.200.000
256	0,000000417	1.600.000
512	0,000000834	800.000
1024	0,000001667	400.000
2048	0,000003334	200.000
3008	0,000004897	136.170

Tabela 4.1: Preço da *AWS Lambda* por nível de memória

A plataforma *Lambda* permite a utilização e integração com outros serviços da *AWS*. Entre

[9] https://aws.amazon.com/kinesis/data-streams/
[10] https://aws.amazon.com/pt/api-gateway/
[11] https://aws.amazon.com/sns/
[12] https://aws.amazon.com/cloudwatch/
[13] https://aws.amazon.com/pt/lambda/pricing/

estes estão a *Amazon CloudWatch Logs*[14] que permite a utilização de *loggs* nas funções, a *Amazon CloudWatch*, que regista métricas da plataforma, tais como o número e tempo de execuções por função, e a *AWS X-Ray*[15], que apresenta de forma gráfica a arquitetura das aplicações desenvolvidas e métricas individuais de cada uma das componentes de uma aplicação.

A *AWS Lambda* integra ainda com o *AWS Step Functions*[16] que é um sistema de *workflow* que permite orquestrar um conjunto de funções da *Lambda* numa ordem específica. É possível invocar várias funções *serverless* sequencialmente, passando o *output* de uma para o *input* de outra, e/ou em paralelo, e ainda adicionar tratamento de erros e condições lógicas, ficando o sistema responsável por manter o estado. Com a utilização conjunta do *Step Functions* e da *Lambda* é possível criar aplicações distribuídas com estado. O sistema é baseado no conceito de tarefas, que podem ser funções ou componentes de lógica, e máquina de estados, que são definidas usando a *Amazon States Language*[17] baseada em *JSON*. A consola do sistema permite visualizar de forma gráfica a estrutura da máquina de estados e a lógica da aplicação, assim como verificar e monitorizar as execuções.

Os casos de uso mais comuns para a utilização da *AWS Lambda* são o processamento de ficheiros, o processamento e análise de dados e *backends serverless* (lógica de aplicações) de *websites* e/ou de aplicações *mobile* e/ou de *Internet of Things (IoT)* [19]. Alguns dos utilizadores mais relevantes são a *Netflix*, a *Coca-Cola Company*, a *Nordstrom*, a *Benchling*, a *Thomson Reuters*, a *Guardian News Media*, a *PhotoVogue*, a *MLBAM* e o *Seattle Times*.

4.2 *Microsoft Azure Functions*

Microsoft Azure Functions, lançada em Março de 2016, foi desenhada para expandir a plataforma de aplicações da *Azure* com a capacidade de executar código, num ambiente *Windows*, desencadeado por eventos ocorridos nos serviços da *Azure* ou de terceiros [19]. A plataforma é um serviço de computação *serverless* que permite que os utilizadores executem o seu código sob pedido, sem que se tenham de preocupar com o provisionamento ou gestão da infraestrutura subjacente [20].

Existem duas opções de funcionamento da *Azure Functions*: O “App Service Plan”e o “Comsumption Plan”. Na primeira, as funções correm em Máquinas Virtuais (VMs) dedicadas, que por padrão são em ambiente *windows*, podendo o utilizaddor optar por um ambiente *linux*, e que são alocadas para o efeito. A escalabilidade pode ser feita de forma manual, em que o utilizador adiciona ou remove *VMs*, ou de forma automática, em que é a própria plataforma que se encarrega desta ação. Neste modelo, o utilizador é cobrado pelo numero de máquinas virtuais e pelo tempo que estas se encontram em funcionamento.

[14] `https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/WhatIsCloudWatchLogs.html`
[15] `https://aws.amazon.com/pt/xray/`
[16] `https://aws.amazon.com/pt/step-functions/`
[17] `https://docs.aws.amazon.com/step-functions/latest/dg/concepts-amazon-states-language.html`

Esta opção pode trazer vantagens a nível de custo quando já possuem outras aplicações a correr em *VMs*, que podem ser utilizadas para executar as funções *serverless*. Pode ainda ser uma boa opção quando os limites impostos na outra opçao não são suficientes para o que o utilizador pretende. No "Comsumption Plan"o utilizador não tem qualquer controlo sobre a escalabilidade da aplicação e as funções são executadas em ambiente *windows*. As instâncias são adicionadas e removidas pela *Azure Functions* de forma automática para que lidem com a carga de pedidos. Cada execução tem um tempo limite de 600 segundos para ser processado e cada instância da função está limitada a 1536MB de memória, que é alocada dinamicamente e de forma automática pela plataforma de forma a corresponder as necessidades. O sistema de pagamento, no *"Consumption Plan"*, é baseado no número de execuções, na soma das suas durações e na memória utilizada. A *Azure* oferece um nível gratuito de um milhão de execuções por mês com 400.000 GB segundo de consumo de recursos. Após ser gasto este nível são cobrados 0,20USD por cada milhão de execuções e 0,000016 por cada GB segundo sendo o tempo arredondado aos 100 ms mais próximos e a memória utilizada por função arredondada aos 128MB mais próximos, até ao máximo de 1536MB, sendo este o valor máximo que uma função pode consumir [20]. Na tabela 4.2 são apresentados os preços por cada 100ms de tempo de execução por nível de memória usada.

Memória (MB)	Preço por 100ms (em USD)
128	0,0000002
256	0,0000004
512	0,0000008
1024	0,0000016
1536	0,0000024

Tabela 4.2: Preço da *Azure Functions* por nível de memória

A *Azure Functions* suporta funções escritas em *Node.Js, C#, F#* e *Java*, que podem ser agrupadas em aplicações, possibilitando a partilha de um conjunto de variáveis de ambiente e partilhando a memória alocada, em múltiplos de 128MB até ao máximo de 1.5GB, entre si. São usados, como gestores de dependência de código, o NPM e o NuGet, sendo por isso possível utilizar bibliotecas que possam ser obtidas por ambos.

O portal da *Azure Functions* fornece um editor online para a escrita de código, criado sob o *Visual Studio Online*, que permite desenvolver funções, testá-las e monitorizá-las no *browser*. Para além desta forma de criar funções, ainda é suportada a integração e o *deploy* continuo de código através do *GitHub*, do *BitBucket* e do *Visual Studio Team Services* [20].

A *Azure Functions* faz integração com vários serviços *cloud* da *Azure* e externos que podem ser usados como *triggers* para a execução de funções. Alguns dos serviços suportados de forma nativa são: Azure Cosmos DB, Azure Event Hubs (Streams), Azure Event Grid (pedidos HTTP de notificação de eventos que ocorram nos *publishers*), Azure Mobile Apps (tabelas de dados em aplicações *mobile*), Azure Notification Hubs (*Push Notifications*),

Azure Service Bus (filas e tópicos), Azure Storage (*Blob Storage, Queues* ou tabelas) e GitHub (*webhooks*) [20]. São também ainda suportados *triggers* de pedidos HTTP (*Rest ou WebHook*), em que cada função tem uma rota associada, e *triggers* baseados em tempo/agendamento [20].

A *Logic Apps*[18] é um serviço que permite criar aplicações *serverless* através da definição *workflows serverless*, de forma visual/gráfica, que orquestram e conectam funções permitindo a existência de estado durante a execução dos mesmos. Uma das grandes vantagens deste serviço é a existência de conectores que permitem a comunicação e interação, de forma fácil, com vários serviços e aplicações da *Microsoft* e externas [21].

A *Microsoft* cita os seguintes casos de uso: *Backend* de aplicações *web*, *mobile* ou *IoT*, processamento de ficheiros em tempo-real, processamento de *stream* de dados em tempo-real e automação de tarefas agendadas [21]. Alguns dos utilizadores de *Azure Functions* são a Plexure, a ZEISS, a FUJIFILM, a Quest, a CARMAX e a Navitime [21].

4.3 *Google Cloud Functions*

Google Cloud Functions é o serviço de computação *serverless* orientado a eventos da *Google*, fazendo parte da sua plataforma de *cloud* desde 2016 e que continua em versão *Beta*. A plataforma gere e escala de forma automática a infraestrutura que permite responder à carga de pedidos de execução e garante alta disponibilidade e tolerância a falhas [22]. As invocações e gestão de funções podem ser geridas pela consola de administração da *Google Cloud* ou através da interface da linha de comandos, a *Google Cloud CLI*.

As funções apenas podem ser escritas em *JavaScript* (*Node.js*), de forma *stateless*, e não podem ter mais de 100MB de tamanho cada, sendo que as dependências que possam existir com bibliotecas externas são geridas pela *Google Cloud Functions* recorrendo ao uso de npm[19]. O código pode ser carregado para a plataforma através de *upload* de um ficheiro *zip* ou ainda através da *Google Cloud Source Repositories*[20] que permite importação direta do *GitHub* e/ou do *Bitbucket*. Em situações que o utilizador necessite de persistir estado devem ser usados os serviços de armazenamento da *Google Cloud*, como por exemplo *Cloud Storage, Cloud Firestore e Cloud Datastore*, ou ainda serviços externos.

As execuções podem ser ativadas através de pedidos HTTP, suportados de forma nativa pela plataforma, em que cada função tem um domínio dedicado e um certificado SSL/TLS gerado dinamicamente para comunicações seguras [22]. São ainda suportados como *triggers* eventos que ocorram na *Cloud Pub/Sub*[21], como a chegada de uma mensagem a um tópico, e a alteração de um ficheiro/objeto na *Cloud Storage*[22]. O tempo máximo que uma função pode estar em execução é de 540 segundos (9 minutos), embora o valor por omissão seja

[18] `https://azure.microsoft.com/services/logic-apps/`
[19] `https://www.npmjs.com/`
[20] `https://cloud.google.com/source-repositories/`
[21] `https://cloud.google.com/pubsub/`
[22] `https://cloud.google.com/storage/`

de 60 segundos. Cada projeto possui um limite máximo de 1000 funções [22].

Cada função executa no seu próprio ambiente isolado, escala de forma automática e tem um ciclo de vida independente das outras funções. De forma a atender todas as invocações concorrentes, podem ser inicializadas múltiplas instâncias da função que correm em paralelo, pois cada uma apenas suporta um pedido de cada vez. A plataforma reutiliza instâncias de funções que estejam ativas, mas que estejam paradas, de forma a remover o *cold start* nas invocações [22].

A *Google Cloud Functions* é taxada pela utilização, sendo cobrados 0.4USD por cada milhão de invocações, 0.0000025USD por GB/s e 0.00001 por GHz segundo de tempo de computação aproximado aos 100ms mais próximos, e 0.12USD por cada GB de tráfego de rede para comunicação com serviços externos à *Google Cloud Platform.* A plataforma tem um nível gratuito em que são oferecidas 2 milhões de invocações por mês até 400.000 GB segundo de memória, 200.000 GHz segundo de processamento, 1 milhão de segundos e 5 gigabytes de tráfego de rede [22]. Ao criar uma função na plataforma, o utilizador pode escolher a quantidade de memória que lhe deseja alocar, sendo-lhe associada também uma capacidade de CPU. Na tabela 4.3 estão apresentados os diferentes níveis de memória, assim como a capacidade de CPU associada e o preço por 100ms de tempo de execução.

Memória (MB)	CPU (MHz)	Preço por 100ms (em USD)
128	200	0,000000231
256	400	0,000000462
512	800	0,000000925
1024	1400	0,00000165
2048	2400	0,0000029

Tabela 4.3: Preço de computação por 100ms da *Google Cloud Functions* por nível de memória e CPU

Para monitorização e *logging*, a *Cloud Functions* integra com os serviços *Stackdriver Monitoring*[23] e *Stackdriver Logging*[24], que oferecem capacidade de armazenar, pesquisar, analisar e monitorizar os *loggs* produzidos pelas execuções de funções assim como de métricas registadas.

São citados alguns casos de uso para a plataforma, nomeadamente, o *backend serverless* de aplicações (*web, mobile* e *IoT*), o processamento de dados em tempo real (processamento de ficheiros, de *streams* de dados e *ETL*) e aplicações inteligentes (assistentes virtuais, *chatbots* e análise de imagens/vídeo) [22]. Alguns dos utilizadores da *Google Cloud Functions* são a Meetup, a HomeAway, a Smart Parking, a Incentro, a Turner TimeWarner e a Semios [22].

[23] `https://cloud.google.com/monitoring/`
[24] `https://cloud.google.com/logging/`

4.4 *Apache OpenWhisk/IBM Cloud Functions*

Apache OpenWhisk[25] é uma plataforma, robusta e de alta escalabilidade automática, de computação *serverless*, para *cloud*, que executa funções em resposta a eventos ou a invocações diretas. É *open source*[26], sob licença Apache, e tem como maior contribuidor de desenvolvimento a IBM, fazendo parte da *IBM Bluemix* sob o nome *IBM Cloud Functions* desde Dezembro de 2016. A *OpenWhisk* abstrai a infraestrutura subjacente e os conceitos de operação da mesma, permitindo aos programadores focarem-se no código e no desenvolvimento das suas aplicações. A *OpenWhisk* pode ser instalada em qualquer infraestrutura que suporte *containers Docker*.

A plataforma suporta, de forma nativa, funções sem estado escritas em *Node.js, Swift, Java, Go, PHP* e *Python*, suportando ainda outra qualquer linguagem, recorrendo diretamente à criação de *containers Docker* com o ambiente de execução. O *deploy* do código pode ser realizado na interface *web* da *IBM cloud*, caso se esteja a usar a *IBM Cloud Functions*. Outra alternativa é a utilização da interface de linha de comandos da plataforma - a *OpenWhisk/Cloud Functions CLI* - que comunica com a *API Rest* do *OpenWhisk*, que permite a interação com a plataforma [23]. Cada função tem um tamanho máximo de 48MB e pode ser alocada com 128, 256, 384 ou 512 MB de memória [8].

É utilizado um modelo de programação baseado em quatro conceitos: Pacotes (*Packages*), *triggers*, ações (*actions*), e regras (*rules*). Os *packages* fornecem *feeds* de eventos que, ao serem associados a *triggers*, fazem com que estes sejam despoletados quando ocorre um evento. Os utilizadores podem mapear *triggers* com ações, que são as funções *serverless*, usando regras. A *CLI* pode ser usada para interagir com a plataforma, permitindo criar, editar e eliminar funções, *triggers* e regras. São suportadas execuções de funções em sequência, até ao máximo de 50, numa ordem definida pelo utilizador, onde o *output* de uma se torna no *input* da seguinte [8]. Na figura 4.3 está representada uma arquitetura de alto nível da *OpenWhisk*, onde é possível visualizar também os quatro conceitos apresentados.

As funções podem ser executadas sob pedido, quer de forma síncrona quer de forma assíncrona, através de invocações diretas realizadas por pedidos HTTP à API *Rest*, ou de forma automática em resposta a eventos. São suportados de forma nativa, como fonte de eventos, através de *triggers*, os serviços *GitHub*, *IBM Cloudant noSQL DB, IBM Mobile Push, IBM Message Hub, IBM Push Notifications* e *Slack*. Para além destes serviços ainda existe integração com o IBM Watson e Weather Company. [23] Existem ainda muitos *packages* não oficiais, que foram criados pela comunidade de utilizadores e publicados na Internet, que permitem a integração com muitos outros serviços, como por exemplo de MQTT[27]. São ainda suportados *triggers* baseados em tempo e em agendamento cronológico. Cada execução tem um tempo limite de 600 segundos, sendo o valor por omissão

[25] `https://openwhisk.apache.org/`
[26] `https://github.com/apache/incubator-openwhisk`
[27] `http://mqtt.org/`

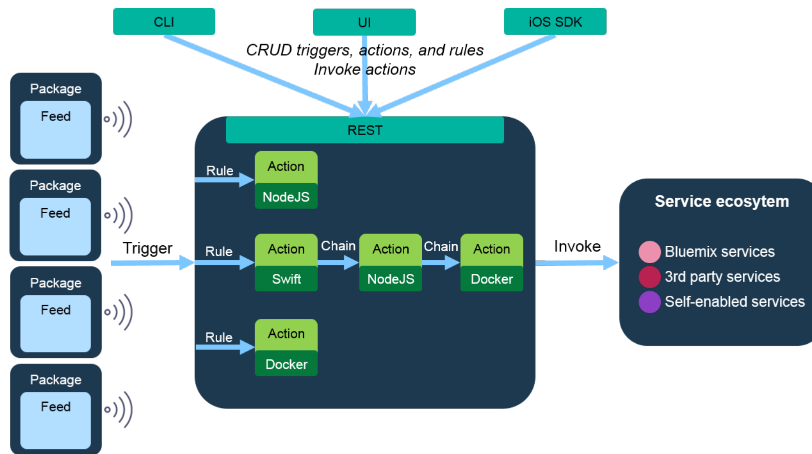


Figura 4.3: Arquitetura de alto nível da *OpenWhisk* [8]

de 60 segundos, e são suportadas 1000 execuções concorrentes por utilizador [8].

A figura 4.4 apresenta uma visão mais detalhada da arquitetura da *Openwhisk*. A plataforma é composta e assenta sobre várias tecnologias *open source*, sendo elas, o *Nginx*[28], o *Kafka*[29], o *Docker*, e a *CouchDB*[30]. Existem ainda duas componentes próprias: o *Controler* e o Invoker [8].
O *Nginx* é um servidor web *open source* que expõe *endpoints* HTTP publicamente e que funciona também como *proxy* reverso para a API. Todos os pedidos feitos à *OpenWhisk* passam por este servidor, sendo depois encaminhados para o *Controller*.
O controlador é uma componente, escrita em *Scala*, que atua como "porteiro"do sistema, sendo responsável por implementar a *API REST OpenWhisk*, por servir de interface para todas as funcionalidades, e por decidir o caminho a tomar por cada pedido garantindo a sua autorização e autenticação. Possui ainda um balanceador de carga que tem uma visão global do estado de todos os *Invokers* disponíveis, verificando o estado destes de forma contínua, escolhendo um deles para invocar cada ação requerida.
A *CouchDB* é um sistema de armazenamento de dados em formato *JSON*, que mantém e gere o estado de toda a plataforma, como por exemplo, credenciais dos utilizadores, metadados, e a definição de ações, regras e *triggers*.
O *Kafka* é um sistema distribuído, de alto rendimento, de mensagens *publish/subscribe* que permite a comunicação entre o *Controller* e o *Invoker* através de mensagens que são armazenadas em memória e persistidas pelo sistema, sendo garantido que mensagens não são perdidas em casos de falha do sistema.
O *Invoker* é a componente, escrita em *Scala*, que permite que uma ação/função seja executada, de forma isolada, rápida e segura. É usado o *Docker* que é responsável, por criar um novo *container* para cada ação invocada. O código é copiado da *CouchDB*, injetado no *container* e executado com os parâmetros de entrada passados. Quando uma execução termina, o *Invoker* é responsável por armazenar o resultado da ativação na *CouchDB*, para que possa ser retornada posteriormente, e por destruir o *container*.

[28] `https://www.nginx.com/`
[29] `https://kafka.apache.org/`
[30] `http://couchdb.apache.org/`

Todos estas componentes são empacotadas e implantadas em *cotainers Docker*, sendo possível desta forma escalar cada uma destas individualmente [24].

De forma a melhorar a performance e a reduzir o *overhead*, a *OpenWhisk* utiliza alguns mecanismos. Nas chamadas à *CouchDB* são usados mecanismos de *cache* em memória, permitindo assim que alguns pedidos não tenham que passar pela base de dados, diminuindo assim o tempo destas operações [25]. Os *containers* podem ser reutilizados, isto é, se uma função foi invocada, a plataforma, mais propriamente o *Invoker*, pode optar por manter o *container* ativo por um período de tempo, sendo que invocações subsequentes da mesma função podem ser atendidas por ele que pode ser denominado de *warm container*. Este processo evita o *cold start* (tempo gasto na criação do *container*) e o tempo de injeção do código nestas ativações. A *OpenWhisk* pré-aloca *prewarmed containers* que contêm o ambiente de execução para as linguagens de programação usadas mais frequentemente, permitindo assim a execução de funções que sejam invocadas no futuro, sem que estas sofram de *cold start* [25].

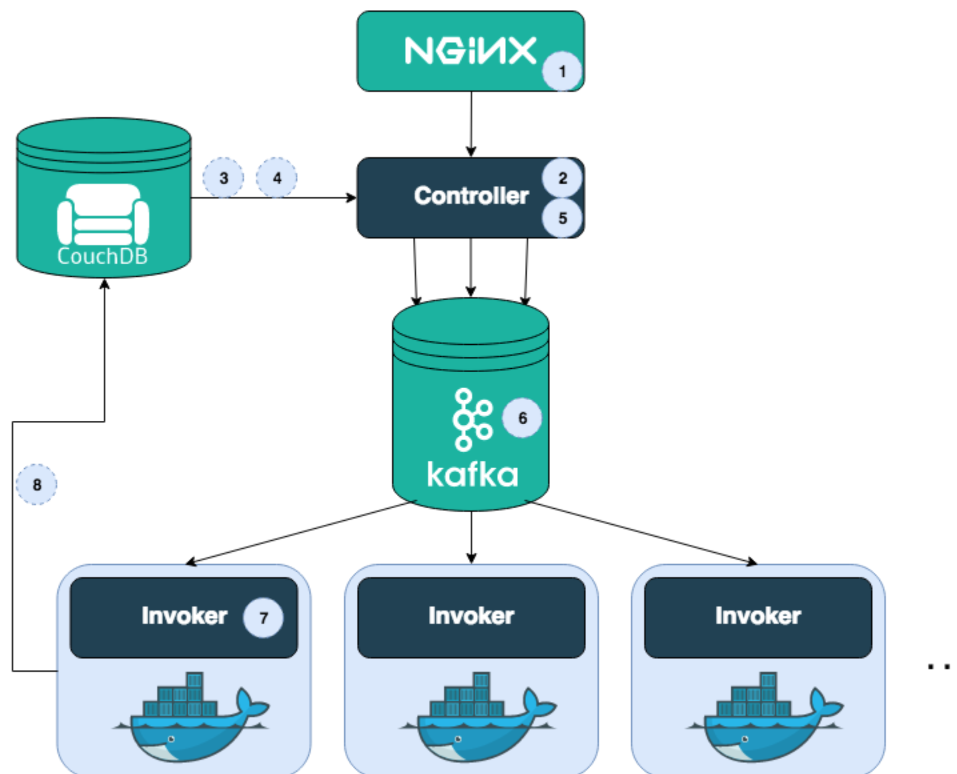


Figura 4.4: Arquitetura e processo de execução de uma função na *OpenWhisk* (Adaptado de [8])

O processo de execução de uma função na *OpenWhisk* está representado seguindo a numeração da figura 4.4 e será explicado de seguida. Um pedido de invocação chega ao *Nginx* através de HTTP (1), que o encaminha para o *Controller*, que numa primeira fase verifica que o pedido se trata de uma invocação de uma função (2), recorrendo depois à *CouchDB* para verificar se o utilizador pode ser autenticado e se este tem autorização para executar a função pretendida (3). Caso se verifiquem as duas condições anteriores, o *Controller*

obtém o registo da função da base de dados (4), que contém o código para executar, os parâmetros *default*, que são utilizados caso não tenham sido passados no pedido de invocação, e todas as definições associadas, como por exemplo a memória. De seguida, o balanceador de carga escolhe um dos *Invokers* (5) e é inserida uma mensagem (que contém a identificação da função a ser executada e os seus parâmetros de entrada) no *Kafka*, que após confirmar a mensagem a envia para o *Invoker* escolhido (6) tendo a execução a partir deste momento um identificador associado, o *ActivationId*, que nos pedidos assíncronos é enviado ao utilizador para que este possa obter mais tarde o resultado da ativação. O *Invoker* verifica se há algum *warm container* para a função e, caso exista, executa a função com os parâmetros passados. Caso não exista, é utilizado um *prewarmed container*, caso exista algum para a linguagem de programação do código, ou é criado um novo. Então, é injetado o código, é executado com os parâmetros e são extraídos o resultado da invocação e os *logs* registados (7). Por fim, o resultado é persistido sob o *ActivationId* na *CouchDB*, juntamente com os *logs* e os tempos de inicio e fim da execução (8) [25].

Na *IBM Cloud Functions* apenas se paga pelo tempo (arredondado aos 100ms mais próximos) em que o código esta a ser executado e pela memória alocada, não havendo um valor associado ao número de invocações. O valor cobrado é de 0.000017USD por GB segundo, sendo que os primeiros 400,000GB/ segundo por mês são gratuitos [23]. Na tabela 4.4 são apresentados os preços, por nível de memória suportados, para cada 100ms de tempo de execução.

Memória (MB)	Preço por 100ms (em USD)
128	0,0000002125
256	0,000000425
384	0,000000638
512	0,00000085

Tabela 4.4: Preço da *IBM Cloud Functions* por nível de memória

O *Composer*[31] é um modulo de programação que permite criar aplicações *serverless* através da composição e orquestração de funções implementadas na *OpenWhisk*, mantendo a escalabilidade e o modelo de pagamento pelo que se gasta. O *Composer* é uma extensão que adiciona controlos de fluxo e gestão automática de estado às funções e às sequências suportadas de forma nativa pela *OpenWhisk* [26]. As composições são programadas em *JavaScript* e é possível usar a *Cloud Functions Shell*[32], que é uma aplicação gráfica, para criação, visualização de forma gráfica e edição de composições, permitindo também executá-las. A aplicação permite ainda o *deploy* de funções na *OpenWhisk* e a atualização das mesmas caso seja necessário [26].

A plataforma pode ser aplicada a diferentes casos de uso, nomeadamente, *backends serverless web, IoT e mobile*, processamento de dados, processamento cognitivo de dados, IoT, processamento de *streams* de eventos, cenários de conversação (*chatbots*) e tarefas

[31] https://github.com/ibm-functions/composer
[32] https://www.npmjs.com/package/@ibm-functions/shell

agendadas. [23]. Alguns dos clientes da *IBM Cloud Functions* são os seguintes: *DOV-E, KONE, AbiliSense, Articoolo, BIGVU, GreenQ, Magentiq Eye Ltd., NeuroApplied, SiteSpirit. Croosing, The Weather Gods e a Qanta.ai* [27].

Foi realizada uma instalação local da plataforma de computação *serverless OpenWhisk*, recorrendo-se a uma máquina virtual *Vagrant*[33], que é uma ferramenta de configuração e provisionamento de máquinas virtuais que permite reproduzir ambientes através da utilização de ficheiros de configuração. Foi utilizado o hipervisor *VirtualBox*[34], que é um sistema de virtualização que permite a criação, execução e gestão de VMs. A máquina virtual possui um ambiente com 4 cores de CPU e 4GB de memória e nela correm todos os componentes da arquitetura da *OpenWhisk*.

Esta instalação local da plataforma teve como principais objetivos o teste de funcionamento da própria plataforma e o desenvolvimento e execução de alguns *workflows serverless* recorrendo ao *Composer*. Na figura 4.5 é possível visualizar um dos *workflows* criados, que simula um sistema que obtém valores de sensores de CO2 e, mediante a média, emite ou não um alerta. Foram utilizadas quatro funções *serverless* que foram *deployed* na *OpenWhisk*, sendo elas as seguintes: "getSensorList2", que obtêm uma lista de endereços HTTP para sensores através da invocação de um serviço *Rest* externo; "getCO2", que é executada dentro de um ciclo *"while"*, que permite que se percorra a lista dos sensores, e em cada uma das iterações a função recolhe o valor atual de CO2, adicionando-o a uma lista, capturado pelo sensor através de um pedido HTTP ao seu endereço; "calCO2Med", calcula a média dos valores de CO2, que se encontram na lista obtida da função anterior, passando esse valor como *output* da função; "sendAlert2", que é executada caso o *workflow* verifique que a média é superior a 300, e faz uma chamada a um serviço externo.

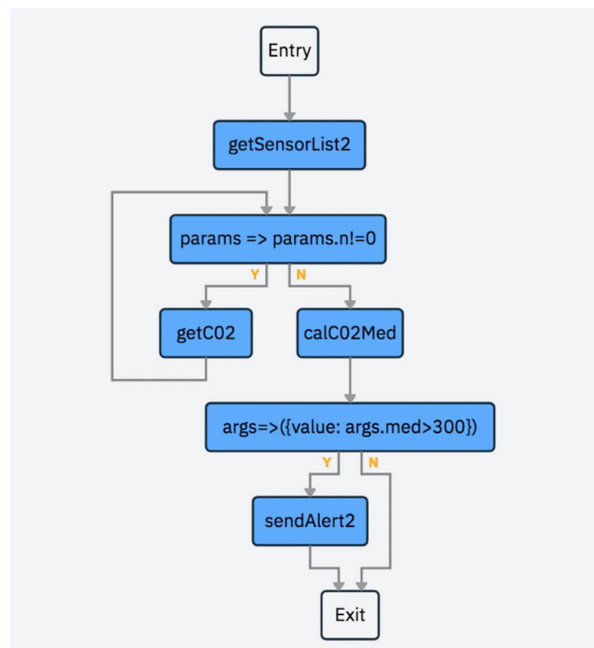


Figura 4.5: *Workflow serverless* criado recorrendo à *OpenWhisk* e à *IBM Composer*

[33] `https://www.vagrantup.com/intro/getting-started/`

[34] `https://www.virtualbox.org/`

4.5 *Picasso*

O *Picasso*[35] é um projeto de *Function-as-a-Service (FaaS)* para *OpenStack* cuja missão é fornecer uma API para executar *FaaS* em num ambiente de *cloud OpenStack*, abstraindo a camada de infraestrutura, permitindo simplicidade, eficiência e escalabilidade quer para programadores quer para operadores.

O *Picasso* é composto essencialmente por duas componentes principais. A API *Picasso* que utiliza o serviço de autenticação e autorização *Keystone*[36] da *OpenStack*; E a *Iron-Functions*[37], que é uma plataforma de código-aberto *Serverless* baseada em *Docker* e que foi utilizada pelo *Picasso* como motor de *backend* de *containers*.

É possível criar funções privadas ou públicas. As primeiras são definidas como pertencendo a um projeto *OpenStack* e a execução das mesmas requer que seja passada um cabeçalho de autorização nos pedidos *http* para aceder ao projeto. As segundas, apesar de também pertencerem a um projeto, não requerem qualquer cabeçalho de autorização para serem executadas, podendo ser partilhadas por qualquer utilizador.

Apesar de ser um projeto recente, recebeu o primeiro *commit* em 14 de Dezembro de 2016, parece já ter sido abandonado pois não recebeu qualquer atualização depois de 2 de Março de 2017. Aliado a este aparente abandono está também a fraca e quase inexistência documentação do mesmo.

4.6 *Qinling*

Qinling[38] é um projeto *OpenStack* de *FaaS*, que pretende fornecer uma plataforma de execução de funções *serverless* através da utilização de sistemas orquestração de *containers*, sendo suportados, através da utilização de mecanismo de *plugins*, o *Kubernetes*[39], o *Magnum*[40] e o *Docker Swarm*[41] [28].

As linguagens de programação admitidas para o desenvolvimento das funções são o *python* e o *node.Js* e estas podem ser invocadas, de forma síncrona ou assíncrona, através de *triggers* dos serviços da *OpenStack* ou através de invocações diretas de qualquer aplicação *web* ou *mobile*.

Relativamente à autenticação, o *Qinling* pode usar o serviço *Keystone*[42] da *OpenStack*, mas também pode ser usado sem qualquer tipo de autenticação.

A iteração com a plataforma pode ser feita através da *python-qinlingclient*, que é uma

[35] `https://wiki.openstack.org/wiki/Picasso`
[36] `https://docs.openstack.org/keystone/pike/`
[37] `https://github.com/iron-io/functions`
[38] `https://docs.openstack.org/qinling/latest/`
[39] `https://kubernetes.io/`
[40] `https://wiki.openstack.org/wiki/Magnum`
[41] `https://docs.docker.com/engine/swarm/`
[42] `https://docs.openstack.org/keystone/latest/`

interface de linha de comandos, ou através de pedidos HTTP realizados diretamente à *API RESTful*.

A plataforma garante uma escalabilidade automática que permite lidar com as alterações da carga de pedidos. Caso seja detetado um grande volume de requisições de execução de uma função, o número de *workers* (componentes que executam as funções) é aumentado num valor predefinido e após um tempo sem que estes sejam necessários, são libertados.

É um projeto bastante recente, tendo sido iniciado em 11 de Abril de 2017, e que continua bastante ativo continuando a receber atualizações. Estão sob desenvolvimento algumas funcionalidades como o suporte de versionamento do código e a capacidade de o utilizador definir a memória e o CPU para cada função. Contudo, a documentação ainda é bastante fraca o que torna o estudo profundo deste projeto complicado.

4.7 *OpenFaaS*

OpenFaaS[43] é uma plataforma *open source* para o desenvolvimento de funções *serverless* recorrendo à utilização da tecnologia de *containers Docker* e aos seus sistemas de orquestração *Kubernetes*[44] ou *Docker Swarm*[45]. Encontra-se ainda em desenvolvimento e em expansão de funcionalidades, recebendo frequentes atualizações no seu repositório[46] e contando com mais de dez mil estrelas no *GitHub*.

A plataforma mantém uma instância de *container* de cada função que tenha sido implementada, o que permite mitigar o tempo de latência de *cold start*, contudo esta abordagem apresenta um *overhead* no consumo de recursos, pois funções que não estejam a ser executadas continuam a consumi-los. A escalabilidade é obtida através da alteração do número de réplicas de serviço, que executam as funções, do *Docker Swarm* ou do *Kubernetes* de acordo com a carga dos pedidos [29] [30].

A *faas-cli* é uma interface de linha de comandos que permite criar funções, a partir de *templates*, e fazer o deploy destas na *OpenFaaS* de forma rápida. O utilizador apenas tem de escrever o seu código e a interface encarrega-se de criar a imagem *Docker* que irá ser usada para a execução [29]. As linguagens suportadas de forma nativa são *Node.js, Python* e *Go*. Contudo, é possível escrever código em qualquer linguagem e empacotá-lo num *container Docker* que será suportado pelo sistema [29]. A plataforma conta ainda com uma interface web visual que permite visualizar, criar e executar funções.

A API *Gateway* da *OpenFaaS* permite a criação de rotas de pedidos *HTTP* para as funções que são o único *trigger* suportado para as execuções. Estas podem ser síncronas ou assincronas. Permite ainda, em conjunto com o *Prometheus*[47], a recolha e visualização

[43] `https://www.openfaas.com/`
[44] `https://kubernetes.io/`
[45] `https://docs.docker.com/swarm/overview/`
[46] `https://github.com/openfaas/faas`
[47] `https://prometheus.io/`

de métricas relacionadas com a execução das funções [29].

Os casos de uso apresentados pela plataforma são os *backends web*, *mobile* e de *IoT*, *APIs HTTP*, *machine learning*, *batch jobs* e *chat bots*.

4.8 *Fission*

Fission[48] é uma *framework* de FaaS *open source*, sob licença *Apache*, mantida pela Platform 9[49] construída sob *kubernets*[50], que é um sistema de orquestração de *containers*. A plataforma é focada no aumento da produtividade dos programadores e na alta performance e pode ser instalada em qualquer dispositivo que contenha/suporte um *cluster Kubernetes*, desde uma *cloud* até um computador pessoal [31]. O utilizador apenas tem de escrever código e fazer o seu *deploy*, através do uso de uma interface de linha de comandos. O sistema é que se encarrega das configurações necessárias para que seja possível executar a função, abstraindo por completo a plataforma [32].

De forma nativa são suportadas funções escritas em *Python, Node.js, Go, Ruby, C# (.Net), Perl e PHP* e não existe, por omissão, um tempo limite para as suas execuções. Os *triggers* suportados para a ativação são pedidos *HTTP*, escalonamento baseado em tempo, e mensagens em formato *JSON* num tópico de mensagens [33].

Fission oferece escalabilidade automática das funções, de forma a lidar com aumentos de carga de pedidos de execução, baseando-se na utilização de *CPU* [34]. São mantidos um número configurável de *warm containers* de forma a que se tenham latências de *cold start* bastante baixas, na ordem dos 100ms. Esta velocidade é conseguida porque cada um dos *warm containers* inclui um carregador dinâmico que permite que quando uma função é invocada pela primeira vez, ou quando não existe uma instância ativa da função, um dos *containers* é escolhido e a função é rapidamente carregada [33]. Instâncias de funções que não estejam ativas são eliminadas após um tempo configurável, por omissão após 10 minutos de inatividade. Para casos em que se queira uma menor latência, o utilizador pode optar por definir um valor mínimo e máximo de instâncias de uma função. Caso o valor seja maior que zero, o sistema irá manter sempre esse número de instâncias activas, não estando estas sujeitas a eliminação por inactividade, contudo são consumidos recursos mesmo quando funções não estão a ser executadas [33].

Os *logs* são incorporados diretamente na interface de linha de comandos da plataforma, através da integração com o *Fluentd*[51] (que é um receptor de dados *open source* para a camada de *logging*). Para monitorização e visualização de métricas são suportadas as plataformas *Prometheus*[52], que permite a captação de métricas e a sua visualização, e

[48] `https://fission.io/`
[49] `https://platform9.com/`
[50] `https://kubernetes.io/`
[51] `https://www.fluentd.org/`
[52] `https://prometheus.io/`

Istio[53], que permite monitorizar a utilização das funções e a latência dos pedidos [32] [33].

Fission Workflows permite criar, de forma rápida e elegante, aplicações *serverless* complexas através da integração e orquestração de funções (tarefas) que podem ser executadas em paralelo e/ou sequência sendo ainda possível adicionar controlo dinâmico de fluxo com a utilização de decisões (condições "if") e ciclos. Os dados fluem pelo sistema, sendo passado como *input* de uma função o *output* da função anterior [31]. Os *workflows* são definidos em *YAML*, que é um strandard de serialização de dados de fácil interpretação para humanos. [33].

Os casos de uso apontados para a utilização da *Fission* são a criação de *APIs* de *backend* para aplicações *Web* e *mobile* e a implementação de *Webhooks*, em que a função é invocada quando um evento ocorre num serviço externo. Um exemplo de *Webhook* é o caso da deteção de certas mensagens ou palavras no *slack*. Alguns dos utilizadores da *Fission* são a *Cadence Design Systems*, a *Autodesk* e a *Cielo24* [31].

[53] `https://istio.io/`

Esta página foi propositadamente deixada em branco.

Capítulo 5

Comparação das plataformas *Serverless*

De forma a ter uma visão geral das plataformas *serverless* abordadas no capítulo 4, e para que seja possível compará-las em termos de funcionalidades e características, proponho um conjunto de parâmetros, que poderão também ser usados para que os utilizadores possam escolher a plataforma que melhor satisfaz os seus requisitos. Os parâmetros propostos são apresentados abaixo e foram escolhidos pois permitem resumir as diferenças de características e de funcionalidades entre as plataformas de computação *serverless*, de acordo com o estudo realizado às mesmas no capítulo 4. Alguns destes são também propostos por Baldini *et. al.* [16] como sendo as características que permitem distinguir as várias plataformas.

- Modelo de *deploy* da plataforma: Modelo de implementação da plataforma, que pode ser *open source*, proprietário e/ou público.
- Linguagens de programação suportadas: As linguagens de programação suportadas pelas plataformas e outros mecanismos que permitam a escrita de funções.
- Fontes de eventos de *trigger*: Provedores de eventos ou serviços que permitem através de *triggers* executar as funções que foram *deployed* na plataforma.
- Gestão de acessos: De que forma e com que dados é feito o controlo de acesso e execução de funções.
- Gestão de dependências: De que forma a plataforma gere as dependências do código das funções.
- Preço: O modelo de preços aplicado pela plataforma.
- Escalabilidade: De que forma a escalabilidade é oferecida pela plataforma, para que suporte as invocações.

- Máximo número de funções: Máximo número de funções que um utilizador pode fazer *deploy* na plataforma.
- Máximo número de execuções concorrentes: Numero máximo de execuções de uma determinada função que podem estar a ocorrer em paralelo.
- Máximo tempo de execução: Tempo durante o qual uma função que tenha sido invocada pode estar ativa em execução.
- Tamanho máximo do código: Tamanho máximo que o ficheiro, ou pacote, que contém o código pode ter.
- Memória máxima: Memória máxima que cada função pode utilizar na sua execução.
- Modos de *deploy* de código: Formas que podem ser utilizadas para o *deploy* do código das funções na plataforma.
- *Logging*: Se a plataforma suporta gestão de *logs* ou se utiliza alguma ferramenta para o efeito.
- Workflows: Sistemas de *workflows serverless* ou outros sistemas de orquestração de funções suportados.
- Invocações HTTP: Se a plataforma suporta invocações de funções através de pedidos HTTP ou se suporta outros serviços que permitam este tipo de invocações.
- Monitorização: Ferramentas suportadas pelas plataformas que permitem monitorizar as funções e suas execuções.
- Serviços de armazenamento: Serviços com os quais a plataforma integra ou interage de forma a poder armazenar o estado das funções ou qualquer tipo de dados que o utilizador pretenda.

Na tabela 5.1 é exposta a comparação das plataformas *serverless* apresentadas no capítulo 4, recorrendo aos parâmetros acima indicados. O *Picasso* e o *Qinling* não são considerados nesta comparação, pois, por serem projetos recentes e por terem pouca documentação, seria difícil obter informação necessária à sua inclusão.

Caraterística	*AWS Lambda*	*Microsoft Azure Functions*	*Google Cloud Functions*	*OpenWhisk /IBM Cloud Functions*	*OpenFaaS*	*Fission*
Modelo de deploy da plataforma	Proprietário, cloud pública	Proprietário, cloud pública	Proprietário, cloud pública	Open Source em qualquer sistema que suporte Docker/Cloud pública	Open Source e instalação em qualquer cluster Kubernetes ou Docker Swarm	Open Source e instalação possível em qualquer cluster Kubernetes
Linguagens de programação suportadas	Node.js, Python, Java, C# e Go	Node.Js, C#, F# e Java	JavaScript	Node.js, Swift, Java, Go, PHP e Python e containers *Docker*	Node.js, Python, Go, e containers Docker	Python, Node.js, Go, Ruby, C# (.Net), Perl, PHP e containers Docker
Fontes de eventos de *Trigger*	Amazon DynamoDB, Amazon Kinesis Data Streams, Amazon S3, API Gateway, Amazon SNS e Amazon CloudWatch	Azure Cosmos DB, Azure Event Hubs, Azure Event Grid, Azure Mobile Apps, Azure Notification Hubs, Azure Service Bus, Azure Storage, GitHub, HTTP e triggers baseados em tempo	Cloud Pub/Sub, Cloud Storage e pedidos HTTP	GitHub, IBM Cloudant noSQL DB, IBM Mobile Push, IBM Message Hub, IBM Push Notifications e Slack e outras fontes criadas pela comunidade	Apenas suporta triggers HTTP	Triggers baseados em tempo e agendamento, triggers de queues de mensagens (nats-streaming e azure-storage-queue) e triggers HTTP
Gestão de acessos	AWS IAM	Azure IAM	Google IAM	Autenticação básica através de contas locais na plataforma/ IBM IAM	Não suporta	Não suporta

Gestão de dependências	Incluídas no pacote de deploy	NPM e NuGet	NPM	Incluídas no pacote de deploy e NPM	Incluídas no pacote de deploy	Incluídas no pacote de deploy
Preço	1 milhão de execuções e 400.000 GB/segundo gratuitos por mês e depois disso 0,20USD por cada 1 milhão de execuções e 0,00001667USD por cada GB/segundo	1 milhão de execuções e 400.000 GB/segundo gratuitos por mês e depois disso 0,20USD por cada 1 milhão de execuções e 0,000016USD por cada GB/segundo	1 milhão de execuções e 400.000 GB/segundo gratuitos por mês e depois disso 0.4USD por cada 1 milhão de execuções, 0.0000025USD por cada GB/segundo e 0.00001 por GHz/s	Sem preço/ 0.000017USD por GB/s, sendo que os primeiros 400,000GB/s por mês são gratuitos	Sem preço	Sem preço
Escalabilidade	Automática não configurável	Automática ou manual de acordo com o plano escolhido	Automática	Automática	Automática configurável, sendo possível manter uma instância de cada função sempre ativa	Automática configurável, permite definir o máximo e mínimo de instâncias
Máximo número de funções	Ilimitado	Ilimitado	1000 por projeto	Ilimitado	Ilimitado	Ilimitado
Máximo número de execuções concorrentes	1000	Ilimitado	Ilimitado	1000 por utilizador	Ilimitado	Ilimitado
Máximo tempo de execução	300 segundos	600 segundos	540 segundos	600 segundos	Ilimitado mas configurável	Ilimitado

Tamanho máximo do código	50MB por pacote de deploy compactado, 250MB descompactado	Ilimitado	100MB para código compactado e 500MB para código descompactado e para módulos	48MB por função	Ilimitado	Ilimitado
Memória máxima	3008MB por função	1536MB	2048MB	512MB	Ilimitada, depende da infraestrutura	Ilimitada, depende da infraestrutura
Modos de *deploy* de código	Upload de pacote de *deploy* diretamente para a plataforma ou para o S3 ou através do editor da interface web	Editor online ou através do GitHub,do BitBucket e do Visual Studio Team Services	Upload de um ficheiro *zip* ou ainda através da *Google Cloud Source Repositories*	Ficheiros de código através da utilização da CLI ou no editor online da IBM	Usando templates através da faas-cli ou através da interface web	Ficheiros com o código ou ficheiro .zip através da CLI
Logging	AWS CloudWatch Logs	App Services monitoring	Stackdriver Logging	Suporta loggs de forma nativa	Não suporta	Através da utilização do Fluentd
Workflows	AWS Step Functions	Logic Apps	Não suporta	IBM Composer	Não suporta	Fission Workflows
Invocações HTTP	AWS API Gateway	Suporta de forma nativa	Suporta de forma nativa	Suporta de forma nativa	Suporta de forma nativa	Suporta de forma nativa
Monitorização	CloudWatch e X-Ray	Application Insights	Stackdriver Monitoring	IBM Cloud Functions Dashboard	Através da integração com o Prometheus	Através da integração com o Prometheus e com o Istio

Serviços de armazenamento	Amazon DynamoDB e Amazon SimpleDB e S3	Azure Cosmos DB e Azure Storage	Cloud Storage, Cloud Firestore e Cloud Datastore	IBM Cloudant	Não suporta	Não suporta

Tabela 5.1: Comparação de plataformas *Serverless*

Analisando a tabela é possível retirar algumas conclusões, das quais se destacam as seguintes:

- A única linguagem de programação, suportada por todas as plataformas *serverless* abordadas, é *JavaScript (Node.js)*, sendo esta a única aceite pela *Google Cloud Functions*. *Java* é admitida pelas restantes três plataformas comerciais. Nas opções *open source* é possível a utilização direta de *containers* para que sejam suportadas quaisquer linguagens de programação para escrita do código.
- Como fontes de eventos de *triggers*, as quatro alternativas proprietárias oferecem integração com uma vasta gama de serviços das *clouds* em que estão inseridas, enquanto que nas opções *open source* a *OpenWhisk* apresenta uma maior oferta que as restantes duas.
- Na *OpenFaaS* e na *Fission* não existe qualquer controlo de acesso e de execução de funções, ao contrário das restantes plataformas, em que este é conseguido através dos serviço de gestão de acessos dos respetivos provedores de *cloud*, sendo que, a *OpenWhisk* suporta contas que tenham sido criadas na própria plataforma.
- A escalabilidade é oferecida de forma automática por todas as plataformas, havendo a possibilidade de ser definido um número mínimo de instâncias, de cada função, que se mantêm ativas na *Fission* e na *OpenFaaS*.
- Relativamente ao tempo máximo de execução de uma função a *AWS Lambda* é a mais limitada, sendo que não existe qualquer restrição na *Fission* e na *OpenFaaS*.
- Para *logging* apenas a *OpenFaas* não suporta qualquer tipo de ferramenta ou serviço. Já a monitorização é conseguida em todas as plataformas através de integração com serviços ou ferramentas.
- *Worflows* são possíveis na *AWS Lambda*, *Azure Functions*, *OpenWhisk*, e *Fission*, recorrendo-se respetivamente à *AWS Step Functions*, *Logic Apps*, *IBM Composer* e *Fission Workflows*.

Das plataformas proprietárias, oferecidas por *clouds* públicas a *AWS Lambda* parecer ser a mais poderosa em termos de integração com outros sistema, devido ao grande ecossistema que a *AWS* disponibiliza. Contudo, é também a que tem mais restrições relativamente ao número de execuções e ao tempo que cada uma pode ocupar. A *Google Cloud Functions* é a mais pobre, por ser a mais recente e por se encontrar também em versão *beta*. Uma das suas lacunas mais relevantes é a falta se um sistema de *workflows*. Das opções *open source*, a *OpenWhisk* é possivelmente a mais completa, beneficiando bastante da ferramenta de *workflows IBM Composer*. Por ser disponibilizada na *cloud* pública da IBM, poderá beneficiar duma mais constante atualização, melhoria em termos de performance e expansão do ecossistema, como se pode comprovar pela frequente atualização do seu repositório no *github*[1].

[1] `https://github.com/apache/incubator-openwhisk/commits/master`

Esta página foi propositadamente deixada em branco.

Capítulo 6

Benchmarking de plataformas *Serverless*

Para além das características e funcionalidades abordadas no capítulo anterior, as plataformas de computação *serverless* podem apresentar performances distintas entre si, e, por isso, pode ser complicado para um utilizador perceber qual a que melhor se encaixa nas suas necessidades e nos seus requisitos. Neste capítulo, serão apresentados estudos e *benchmarkings* existentes de comparação de plataformas *serverless*. Face às lacunas encontradas será proposto um conjunto de testes para um *benchmark* que irá ser usado para comparação das plataformas apresentadas no capítulo 4 e, por fim, será apresentada uma arquitetura de uma aplicação que irá permitir automatizar esses *benchmarkings*.

6.1 *Benchmarkings* existentes

Nesta secção serão apresentados alguns estudos de performance e *benchmarkings* realizados a plataformas *serverless*, sendo apresentado na tabela 6.1 um resumos destes.

McGrath e Garret [17] [18] apresentam a arquitetura de uma nova plataforma focada na performance de computação *serverless*. Esta foi implementada e comparada, em Março de 2017, com as plataformas comerciais existentes, *AWS Lamba*, *Azure Functions*, *Google Cloud Functions* e *Apache OpenWhisk*. Os autores desenvolveram uma ferramenta que possibilitou a realização dos testes que permitiram a comparação, tendo sido utilizada uma função simples, que completa a execução imediatamente e retorna, alocada com 512MB de memória, que é invocada de forma síncrona através de um *trigger* HTTP nas várias plataformas.
Foram modelados dois testes para avaliação de performance, o teste de concorrência e o teste de *Backoff*. O primeiro foi desenhado para medir a capacidade das plataformas *serverless* invocarem com eficiência uma função em escala e consiste na realização de pedidos em sequência (reemite cada pedido imediatamente depois de receber a resposta do pedido anterior) com vários níveis de concorrência, incrementais até ao máximo de

15, medindo-se o número de respostas recebidas por segundo, ou seja, o *throughput* por segundo. Os resultados deste teste mostram que o protótipo dos autores apresenta o maior *throughput* por segundo. A *AWS Lambda* é a plataforma comercial que apresenta melhores resultados com 15 pedidos concorrentes, escalando linearmente. A *Google Cloud Functions* exibe um escalonamento sub-linear que tende a estabilizar quando os pedidos concorrentes se aproximam do 15. A *Azure Functions* apresenta um comportamento bastante variável, apresentando valores de *throughput* bastante altos para níveis de concorrência baixos, mas tendo depois um comportamento aleatório. Por fim, a *OpenWhisk* apresenta um valor de *throughput* bastante baixo até ao nível de concorrência de 8, passando depois a escalar de forma sub-linear. Os autores apresentam uma possível causa para este comportamento, que passa pelo facto da plataforma *OpenWhisk* lançar vários *containers* antes de começar a reutilizá-los, isto é, antes de usar os *warm containers*.
O segundo teste proposto, chamado de teste de *Backoff*, pretende estudar o tempo de *cold start* das instâncias das funções, assim como o tempo que durante o qual os *warm containers* se mantêm ativos nas várias plataformas. Para isto, são efetuados pedidos de execução da função espaçados por um tempo de intervalo incremental, variado entre 1 e 30 minutos, medindo-se a latência da resposta. Os resultados demonstram que o protótipo e a *Azure functions* apresentam os piores tempos de latência em situação de *cold start*, mantendo os *warm containers* ativos por cerca de 15 minutos após a ultima execução. A *OpenWhisk* liberta os *warm containers* após 10 minutos, mas apresenta muito melhores tempos de *cold start* que as duas plataformas anteriores. A *AWS Lambda* e a *Google Cloud Functions*, pelos resultados apresentados, parecem não ser afetadas pelo *cold start* podendo, segundo os autores, este comportamento se dever a um tempo de inicialização de *containers* extremamente rápido, ou à existência de uma pré-alocação dos mesmos. Este estudo apresenta resultados bastante interessantes, contudo, seria importante no teste de concorrência se ter aumentado um pouco mais o nível de concorrência, de forma a poder-se confirmar o comportamento apontado, pelos autores à *OpenWhisk* e à *Google Cloud Functions*. Para além disto, no teste de *Backoff* deviam ter sido usados tempos de intervalo entre invocações superiores, para que se percebesse efetivamente se no caso da *AWS Lambda* e da *Google Cloud Functions* os tempos de *cold start* estão a ser afetados por algum tipo de pré-alocação.

Ribenzaft [35] realizou um estudo de performance na *AWS Lambda* para identificar, numa função recursiva que calcula a sequência de *Fibonacci* de 25, escrita em *Python*, qual o nível de memória alocada, entre 128MB e 3008MB, que permite obter uma melhor otimização a nível do preço. Relembrando que nesta plataforma a memória alocada afeta proporcionalmente o CPU, a largura de banda e o I/O alocado, tal significa que uma maior alocação de recursos permite na teoria uma maior rapidez na execução e um maior preço por tempo de execução. O *script* utilizado para a realização dos testes garante que não existe tempo de *cold start* que possa influenciar o estudo. Os resultados obtidos demonstram que o aumento da memória alocada, leva de facto, a uma redução do tempo de execução. Contudo para o teste executado pelo autor, apenas compensa a alocação de 2048MB, pois, a partir desse valor, a redução do tempo de execução não compensa

o aumento do custo. O preço aumenta apesar de continuar a existir uma melhoria na latência. Seria interessante o teste ser realizado com vários n's da sequência de *Fibonacci* de forma a poder-se testar se, para funções mais pesadas computacionalmente, o aumento de memória pode levar ou não a maiores reduções de custo.

John Chapin [36] apresenta um artigo em que aborda a dificuldade de prever a performance da *AWS Lambda*, especialmente para funções que são alocadas com baixa memória, e refere que não é suficiente usar apenas algumas invocações para prever o comportamento da plataforma. O autor coloca a questão se efetivamente a *Lambda* escala de forma proporcional às configurações de memória, e, se não, como é que a performance difere do esperado. De forma a dar uma resposta, é realizada uma experiência usando uma função de *Fibonacci* recursiva, escrita em *Java*, que foi implementada na plataforma com 7 níveis de memória distintos (128, 256, 512, 768, 1024, 1280 e 1536 MB). Invocada através de *triggers* da *CloudWatch*, que permitem que a função seja executada a cada quatro minutos. A experiência foi realizada durante 48 horas, sendo que cada função foi invocada 720 vezes, e foram medidos os tempos de duração de cada execução através das métricas da *CloudWatch*. Os resultados, apresentados sob a forma de gráfico, mostram que a escalabilidade não é proporcional ao nível de memória. Por vezes, é possível obtermos a mesma performance com uma função alocada com 128MB que teríamos com uma alocada com 1536MB. Outras vezes, temos tempos de execução da primeira que são 12 vezes superiores ao da segunda. O autor refere, após analisar os resultados, que a performance de escalabilidade documentada geralmente representa o pior cenário possível para um determinado nível de alocação de memória, e, por isso, no pior caso as funções escalam de acordo com a configuração de memória. O estudo apresenta resultados bastante interessantes e poderia ser aplicado às restantes plataformas de forma a verificar-se como se comportam com a variação da memória.

Kaviani e Maximilien [37] sugerem metodologias e um conjunto de testes que podem vir a servir como standard para o *benchmark* de plataformas *serverless*. Para a caracterização da performance *serverless* são consideradas quatro classes de testes que variam de acordo com a categoria das funções: Funções com uso intensivo de CPU; Funções que necessitem de memória intensiva; Funções que necessitem de aceder a bases de dados; Funções que necessitem de comunicação através da rede. São também apresentadas três variáveis que podem ser usadas para replicar instâncias de cada classe de testes, de forma a que seja possível obter um vasto conjunto de testes e resultados para a obtenção de conclusões sobre a performance. As variáveis são: A variação das invocações (que podem ser aleatórias, com picos, ou periódicas); O tamanho do *payload* (que é o tamanho do input e do output das funções); Nível de concorrência (que é o número de funções que são executadas em paralelo). São realizadas algumas experiências com funções de alto e baixo *throughput*, com funções de uso intensivo de CPU e de memória e é analisado o comportamento da gestão dos *containers*. Os autores apresentam alguns resultados preliminares da execução das experiências nas plataformas *OpenWhisk*, *Azure Functions* e *AWS Lambda*. Contudo, não apresentam qualquer comparação entre elas. O trabalho elaborado pelos autores parece ser bastante ambicioso, mas o artigo que iniciaram não se encontra ainda terminado.

Simon Shillaker [38] apresenta uma experiência realizada numa instalação local da *Apache OpenWhisk*, com 2 *invokers*, suportando cada um 64 *containers*, e um *controller*. A plataforma é carregada com pedidos de execução com 4 níveis de concorrência distintos, sendo eles 1, 5, 25 e 50 execuções paralelas, e são medidos o *througput* por segundo e a latência associada. Dos resultados obtidos, pode-se retirar que sob carga baixa se observa alta latência, pois os *containers* são instanciados, usados e eliminados em todas as invocações. À medida que o *througput* aumenta a latência também aumenta, sendo que, quanto maior for *o throughput* menos crescerá a latência para um numero de execuções concorrentes maior, pois existiram mais *warm containers* ativos. Estes resultados permitem demonstrar que quanto maior for a carga menor tenderá a ser a latência, pois a reutilização de *containers* existe com maior probabilidade. Contudo, caso a carga seja demasiado alta, os *invokers* podem atingir o seu limite, não conseguindo escalar mais, levando a que a plataforma evite a utilização de *warm containers* o que pode causar um agravamento da performance. Este estudo apesar de apresentar alguns resultados, poderia ser mais importante se existissem mais níveis de concorrência, e não apenas os 4 apresentados. Também poderia ter sido retirada informação sobre os padrões de utilização de *containers*, como por exemplo a partir de que nível de concorrência eles começam a ser reutilizados.

Casalboni [39] realizou testes de carga na *AWS Lambda* e na *Google Cloud Functions*, utilizando uma função que gera 1000 *hashes md5* por cada invocação e uma carga incremental linear de cinco minutos, que atinge no máximo 70 invocações concorrentes por segundo. Os resultados mostram que há uma diferença notável no tempo de resposta das plataformas. Enquanto que os tempos de resposta da *Google Cloud Functions* se mantêm entre os 130 e os 200ms, com um aumento durante os minutos finais do teste para mais de 400ms, a *AWS Lambda* apresenta valores entre os 400 e os 600ms, sendo que nos minutos finais do teste este valor tende a decrescer. Visto que cada invocação da função retorna um JSON relativamente pesado, o autor assume que a performance da rede pode ter um impacto grande nos tempos obtidos e realiza novos testes usando uma função que apenas retorna uma mensagem com "OK". Ele refere que com esta modificação obteve melhoria nos tempos, passando a *AWS Lambda* a apresentar valores entre 200 e 300ms e a *Google Cloud Functions* valores de cerca de 100ms o que lhe permitiu concluir que a ligação de rede na *cloud* da *Google* funciona melhor e assume que a integração de *triggers* HTTP de forma nativa pode ser mais rápida quando comparada com a utilização da *API Gateway* da *AWS*. Podemos concluir através deste estudo que o tamanho do *payload* pode também afetar a performance destas plataformas, e seria interessante ter uma avaliação mais profunda neste tópico.

Parezan [40] realiza o mesmo teste que Casalboni, apresentado acima utilizando apenas a primeira função (função que gera 1000 *hashes md5*), mas para a plataforma *Azure Functions*. Os tempos de resposta obtidos compreendem-se entre os 80ms e os 600ms e exibem um comportamento algo aleatório, existindo alguns picos. Não é apresentado pelo autor nenhum possível motivo para o comportamento o que torna o estudo pouco relevante.

Yan Cui [41] apresenta um estudo realizado sobre a *AWS Lambda* utilizando diferentes linguagens de programação suportadas pela plataforma, de forma a verificar se diferentes linguagens podem oferecer performances distintas. O teste realizado envolve a utilização da *API Gateway* como *trigger* para a invocação, através de pedidos HTTP, de uma simples função *Lambda*, (que apenas retorna um "Hello") criada recorrendo a *Node.js, Java, C#* e *Python* e alocada com 1024MB de memória. A experiência foi realizada correndo a função em cada linguagem durante 1 hora e simula 10 pedidos concorrentes a serem enviados a cada segundo. Os resultados obtidos permitem verificar que a função escrita em *C#* apresenta tempos de respostas consistentemente e consideravelmente superiores às restantes e que a de *Java* é a que apresenta menores tempos e a que sofre menos variação entre as várias invocações. Este teste permitiu verificar que existem diferenças de performance para diferentes linguagens de programação, na *AWS Lambda*, e por isso, deve ser implementado noutras plataformas, de forma a averiguar se nestas, diferentes linguagens apresentam também diferentes performances.

Nolet [42] apresenta um *benchmark* de performance na comparação de uma função, que calcula os primeiros 30 números da sequência de *Fibonacci*, escrita em *Go* e em *Node.js*. Para o teste, as duas funções foram *deployed* na *AWS Lambda*, com uma alocação de 128MB de memória para cada, e foi utilizada a *API Gateway* como *trigger* de invocação. Foram feitas 1000 invocações de cada uma, a uma taxa de 10 por segundo e os resultados não permitem retirar qualquer conclusão sobre qual das linguagens permite obter uma melhor performance. O utilizador optou por realizar mais um teste que represente um cenário de utilização real, de processamento de ficheiros. Criou então uma função, nas mesmas duas linguagens, que tinha como objetivo recolher uma imagem da *Amazon S3* e escrever um *timestamp* numa tabela da *Amazon DynamoDB*. O teste foi realizado no mesmo modelo que o anterior e os resultados mostram que para este caso a função escrita em *Node.js* apresenta uma performance média duas vezes pior que a implementada recorrendo a *Go*. Este estudo acaba por ser bastante interessante, mostrando que para uns casos não é possivel tirar qualquer conclusão, mas para outros é possivel ver grandes alterações de performance, e por isso, as generalizações em performance de plataformas *serverless* podem nem sempre ser verdadeiras.

Billock [43] compara a performance da *AWS Lambda*, da *Google Cloud* e da *Azure Functions*, focando-se no tempo de resposta. Escreveu em *Node.js*, por ser uma linguagem transversal às 3 plataformas, uma função simples que apenas realiza uma concatenação de frases. O teste foi executado, recorrendo a um script de automatização, invocando a função em cada plataforma 10000 vezes, guardando o tempo de resposta de cada pedido e o numero de pedidos que foram completos com sucesso ou com erro. A cada 100 pedidos foi adicionada uma pausa de 1 a 1200 segundos de forma a poder testar a performance da utilização de *Hot/Warm* e *Cold containers*. Os resultados mostram que, em média, os tempos de execução são melhores na *Azure Functions*, contudo, esta apresenta uma taxa de falhas superior às restantes plataformas. A *Google Functions* apresenta resultados mais distribuídos entre os modos *Hot* e *Cold* o que torna o seu desempenho difícil de prever ao contrário da *AWS Lambda* em que os tempos estão bem organizados nesses dois modos.

Autor(es)	Plataformas testadas	Objetivo do estudo	Descrição do estudo	Resultados
McGrath e Garret [17] [18]	Protótipo (focado em performance), *AWS Lambda*, *Azure Functions*, *Google Cloud Functions* e *Apache OpenWhisk*	Apresentam o seu protótipo e uma ferramenta que permite a realização de dois testes: Um teste de concorrência para medir a capacidade das plataformas invocarem com eficiência funções em escala e um teste de *Backoff* para estudar o comportamento das plataformas face à reutilização de *containers* (tempo em que ficam em estado *warm* e impacto dos *cold containers*)	Utilizaram um função simples (apenas faz um retorno), alocando-lhe 512MB de memória que é invocada através de um *trigger* HTTP. O teste de concorrência é realizado efetuando invocações de funções durante um período de tempo aumentando o nível de concorrência até 15 medindo o *throughput* por segundo obtido. No teste de *Backoff* são realizadas invocações da função de forma sequência espaçadas por um tempo incremental entre 1 e 30 minutos registando a latência de resposta	No teste de concorrência a *AWS Lambda* apresenta os melhores resultados escalando de forma linear. A *google* apresenta uma escalabilidade sub-linear, tendendo a estabilizar para valores de concorrência próximos de 15. A *azure* apresenta um comportamento aleatório. A *OpenWhisk* apresenta um *throughput* baixo até ao nível de concorrência 8 passando depois a escalar de forma sub-linear. Para o teste de *Backoff* a azure apresenta os piores tempos em situação de *cold start*, mantendo os *warm containers* por cerca de 15 minutos. No caso da *OpenWhisk* os *containers* são libertados após cerca de 10 minutos. A *AWS* e a *Google* parecem não ser afetadas pelo *cold start*.
Ribenzaft [35]	*AWS Lambda*	Apresenta um teste de performance efetuado para que possa identificar qual o nível de memória alocada que lhe permite obter uma melhor otimização a nível de preço	Na *AWS Lambda* a memória alocada afeta proporcionalmente também o CPU, a largura de banda e o I/O. É utilizada uma função recursiva que calcula a sequência de *Fibonacci* de 25 alocada com diferentes níveis de memória entre 128MB e 3008MB. É garantido que não existem *cold starts* a influenciar o estudo.	Os resultados obtidos demonstram que quanto for maior a quantidade de memória alocada menor será o tempo de execução. No teste efetuado pelo autor apenas compensa alocar memória até 2048MB pois a partir desse valor o aumento do custo não compensa a redução no tempo de execução.

John Chapin [36]	*AWS Lambda*	Apresenta um artigo em que aborda a dificuldade em prever a performance da *AWS Lambda* e questiona se efetivamente a plataforma escala de forma proporcional às configurações de memória. De forma a dar resposta é realizada uma experiência.	Na experiência realizada foi utilizada uma função de *Fibonacci* recursiva, em *Java*, que foi implementada na plataforma com 7 níveis de memória alocada entre 128 e 1536MB. Foram realizadas 720 invocações utilizando-se um *trigger* da *cloudwatch* e registado o tempo de execução de cada invocação.	Os resultados foram apresentados sob forma de gráfico e mostram que nem sempre a escalabilidade é proporcional à memória alocada, sendo este facto só verdade para o pior cenário possível para cada nível de alocação de memória.
Kaviani e Maximilien [37]	*Apache OpenWhisk*, *AWS Lambda* e *Azure Functions*	Sugerem metodologias e um conjunto de testes que podem vir a servir como standard para o bechmark de plataformas *serverless* e apresentam alguns resultados preliminares.	Para a caracterização da performance são consideradas quatro classes de testes, que variam de acordo com a categoria da função: Funções com uso intensivo de CPU; Funções que necessitem de memória intensiva; Funções que necessitam de base de dados; Funções que comunicam através da rede. São ainda apresentadas 3 variáveis para que seja possivel replicar e variar as classes de testes: A variação das invocações; O tamanho do *payload*; O nível de concorrência.	Apenas foram apresentados alguns resultados preliminares na execução das experiências, sendo que não foi apresentada qualquer comparação entre as plataformas. O artigo que iniciaram não foi ainda terminado.
Simon Shillaker [38]	Instalação local do *OpenWhisk*	Experiência realizada numa instalação local do *Openwhisk* de forma a verificar como este se comporta com níveis de concorrência distintos.	*Openwhisk* local com dois *invokers* e um *controller*. A plataforma foi carregada com invocações de função com 4 níveis distintos de concorrência: 1, 5, 25 e 50 execuções paralelas. Registo de latência e *throughput* por segundo.	Com baixa carga é observada uma alta latência, pois não existe reutilização de *containers*. Quanto maior for a carga menor tenderá a ser a latência, pois a probabilidade de utilização de *warm containers* será maior. Se a carga for demasiada pesada, os *invokers* atingem o seu limite e a plataforma evita a reutilização de *containers* levando a uma degradação de performance.

Casalboni [39]	*AWS Lambda* e *Google Cloud Functions*	Apresenta resultados de um teste de concorrência com dois tipos de funções nas duas plataformas	O teste foi efetuado realizando a carga de invocação das funções de forma incremental linear de 5 minutos, atingindo o máximo de 70 invocações concorrentes por segundo. Numa primeira experiência fui utilizada uma função que gera 1000 *hashes md5* e posteriormente foi utilizada uma função que apenas retorna uma pequena mensagem medindo em ambas o tempo de resposta .	De ambas as experiências o autor retira que a rede ou a integração nativa de *triggers* HTTP na *google* funcionam melhor e mais rapidamente quando comparada com a utilização da integração da *AWS API Gateway*.
Parezan [40]	*Azure Functions*	Apresenta resultados de um teste de concorrência	O teste foi efetuado realizando a carga de invocação das funções de forma incremental linear de 5 minutos, atingindo o máximo de 70 invocações concorrentes por segundo. Foi utilizada uma função que gera 1000 *hashes md5* e foi registado o tempo de resposta.	O tempos de resposta apresentados variam de forma aleatória apresentando vários picos, sendo que não é dada nenhuma possível explicação.
Yan Cui [41]	*AWS Lambda*	Estudo do impacto das linguagens de programação, suportadas pela plataforma, na performance	O teste foi realizado utilizando pedidos HTTP como *triggers* e foi utilizada uma função simples, alocada com 1024MB de memória, que retorna uma palavra criada recorrendo a *Node.js, Java, C#* e *Python.* A experiência foi realizada através de invocações para cada linguagem durante uma hora e simulando-se 10 pedidos concorrentes por segundo, registando-se para cada um o tempo de resposta.	A função escrita em *C#* apresenta tempos consistentemente e consideravelmente superiores. *Java* apresenta os tempos mais consistentes e ainda os mais baixos.

Nolet [42]	*AWS Lambda*	*Benchmark* de performance na comparação de uma função escrita em diferentes linguagens	Para invocação foi utilizado o *trigger* HTTP e as funções foram alocadas com 128MB de memória. Foram realizadas 1000 invocações a uma taxa de 10 por segundo. Num primeiro teste foi utilizada uma função em *Go* e em *Node.js* que calcula os primeiros 30 números da sequência de *Fibonacci*. Num segundo teste foram utilizadas as mesmas linguagens numa função que interage com a *Amazon S3* e a *Amazon DynamoDB*	Para o primeiro teste não foi possível tirar qualquer conclusão de qual a linguagem que apresenta uma melhor performance. Já para o segundo teste a função escrita em Node.js apresenta uma performance média 2 vezes pior que a escrita em Go.
Billock [43]	*AWS Lambda*, *Google Cloud Functions* e *Azure Functions*	Comparação de performance entre as três plataformas baseando-se no tempo de resposta e comparação da performance na utilização de *warm/hot* e *cold containers*	No teste que permitiu a comparação foi utilizada uma função, escrita em *Node.js*, que realiza uma concatenação de frases. Foi utilizado um *script* que invoca a função de cada plataforma 10000 vezes, guardando o tempo de resposta de cada, assim como o numero de pedidos que foram executados com sucesso e com insucesso. De forma a testar a utilização de *cold* e *hot containers* a cada 100 pedidos de invocação foi adiciona uma pausa entre 1 e 1200 segundos.	Os resultados apresentados mostram que em média os tempos de resposta são menores na *Azure*, contudo a taxa de invocações que obtiveram falha é superior às restantes plataformas.

Tabela 6.1: Resumo dos estudos de performance e *benchmarkings* apresentados

Apesar de já existirem alguns estudos e testes de performance propostos e realizados, nenhum deles é suficientemente abrangente e completo que possa ser considerado um *benchmark* amplo de plataformas *serverless*. Existe assim, uma lacuna na definição e realização de um conjunto de testes para um *benchmark*, que permita a análise e comparação de performance, entre diversas plataformas, em múltiplas vertentes. Também não existe uma aplicação que permita a automatização na realização desses testes. Neste sentido existe um espaço para a proposta de uma suite de testes ampla para um *benchmark*, de uso genérico e para a definição e implementação de uma aplicação de automatização da realização dos testes que constituem a suite.

6.2 Proposta de *benchmark* de plataformas *serverless*

Esta secção tem como objetivo apresentar uma proposta de *benchmark* de plataformas *serverless* para uso genérico, em que qualquer pessoa que queira, possa executar o *benchmarking*. Este executará um conjunto de testes, definidos na sub-secção 6.2.1, de forma a avaliar e comparar o desempenho e performance das plataformas em múltiplas vertentes e será também modular, para que se possam adicionar, alterar e modificar testes de forma fácil. Para a automatização do processo de execução, recolha e apresentação de resultados dos testes, será desenvolvida uma aplicação, cuja arquitetura se encontra descrita na sub-secção 6.2.2. Numa primeira fase, o *benchmark* foi realizado às plataformas comercias *AWS Lambda*, *IBM Cloud Functions*, *Microsoft Azure Functions* e *Google Cloud Functions*, que são disponibilizadas como serviço nas respetivas *clouds*. Numa segunda fase, foi realizada uma instalação local da plataformas *open Source Apache OpenWhisk* na qual foi corrido o *benchmark*. Uma vez que a *OpenWhisk* e a *IBM Cloud Functions* usam a mesma plataforma, a comparação entre elas permite avaliar o impacto da infraestrutura nos seus desempenhos.

6.2.1 Propostas de testes para o *benchmark*

O *benchmark* vai conter um conjunto de testes que estão definidos e explicados nas tabelas seguintes. Para cada teste está definido o seu nome, um ID, o seu objetivo e comentários/informação adicional, cujo objetivo é descrever de forma mais profunda como deve ser efetuado e o motivo de ser realizado. Os testes vão incidir sobretudo na latência (tempo que demora desde o pedido de invocação ser feito, até que se começa a receber a resposta) e no *throughput* (numero de pedidos de invocação de funções que o sistema consegue satisfazer por unidade de tempo), pois são duas métricas que são possíveis de recolher em todas as plataformas *serverless* e permitem modelar bem o seu comportamento e performance em diferentes cenários. Vão ser também medidas a taxa de sucesso e de falha de execuções em cada teste. Estes dados vão ser utilizadas também no processo de comparação, pois, em algumas situações, a plataforma pode não conseguir executar uma função, por exemplo, devido a uma carga demasiado elevada. Os testes vão ainda

compreender diversos cenários, nomeadamente, de reutilização de *containers*, de diferentes níveis de carga concorrente, de tamanho de *payload* distintos, de utilização de diferentes linguagens de programação e de valores díspares de memória alocada.

Nome do teste:	Teste de *overhead* das plataformas *serverless*
ID do teste:	T1
Objetivo:	Medir a latência de uma sequência de pedidos de invocação de uma função de forma a obter e comparar o *overhead* de cada plataforma *serverless*.
Comentários/ Informação adicional:	No processo de execução de uma função as plataformas *serverless* adicionam algum *overhead*. Neste teste será utilizada uma função simples (pouca carga computacional), para que o tempo da sua execução não tenha impacto nos resultados e se consiga assim comparar apenas o *overhead* inerente às plataformas. A função será executada de forma sequencial durante um período de tempo, medindo-se a latência de cada invocação em cada plataforma. Como o teste será executado nas diferentes plataformas usando a mesma função, através dos tempos recolhidos será possível comparar os seus *overheads*

Tabela 6.2: Definição do teste de *overhead* das plataformas *serverless*

Nome do teste:	Teste de carga concorrente
ID do teste:	T2
Objetivo:	Medir a latência e o *throughput* das plataformas sujeitas a diferentes níveis de carga concorrente de forma a que se consiga perceber a capacidade de escalabilidade
Comentários/ Informação adicional:	Apesar das plataformas oferecerem, em teoria, escalabilidade "infinita", na prática podem reagir de forma diferente dependendo da carga. Este estudo irá permitir verificar como elas realmente se comportam e reagem com níveis de carga distintos. O teste consistirá na invocação, de forma sequencial, de uma função durante um período de tempo, sendo que a cada período de tempo o número de funções a serem invocadas paralelamente aumentará. Serão registados os tempos de latência e o *throughput*, que são métricas que permitem exprimir bem a capacidade de escalabilidade, que serão utilizadas para comparação entre as diferentes plataformas

Tabela 6.3: Definição do teste de carga concorrente

Nome do teste:	Teste de reutilização de *containers*
ID do teste:	T3
Objetivo:	Medir a latência em cenários de utilização de *warm/hot* e *cold containers*, de forma a estabelecerem-se padrões de reutilização de *containers* por parte das plataformas e verificar o impacto que estes cenários apresentam nos tempos de execução de uma função
Comentários/ Informação adicional:	As plataformas *serverless* possuem mecanismos de otimização que permitem atingir uma melhor performance. Um destes mecanismos é a reutilização de *containers*, isto é, quando uma função é invocada pode ser executada num *warm/hot container* dessa função que já esteja ativo, ou poderá ter de ser alocado um novo *cold container*, sendo que neste caso o tempo de execução sofrerá um *overhead* devido ao processo de criação. Normalmente, após cada execução, as plataformas optam por manter os *containers* ativos durante um período de tempo, para que estes possam ser reutilizados em invocações futuras da mesma função. Serão efetuadas invocações de uma função de forma sequencial, espaçadas por um tempo incremental, registando-se o tempo de latência de cada invocação, o que permitirá o estudo e comparação do impacto da reutilização de *containers* nas diversas plataformas, assim como o *overhead* introduzido pela utilização de *cold containers* e o tempo em que as plataformas mantêm *warm containers* ativos.

Tabela 6.4: Definição do teste de reutilização de *containers*

Nome do teste:	Teste de impacto do tamanho do *payload*
ID do teste:	T4
Objetivo:	Perceber o impacto do tamanho do *payload* (dados que entram e saem da função) na performance das plataformas *serverless*
Comentários/ Informação adicional:	O tamanho do *payload*, isto é, o tamanho dos dados que entram e saem de uma função, afeta a performance das plataformas *serverless*. De forma a verificar este facto, será efetuado um teste de invocação de uma função com diferentes níveis de tamanho de dados de entrada e saída. A mesma função e dados serão testados nas diferentes plataformas e o tempo de *latência* irá permitir validar ou não o agravamento de performance assim como a comparação entre plataformas da mesma

Tabela 6.5: Definição do teste do impacto do tamanho do *payload*

Nome do teste:	Teste de impacto da linguagem de programação na performance
ID do teste:	T5
Objetivo:	Identificar e medir o impacto de performance das diferentes linguagens de programação nas plataformas *serverless*
Comentários/ Informação adicional:	Diferentes linguagens de programação podem apresentar diferentes performances. Por este motivo uma função será escrita nas diferentes linguagens de programação suportadas pelas plataformas e será realizada uma comparação dos tempos de latência entre as linguagens dentro da mesma plataformas e entre possivelmente entre plataformas, de modo a que se verifique em quais cada linguagem é melhor e pior e quais as que permitem obter um melhor desempenho

Tabela 6.6: Definição do teste de impacto da linguagem de programação na performance

Nome do teste:	Teste de performance em função da memória alocada
ID do teste:	T6
Objetivo:	Comparar a performance de execução de uma função com diferentes níveis de memória
Comentários/ Informação adicional:	Algumas das plataformas de computação *serverless* permitem que se configure a memória que irá ser disponibilizada à função para a sua execução. Ao ser definido um determinado valor, este poderá influenciar também a capacidade de CPU e de largura de banda atribuídas à função, o que poderá influenciar bastante o tempo de execução. Será realizado um teste correndo uma função com diferentes níveis de memória e será registada a sua latência para cada invocação. Com os tempos registados poderemos comparar as diferenças de performance dentro da mesma plataforma, e entre plataformas, em função da memória de execução para cada nível

Tabela 6.7: Definição do teste de performance em função da memória alocada

Nome do teste:	Teste de performance para execução de funções computacionalmente pesadas
ID do teste:	T7
Objetivo:	Verificar como se comportam as plataformas de computação *serverless* em termos de performance quando têm de executar funções pesadas computacionalmente
Comentários/ Informação adicional:	Funções computacionalmente pesadas, como por exemplo, uma *Fibonacci* recursiva, podem consumir mais recursos da plataforma e por isso afetar a performance geral. Vai ser utilizada neste teste uma função deste tipo, que vai ser invocada registando-se a métrica de *latência* por invocação para verificação do impacto de performance entre as diversas plataformas

Tabela 6.8: Definição do teste de performance para execução de funções computacionalmente pesadas

6.2.2 Sistema de execução do *Benchmark*

De forma a automatizar o processo de execução do *benchmark*, cujos testes foram apresentados na sub-secção anterior, será desenvolvido um sistema, que será disponibilizado em *open source*, tendo como objetivo permitir a realização dos testes, a recolha e registo de resultados e sua apresentação. Este terá como principais características a modularidade e a extensibilidade, permitindo que se adicionem, modifiquem e adaptem testes com facilidade. Inicialmente o sistema será utilizado sob uma interface de linhas de comandos, podendo posteriormente ser desenvolvida uma interface gráfica ou *web*.

De seguida é apresentada a arquitetura do sistema, seguindo o modelo "C4"[1], sob a forma de três diagramas, começando-se por um mais geral e terminando-se num mais particular, como se fossemos fazendo zoom no sistema.

Na figura 6.1 está representado o diagrama de contexto, que apresenta os utilizadores que irão interagir com o sistema, os sistemas de *software* a implementar e os já existentes que irão ser utilizados. Como se pode verificar, para além da aplicação de testes a desenvolver, que irá ser a unidade central do sistema e que conterá toda a lógica dos testes, serão utilizados dois *softwares* já existentes: A *Serverless Framework*[2] é um conjunto de ferramentas *open source* que permitem o *deploying* e a operação de arquiteturas *serverless* em diferentes provedores, sendo suportadas as plataformas *AWS Lambda*, *Google Cloud Functios*, *Azure funtions* e *Apache OpenWhisk/IBM Cloud Functions* [44]. O *JMeter*[3] é uma ferramenta que permite a realização de testes de carga, que irá ser usado para realizar os pedidos de invocação e para guardar os resultados de latência e *throughput* dessas invocações.

O diagrama de *containers*, representado na figura 6.2, permite ter uma visão mais profunda do sistema de testes a desenvolver, através de um zoom realizado na aplicação de testes do diagrama de contexto. O sistema será composto por dois *containers* principais, que são a interface de linha de comandos, desenvolvida em *Python* e que irá permitir a interação do utilizador com o sistema, e a aplicação de gestão de testes, que será um *backend*, também desenvolvido em *Python*, e que executará toda a lógica da aplicação. Existirão ainda três repositórios de ficheiros, que irão permitir o armazenamento do código das funções utilizadas nos testes, o armazenamento das métricas registadas pelo *JMeter* e dos resultados retirados pela aplicação através dessas métricas e por fim um que irá conter os *templates* dos ficheiros de configuração necessários que serão utilizados pelo *JMeter* na execução dos testes de carga.

Por fim, no diagrama de componentes, apresentado na figura 6.3, é possível ter uma visão mais detalhada dos componentes que fazem parte da aplicação de gestão de testes. Cada um tem as suas responsabilidades e irá ser implementado em *Python*. O controlador de *deploy* irá permitir, através da utilização da *Framework Serverless*, o *deploy* das funções

[1] `https://c4model.com/`
[2] `https://serverless.com/`
[3] `https://jmeter.apache.org/`

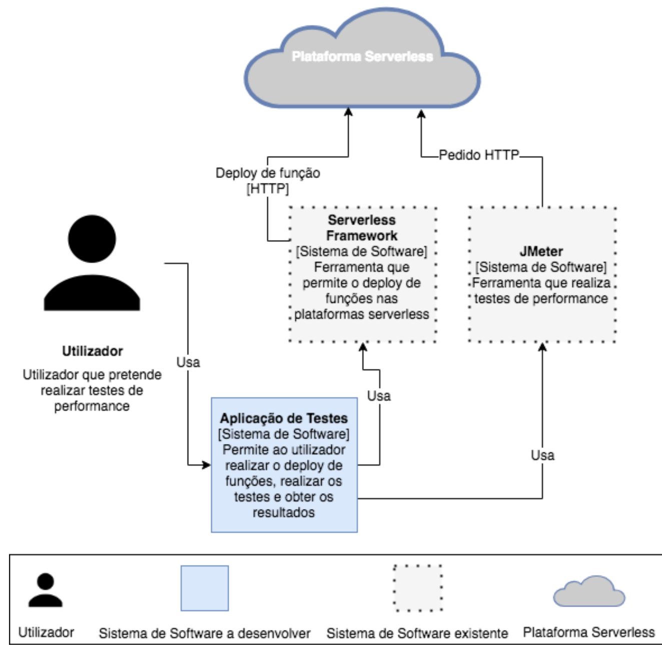


Figura 6.1: Diagrama de contexto do sistema de *Benchmarking* a implementar

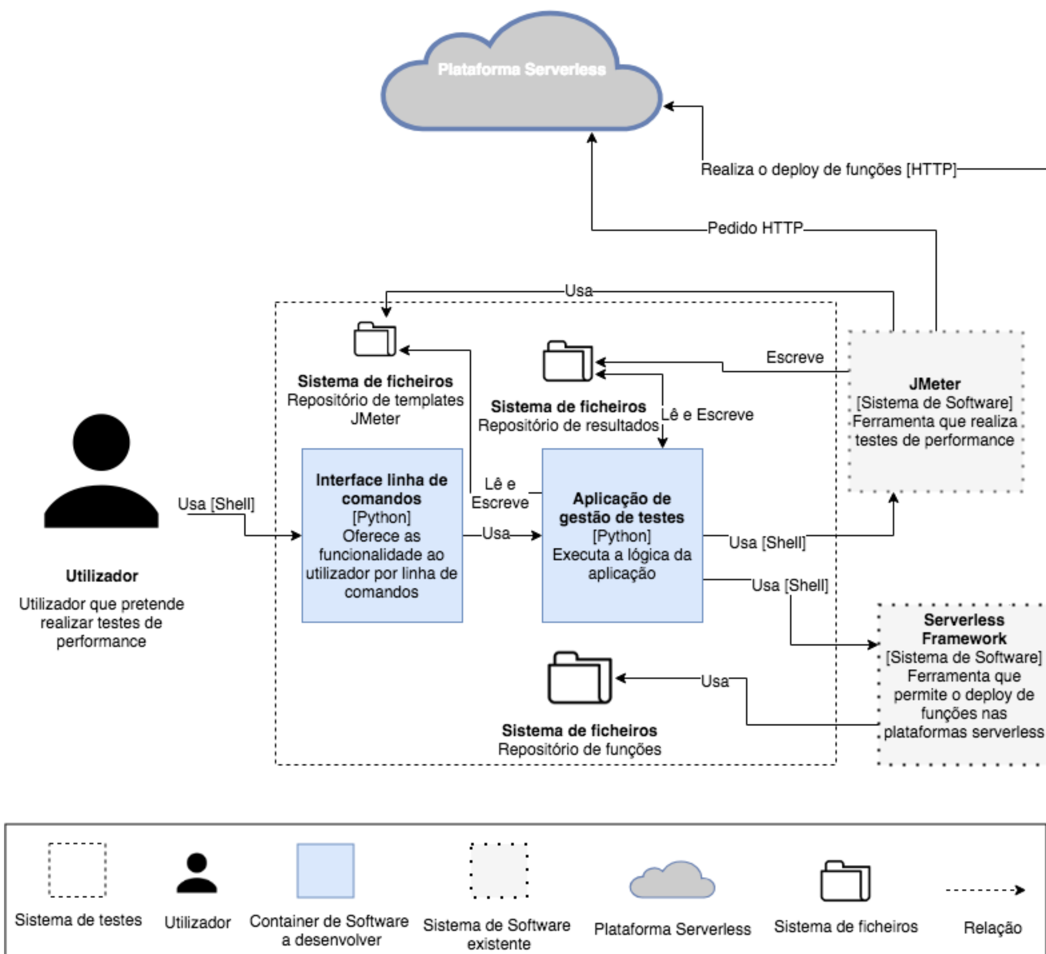


Figura 6.2: Diagrama de *container* do sistema de *Benchmarking* a implementar

nas diversas plataformas onde irão ser executados os testes. O controlador de testes irá permitir a realização dos testes apresentados atualizando os *templates* que irão ser utilizados pelo *JMeter*. O controlador de resultados irá utilizar as métricas recolhidas pelo *JMeter*, de forma a gerar os resultados dos conjuntos de testes, nomeadamente através da criação e armazenamento de gráficos.

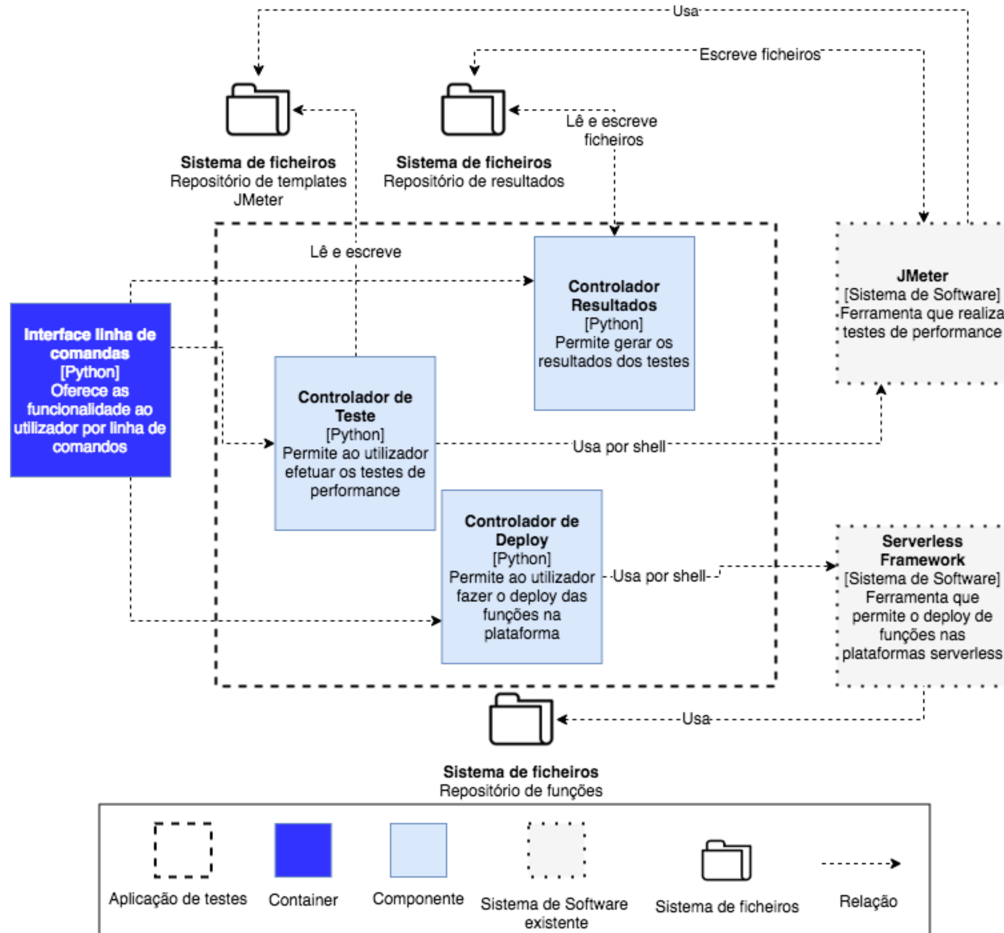


Figura 6.3: Diagrama de componentes do sistema de *Benchmarking* a implementar

A arquitetura proposta serviu de suporte no processo de desenvolvimento do sistema de execução do *benchmark*, que é apresentado no capítulo 7, seguindo estritamente o que aqui foi apresentado.

Esta página foi propositadamente deixada em branco.

Capítulo 7

Implementação e configuração do sistema de execução do *Benchmark*

Partindo da arquitetura apresentada no capítulo anterior, procedeu-se à implementação do sistema que permite automatizar a execução do conjunto de testes de performance, também aí proposto. Neste capítulo são apresentadas as duas ferramentas externas, a *Serverless Framework* e o *JMeter*, utilizadas pelo nosso sistema de execução do *Benchmark*, sendo explicado como estas foram instaladas, configuradas e utilizadas. São também descritas as componentes do sistema desenvolvidas de raiz, explicando-se qual a responsabilidade de cada uma delas no sistema e de que forma é feita a integração com as ferramentas externas.

7.1 Ferramenta auxiliar *Serverless Framework*

A *Serverless Framework*[1] é uma interface de linha de comandos para a construção e *deploy* de aplicações *serverless* em diferentes provedores. Possibilita a criação de funções através da utilização de *templates* existentes para diferentes ambientes e para diferentes linguagens de programação, permite a criação e configuração de eventos de *trigger* que permitem invocar as funções criadas e proporciona ainda o *deploy*, a remoção e a execução das funções.

No nosso sistema, a *Serverless Framework* permite abstrair a complexidade de estarmos a avaliar a performance de plataformas *serverless* de diferentes provedores, com especificidades diferentes, tais como, formas de realizar o *deploy*, formas de criação de *triggers* HTTP, forma como as funções tem de ser declaradas, de como recebem os parâmetros e como retornam os resultados. Concretamente, a *framework* é integrada com a plataforma

[1] `https://serverless.com/`

de execução de *bechmark* proposta através da execução de comandos de terminal que permitem interagir com a ferramenta, e utilizada para a criação, *deploy* e remoção das funções e dos respetivos *triggers* nos diferentes provedores.

Utilizando a *Serverless Framework* apenas é necessário configurar as credenciais dos provedores que se querem utilizar, criar os *templates* para os respetivos provedores e para a linguagem de programação desejada, sendo apenas necessário atualizar o código da função para o que se pretende. Caso se pretenda pode-se ainda definir algumas configurações tais como os *triggers*, a região em que se deseja realizar o *deploy* e a memória que se deseja alocar. A *framework* efetua de forma automática as configurações necessárias nos provedores, baseadas na linguagem de programação utilizada e no provedor em que está a ser feito o *deploy*.

Nas sub-secções seguintes será explicado como é realizada a instalação da *Serverless Framework*, como é efetuada a configuração das credenciais dos provedores utilizados no *benckmark* e como é feita a criação, configuração e *deploy* das funções *serverless*.

7.1.1 Instalação da *Serverless Framework*

Para que seja possível instalar a *Serverless Framework* na máquina em que irá ser executado o sistema de *benchmark* é necessário ter instalado o *Node.Js* e o "npm", que é um gestor de pacotes da linguagem *JavaScript*. Após cumpridos os requisitos anteriores, apenas será necessário executar o comando "npm install -g serverless" no terminal ou na linha de comandos. Este comando, para além de realizar a instalação da *framework*, permite também que esta seja atualizada caso já se encontre instalada. Após o término da execução, a ferramenta estará instalada no sistema e acessível para utilização no terminal, através do comando "serverless".

7.1.2 Configuração da *AWS Lambda*

Caso se pretenda utilizar a plataforma de computação *serverless AWS Lambda* é necessário configurar as respetivas credenciais. Este passo permite que a *framework* possa aceder à conta de utilizador do provedor de *cloud* de forma a criar e gerir os recursos necessários. É necessário que o utilizador que pretende executar o *benchmark* possua uma conta na Amazon Web Services (AWS), com um método de pagamento válido configurado, caso contrario não será possível realizar o *deploy* das funções. A pessoa que cumprir o requisito anterior pode então aceder a sua conta e gerar uma nova chave de acesso à AWS ("AWS Access Keys"), através da criação de um novo utilizador na página de gestão de identidades e acessos ("IAM"). Este deverá ter acesso de programador e possuir permissões de acesso administrador. Finalizada a criação deste utilizador deverá ser guardada, em qualquer local seguro, a chave de acesso ("Acess key ID") e a chave de acesso secreta ("Secret access key").

Com as chave de acesso à AWS, o utilizador deve configurar a *Framework Serverless* para

que as use, através da utilização do comando "serverless config credentials –provider aws –key Acess key ID –secret Secret access key", substituindo a Acess key ID e a Secret access key pelas chaves correspondentes. Após efetuada esta configuração será possível realizar o *deploy* das funções na *AWS Lambda* do utilizador através da *framework*.

7.1.3 Configuração da *Google Cloud Functions*

Para que se possa executar o *benchmark* na plataforma *Google Cloud Functions*, deverá possuir-se uma conta com um método de pagamento válido configurado. O utilizador tem que criar um novo projeto *Google Cloud* seguindos os seguintes passos:

1. Aceder e realizar a autenticação na consola da *Google Cloud*[2]
2. Selecionar "Criar projeto"no menu presente no canto superior esquerdo da página
3. Inserir um nome e criar o projeto

Após realizados os passos anteriores, para que a *framework* possa criar e gerir os recursos necessários para o *deploy* de funções e eventos, deverão ser autorizadas, através da consola de APIs, para o projeto criado, as seguintes APIs: "Google Cloud Funtions"; "Google Cloud Deployment Manager"; "Google Cloud Storage"; "Stackdriver Logging".

Neste momento deverão ser criadas as credencias que irão permitir que a *Framework Serverless* aceda e utilize a conta do utilizador. Para tal, devem ser seguidos os seguintes passos:

1. Aceder e realizar a autenticação na consola da *Google Cloud*[3]
2. Criar uma nova conta de serviço na secção de contas de serviços ("Service accounts") no menu "IAM & admin"
3. Adicionar as seguintes funções: "Deployment Manager Editor; Storage Admin; Logging Admin; Cloud Functions Developer"
4. Criar uma nova chave, selecionando o tipo *JSON*
5. Guardar o ficheiro *JSON* gerado, que contém as chaves, numa localização da máquina que irá operar o sistema de execução do *benchmark*

Para que a *framework* utilize o projeto e as credencias criadas, deverá ser atualizada a secção "provider" do ficheiro de configuração "serverless.yml" da função cujo *deploy* se pretende, com o nome do projeto criado e com o caminho absoluto para o ficheiro *JSON*

[2] `https://console.cloud.google.com/`
[3] `https://console.cloud.google.com/`

que contém as chaves geradas. O processo de criação de uma nova função, através da utilização dos *templates* da *framework serverless*, que permite gerar o ficheiro de configuração referido está descrito na sub-secção 7.1.6.

Para além da configuração das credenciais, é ainda necessário instalar o *plugin* da *Google Cloud Funtions* que permite que a *framework* interaja com a plataforma, através da execução no terminal do comando "npm install –save serverless-google-cloudfunctions".

7.1.4 Configuração da *Azure Functions*

De forma a executar o *benchmark* na plataforma *Azure Functions* deverá possuir-se uma conta da *Azure*. De seguida, deverão ser configuradas as credenciais do provedor. Para tal, devem ser seguidos os seguintes passos:

1. Instalar a interface de linha de comandos da Azure, como indicado na página `https://docs.microsoft.com/en-us/cli/azure/` para o sistema operativo correspondente.
2. Realizar o login na *Azure* através da execução no terminal do comando "az login", que retorna um código que terá de ser introduzido na página *web* que irá abrir automaticamente. Caso tal não aconteça, deverá aceder-se a `aka.ms/devicelogin`.
3. Obter o id da subscrição recorrendo-se ao comando "az account list"
4. Criar um novo serviço utilizando o comando " az ad sp create-for-rbac", guardando os valores do "tenant", do "name" e da "password" retornados.
5. Utilizando os valores obtidos nos passos 3 e 4, atualizar as variáveis de ambiente com os seguintes comandos de terminal:
   (a) export azureSubId="subscriptionId" (valor obtido do passo 3)
   (b) export azureServicePrincipalTenantId="tenant"
   (c) export azureServicePrincipalClientId="name"
   (d) export azureServicePrincipalPassword="password"

Para além da configuração das credenciais, é ainda necessário instalar o *plugin* da *Azure Funtions* que permite que a *framework* trabalhe com a plataforma, através da execução no terminal do comando "npm install -g serverless-azure-functions".

7.1.5 Configuração da *OpenWhisk/IBM Cloud Functions*

Uma vez que a *OpenWhisk* é uma plataforma *open source*, é possível utilizar a plataforma hospedada na *cloud IBM Bluemix* ou então hospedada pelo utilizador ou por terceiros numa infraestrutura própria.

Caso se pretenda utilizar a *OpenWhisk* da *IBM*, denominada de *IBM Cloud Functions*, o utilizador terá de possuir uma conta válida da *IBM Bluemix*. Deve então seguir os passos contidos na página `https://console.bluemix.net/docs/cli/index.html` para que possa instalar a interface de linha de comandos da *IBM Cloud* no seu sistema operativo. Após concluído o passo anterior, deverá ser instalado o *plugin* que permite que a interface possa operar sobre a plataforma, através do comando "ibmcloud plugin install cloud-functions". Neste momento é possível proceder à autenticação do utilizador, recorrendo-se ao comando "ibmcloud login -a region_api -o email", em que a "region_api" deve ser substituída pela url da região da *cloud* que se pretende utilizar e o "email" substituído pelo email associado a conta do utilizador. As regiões em que a plataforma *serverless* está disponível são as seguintes: US-South (api.ng.bluemix.net); US East (api.us-east.bluemix.net); London (api.eu-gb.bluemix.net); Frankfurt (api.eu-de.bluemix.net). Para que a *Framework Serverless* consiga aceder e utilizar as credencias é ainda necessário executar o comando "ibmcloud wsk property get –auth" que permite que o ficheiro "/.wskprops", criado na raiz da máquina, seja preenchido com as credencias necessárias, nomeadamente o *token* de acesso à *API Gateway*, o *host* e a chave de *auth*. Desta forma a *framework* já conseguirá utilizar a conta para *deploy* das funções.

Caso se utilize uma instalação local da *OpenWhisk*, as credenciais poderão ser configuradas através da sua interface de linha de comandos, obtida da pasta que foi clonada do *git* para a instalação da plataforma *serverless*, recorrendo-se ao comando "wsk property set –apihost PLATFORM_API_HOST –auth USER_AUTH_KEY".

Para além da configuração das credenciais, é ainda necessário instalar o *plugin* da *OpenWhisk* que permite que a *framework* interaja com a plataforma, através da execução no terminal do comando 'npm install –global serverless serverless-openwhisk".

7.1.6 Criação, configuração e *deploy* de funções

Quando se pretende compor uma função para *deploy* num dos provedores através *Framework Serverless*, é necessário proceder-se a criação de um "serviço", que pode ser entendido como um projeto onde se podem definir as funções, os seus eventos de *trigger* e todos os recursos requeridos num ficheiro chamado de "serverless.yml".

A criação de um serviço é realizada através de utilização de *templates* que dependem do provedor em que se deseja realizar o *deploy* e da linguagem de programação que se pretende utilizar. Por exemplo, o comando que permite a criação de um serviço, ao qual se deu o nome de "awsBenchmarkT1", para uma função escrita em *Node.js*, para *deploy* na *AWS Lambda* , é "serverless create –template aws-nodejs –path awsBenchmarkT1". Para os restantes provedores e para as restantes linguagens de programação o comando é o mesmo, alterando-se apenas o *template* a utilizar e o nome que se deseja atribuir ao serviço.

Após a criação do serviço é gerada uma nova pasta que contém um ficheiro onde se pode es-

crever o código pretendido para a função e um outro ficheiro, chamado de "serverless.yml", onde se realiza a declaração do serviço, se define o provedor e algumas configurações, e onde se definem as funções que fazem parte do serviço e os eventos que permitem ativar as suas invocações. Algumas das configurações que podem estar neste ficheiro são a região em que se pretende realizar o deploy e a memória que se pretende alocar à função.

Na figura 7.1 é possível ver o código da função escrito para o serviço criado, denominada "awsFunctionT1". Na figura 7.2 encontra-se o ficheiro de definição do serviço criado, onde se encontra indicado o provedor, a região em que se deseja realizar o *deploy* da função, o ambiente de execução e a memória alocada. Como se pode observar pela figura 7.2 a função foi associada a este serviço, sendo que *handler* é o nome do ficheiro onde se encontra a função. Foi ainda associado um evento *HTTP* do tipo *get*, que irá permitir invocar a função através de um *trigger* HTTP.

```
module.exports.awsFunctionT1 = async(event, context) => {
  return {
    statusCode: 200,
    body: JSON.stringify({
      message: 'Hello, World!',
    }),
  };
};
```

Figura 7.1: Exemplo de função escrita em *Node.js* para *deploy* na *AWS Lambda*

```
service: awsBenchmarkT1

provider:
  name: aws
  runtime: nodejs8.10
  region: eu-west-2
  memorySize: 256

functions:
  awsFunctionT1:
    handler: handler.awsFunctionT1
    events:
      - http:
          path: /awsFunctionT1
          method: get
```

Figura 7.2: Exemplo de ficheiro "serverless.yml", utilizado para definição de um serviço

Após a criação do serviço estar completa é possível realizar o *deploy* da função e dos *triggers* definidos no respetivo provedor de computação *serverless*. Para isso, será necessário executar o comando "serverless deploy" dentro da diretoria do serviço, isto é, no local onde se encontra o ficheiro com a definição do serviço. Uma vez executado o comando, a *Framework Serverless* encarrega-se de realizar o *deploy* de uma vez só de tudo o que foi definido no ficheiro. Caso se pretenda proceder a eliminação da função e dos respetivos *triggers*, apenas se terá de correr o comando "serverless remove" na mesma diretoria.

7.2 *JMeter*

O *JMeter*[4] é um *software* que permite a realização de testes de carga a sistemas computacionais. No nosso sistema de *benchmark* o *JMeter* é usado como ferramenta que realiza as invocações das funções, controla os níveis de concorrência, controla o tempo durante o qual serão feitas invocações sequenciais, e regista as métricas necessárias para a geração dos resultados para cada teste. Os pedidos de invocação são realizados através de pedidos HTTP às funções, sendo para isso necessário apenas o *URL* das mesmas. O controlo de concorrência e do tempo de execução do teste são efetuados através da utilização de um grupo de *threads* do *JMeter*, onde é possível especificar o tempo em que o teste fica ativo, ou seja, durante quanto tempo são efetuados pedidos de invocação, e qual o número de *threads* que executa o teste, ou seja, o número de pedidos concorrentes ativo.

Nas sub-secções seguintes será explicado como é feita a instalação do *JMeter* na máquina que irá correr o sistema de *Benchmark*, como é criado o *template* do ficheiro de configurações (também chamado de plano de teste) necessário para que a ferramenta execute os testes e quais são as métricas e em que formato são armazenadas após cada teste efetuado.

7.2.1 Instalação e utilização do *JMeter*

Para que seja possível instalar o *JMeter* na máquina em que irá ser executado o sistema de *bechmark* é necessário ter instalado o *Java* 8 ou 9. Após cumprido este requisito, apenas será necessário fazer o *download* do *software* diretamente da página do *Apache JMeter*[5] e armazena-lo num diretório da máquina. O caminho para o local onde foi guardado será necessário para que o sistema a possa usar.

Na pasta “bin” encontra-se o executável, “ApacheJMeter.jar”, que permite iniciar o *JMeter* recorrendo à sua interface gráfica, utilizada para criação de planos de teste e para a execução dos mesmos, sendo possível visualizar também os resultados. Para além do modo gráfico, é possível correr o *JMeter* através de linha de comandos ou terminal, sendo necessário apenas existir um plano de testes, que pode ter sido criado previamente. Para correr um teste de carga neste modo apenas é necessário executar, dentro da pasta “bin”, o comando “sh jmeter -n -t plano_de_teste -l resultados”. O “-n” indica que não vai ser utilizada interface gráfica; o “-t” indica que de seguida será passado como parâmetro o caminho e nome do plano de teste que irá ser executado, sendo o “plano_de_teste” substituído por esta informação. Por fim, o “-l” permite que os resultados (métricas obtidas pelo JMeter) sejam guardados num local e com o nome indicado no lugar de “resultados”.

No sistema de execução de *benchmark* a interface gráfica é utilizada para a criação de planos de teste padrão, seguindo o processo explicado na sub-secção seguinte. Estes serão utilizados e atualizados pela nossa aplicação, para que com a utilização do *JMeter*, no

[4] `https://jmeter.apache.org/`

[5] `http://jmeter.apache.org/download_jmeter.cgi`

modo de linha de comandos, sejam efetuadas as invocações necessárias para a realização dos testes de performance.

7.2.2 Criação do plano de teste padrão para execução de testes no *JMeter*

Como referido sub-secção anterior, para a execução de testes de carga no *JMeter* é necessário a utilização de um plano de teste, que permite ao *JMeter* saber que componentes deve utilizar e que ações deve executar. O plano é um ficheiro, em formato"xml", que contém toda a informação e todas as configurações necessárias para que a ferramenta possa executar um teste. Os testes que realizamos não são estáticos, pois o tempo durante o qual são executados, o *url* das funções a serem invocadas e o nível de concorrência são alterados em função de qual é executado e do que o utilizador que o realiza pretende. Neste sentido, foi criado um plano de testes padrão que que contem as componentes necessárias para a execução das invocações no *JMeter* e cujo os valores para as suas configurações são atualizados pelo sistema desenvolvido mediante o teste a ser executado e as preferências que o utilizador tem para o mesmo.

O plano de teste padrão é constituído por um grupo de *threads*, que permite controlar a concorrência de pedidos de invocação e o tempo durante o qual estas serão feitas. A criação desta componente pode ser visualizada na figura 7.3, sendo que os valores para o número de *threads* e para a duração dos pedidos de invocação foram preenchidos com valores padrão, que serão atualizados pela plataforma mediante o teste e o pretendido pelo utilizador. Uma vez que as invocações das funções são realizadas através de pedidos HTTP utilizando o método "*GET*" , dentro do grupo de *threads* foi adicionada uma componente que permite efetuar este tipo de pedidos, como se pode ver na figura 7.4, através da definição do respetivo método e do *URL* da função a invocar, que será atualizado pelo sistema de *bechmark* mediante o teste que vai ser realizado e a plataforma *serverless* a testar.

O plano de teste efetuado é guardado num ficheiro, num diretório acessível pelo sistema de execução, para que este o possa atualizar mediante o necessário e o possa utilizar aquando da integração com o *JMeter*, para a realização das invocações e registo de métricas dos testes.

7.2.3 Métricas obtidas da execução de testes no *JMeter*

Ao executar um dos testes no sistema de *benchmark*, quando o *JMeter* realiza pedidos de invocação, após estes terminarem é gerado e guardado um ficheiro no formato ".jtl", que não é mais que um csv, em que, para cada pedido de invocação, são apresentadas as seguintes métricas e informações:

- timeStamp: Identificador da operação que é o tempo no momento em que é realizado

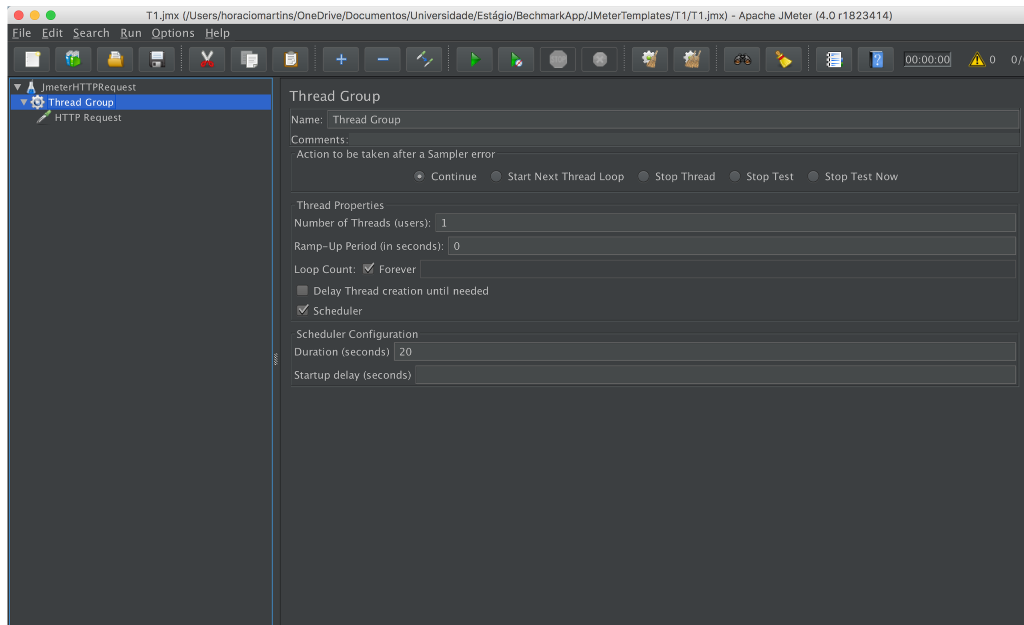


Figura 7.3: Criação do grupo de *threads* do plano de teste padrão do *JMeter*

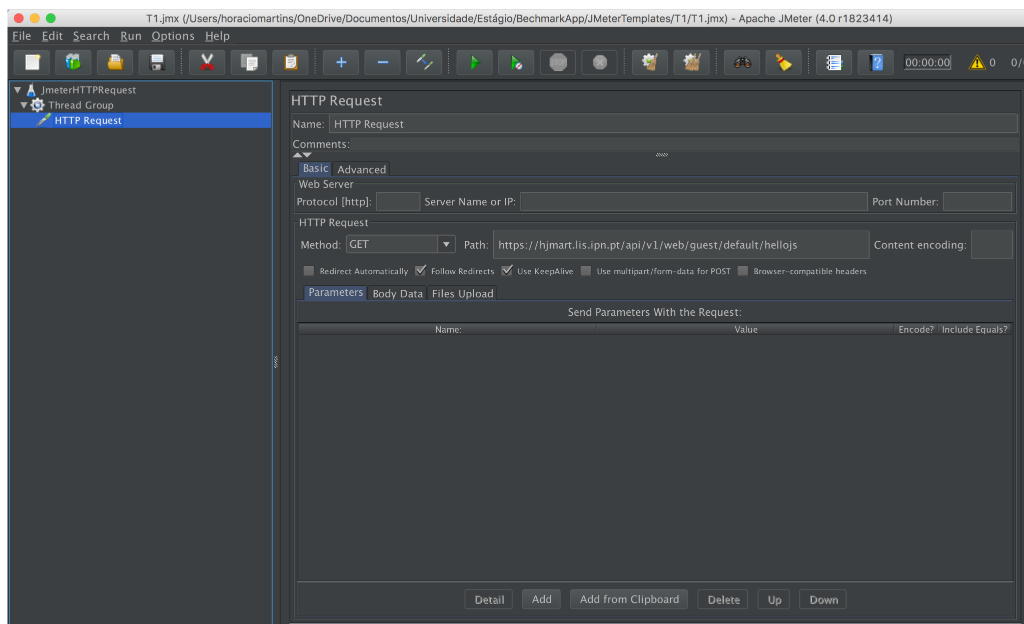


Figura 7.4: Criação do pedido HTTP do plano de teste padrão do *JMeter*

o pedido de invocação.

- elapsed: Tempo em ms ocorrido desde que é mandado o pedido até que seja recebido o último byte da resposta.
- label: Label que identifica o pedido, como por exemplo HTTP.
- responseCode: Código da resposta, normalmente 200 em caso de sucesso e 400 em caso de falha.
- responseMessage: Mensagem contida na resposta ao pedido.

- threadName: Identifica a *thread* que realizou o pedido.
- dataType: O tipo de dados da mensagem de resposta.
- success: Um booleano que indica se o pedido de invocação foi realizado com sucesso ou não.
- failureMessage: Mensagem contida na resposta ao pedido em caso de falha.
- bytes: Tamanho da resposta ao pedido em bytes.
- sentBytes: Tamanho do pedido de invocação.
- grpThreads: Grupo ao qual pertence a *thread* que efetuou o pedido de invocação.
- allThreads: Número de *threads* mantidas pelo *JMeter*.
- Latency: Tempo em ms ocorrido desde que é enviado o pedido até que se comece a receber a resposta.
- IdleTime: Tempo em ms em que o pedido esteve em pausa, normalmente 0.
- Connect: Tempo em ms levado para estabelecer a ligação TCP entre o cliente, neste caso o *JMeter*, e o servidor usando o *handshake*.

Por vezes, o *JMeter* perde a ligação TCP, tendo que estabelecer uma nova em alguns dos pedidos de invocação de função. Isto pode levar a que, em alguns casos, seja levado em consideração nos tempos de latência o tempo de ligação (*connect*), e noutras não, visto que esse tempo está incluído na latência. Para uniformização nos resultados calculados foi definido um novo tempo de latência em que se retira a esse tempo o tempo de ligação, permitindo obter assim um valor mais uniforme dos vários pedidos de invocação.

O ficheiro gerado será usado para o controlador de resultados do sistema gerar os resultados e os gráficos dos respetivos testes, permitindo obter as latências dos pedidos de invocação, o número de invocações realizadas, e se foram executadas com sucesso ou não. O *throughput*, utilizado como resultado em alguns dos testes, é calculado internamente pelo *JMeter* e retornada pelo mesmo a quando da finalização da execução de um teste.

7.3 Interface de linha de comandos e aplicação de gestão de testes

No desenvolvimento do sistema de execução do *benchmark*, para além da utilização e integração da *Serverless Framework* e do *JMeter*, foi desenvolvida de raiz uma parte da plataforma. Esta seguiu a arquitetura descrita na sub-secção 6.2.2 e foi implementada em *Python*. A interface de linha de comandos funciona como entrada do sistema e permite a interação com o utilizador. A aplicação de gestão de testes divide-se em três

controladores: O controlador de *deploy*, que é responsável, através da integração com a *Framework Serverless*, pelo *deploy*/remoção das funções utilizadas no *benchmarking* nos respetivos provedores. O controlador de testes, que efetua toda a gestão e orquestração de todo processo de execução dos testes nos vários provedores. O controlador de resultados, que se encarrega da manipulação dos dados obtidos a quando da execução dos testes, para geração dos resultados do *benchmarking.* O sistema conta ainda com um ficheiro de configuração que permite que um utilizador do sistema o possa configurar. Este ficheiro permite ainda que sejam modificados testes facilmente, por exemplo, através da alteração das funções a utilizar editando-se o parâmetro correspondente. Nas sub-secções seguintes serão abordadas as componentes aqui referidas.

7.3.1 Ficheiro de configuração da plataforma

Para que se possam editar e armazenar algumas configurações necessárias para a plataforma de execução do *benchmark*, de forma a que esta possa funcionar e executar os testes, foi utilizado um ficheiro em formato JSON denominado de "conf.json". Optou-se por este formato por ser de fácil leitura e escrita para humanos. Um utilizador do sistema pode então definir as configurações de acordo com o que pretende, podendo alterar os testes implementados através da edição de alguns dos parâmetros presentes no ficheiro. Por exemplo, o *url* de umas das funções, utilizado num determinado teste, pode ser alterado pelo utilizador para o de uma função que este já possua num dos provedores e que deseje utilizar no testes, através da edição do parâmetro correspondente.

Os parâmetros que fazem parte do ficheiro de configuração são os seguintes:

- "jMeterPath": Identifica o caminho para a localização da pasta "bin" do *JMeter*, utilizado para que o sistema possa utilizar esta ferramenta.
- "jMeterTemplates": Para cada teste é definido o caminho para o *template* do plano de testes do *JMeter* correspondente. Desta forma, a aplicação sabe qual dos *templates* deve utilizar a quando da execução de um teste de *bechmark.*
- "jMeterResultsPath": Permite que se defina, para cada teste, o caminho para a diretoria onde se pretende que sejam guardadas as métricas provenientes da execução dos pedidos de invocação no *JMeter.*
- "bechmarkResultsPath": Permite que se defina, para cada teste, o caminho para a diretoria onde se pretende que sejam guardados os gráficos, em formato "png", gerados pelo sistema de execução do *bechmark*, a quando da realização de um teste.
- "providerFunctions": Para cada um dos provedores é declarado, para cada teste. o nome do(s) serviço(s) criado(s) através da "Serverless Framework" assim como o nome das respetivas funções.
- "functionsURL": Para cada um dos provedores é armazenado, para cada teste, o *url* das funções que necessitam de ser invocadas na realização do teste.

- "functionPath": Para cada um dos provedores é mantido o caminho para cada um dos serviços criado através da "Serverless Framework" para o *deploy* de funções que são necessárias para os testes.

- "providers": Identifica, para cada teste, em que provedores este será executado.

- "T4PayloadSize": Mantém os tamanhos de *payload* que são usados no teste T4, de impacto de tamanho do *payload*.

- "T7Weights": Mantém os valores de "n" que são usados na função de calculo da sequencia de Fibonacci do teste T7, teste de impacto de funções computacionalmente pesadas.

7.3.2 Interface de linha de comandos

A interface de linha de comandos é a componente da plataforma de *benchmark* que permite a interação com o utilizador. Foi desenvolvida através da utilização do módulo "argparse"[6], que torna a tarefa de implementar uma interface deste tipo mais fácil. Nesta foram definidos todos os comandos aceites pela plataforma, assim como os respetivos parâmetros. Quando é executado um novo comando, o módulo para além de o identificar, verifica se o número de argumentos está correto, assim como os seus tipos. Caso o comando não esteja definido ou haja um erro nos argumentos passados é apresentado o erro correspondente. É também possível visualizar mensagens de ajuda e de utilização, como se pode ver na figura 7.5, através da utilização do comando "python3 ServerlessBenchmarkAppInterface.py -h". Esta mensagem inclui todos os comandos existentes e os respetivos parâmetros, assim como uma pequena descrição do que permitem efetuar.

Ao introduzir e executar um comando para o *deploy* ou para a remoção de uma dada função é invocada uma função do controlador de *deploy*, e a partir deste momento é esta componente responsável pela operação pretendida. Caso o comando seja de execução de um determinado teste, é chamada uma função do controlador de testes, que se encarrega da operação que foi pedida pelo utilizador.

Os comandos, aceites pela interface de linha de comandos, que permitem interagir com a plataforma de execução de *benchmark* são apresentados de seguida, assim como os parâmetros requeridos por cada um.

- "python3 ServerlessBenchmarkAppInterface.py -h": Apresenta a mensagem de ajuda e de utilização.

- "python3 ServerlessBenchmarkAppInterface.py -d serverless_provider test_number": Permite o deploy, no provedor de computação *serverless*, da função, ou das funções, necessárias para um determinado teste. Os parâmetros requeridos são os seguintes:

[6]`https://docs.python.org/3/library/argparse.html`

```
MacBook-Pro-de-Horacio:BechmarkApp horaciomartins$ Python3 ServerlessBenchmarkAppInterface.py -h
usage: ServerlessBenchmarkAppInterface.py [-h]
                                          [-d serverless_provider test_number]
                                          [-o serverless_provider test_number execution_time]
                                          [-c serverless_provider test_number min_concurrency max_concurrency concurrency_step level_concurrency_execution_time]
                                          [-b serverless_provider test_number min_wait_time max_wait_time time_step pre_exec_time]
                                          [-p serverless_provider test_number execution_time]
                                          [-l serverless_provider test_number execution_time]
                                          [-m serverless_provider test_number execution_time]
                                          [-w serverless_provider test_number execution_time]
                                          [-r serverless_provider test_number]

Serverless Benchmark Interface!

optional arguments:
  -h, --help            show this help message and exit
  -d serverless_provider test_number, --deploy serverless_provider test_number
                        Deploy in the provider all the functions that are
                        needed for a specified test
  -o serverless_provider test_number execution_time, --overhead serverless_provider test_number execution_time
                        Run in the provider all the functions that are needed
                        for a overhead(latency) specified test
  -c serverless_provider test_number min_concurrency max_concurrency concurrency_step level_concurrency_execution_time, --concurrency serverless_provider test_number min_concurrency max_concurrency concurrency_s
tep level_concurrency_execution_time
                        Run in the provider all the functions that are needed
                        for a concurrency specified test
  -b serverless_provider test_number min_wait_time max_wait_time time_step pre_exec_time, --backoff serverless_provider test_number min_wait_time max_wait_time time_step pre_exec_time
                        Run in the provider all the functions that are needed
                        for a container reuse specified test
  -p serverless_provider test_number execution_time, --payload serverless_provider test_number execution_time
                        Run in the provider all the functions that are needed
                        for payload size test
  -l serverless_provider test_number execution_time, --language serverless_provider test_number execution_time
                        Run in the provider all the functions that are needed
                        for test with different programming languages
  -m serverless_provider test_number execution_time, --memory serverless_provider test_number execution_time
                        Run in the provider all the functions that are needed
                        for test with different memory levels
  -w serverless_provider test_number execution_time, --weightcomputacional serverless_provider test_number execution_time
                        Run in the provider all the functions that are needed
                        for test with different computational weight levels
  -r serverless_provider test_number, --remove serverless_provider test_number
                        Remove from the provider all the functions that are
                        needed for a specified test
MacBook-Pro-de-Horacio:BechmarkApp horaciomartins$
```

Figura 7.5: Resultado da execução do comando de ajuda no sistema de execução de *benchmark*

  - serverless_provider: identifica o provedor de computação *serverless* no qual se deseja realizar o *deploy*, podendo tomar os seguintes valores: aws; azure; google; ow.
  - test_number: identifica o número do teste, de acordo com os testes propostos na secção 6.2.1, do qual se pretende realizar *deploy* das funções de que necessita.

- "python3 ServerlessBenchmarkAppInterface.py -o serverless_provider test_number execution_time": Executa o teste de *overhead* das plataformas *serverless* (T1). Para tal, são efetuados pedidos de invocação sequenciais (é efetuado um pedido de invocação imediatamente após ser recebida a resposta do anterior) durante um período de tempo. Os parâmetros requeridos são os seguintes:
  - serverless_provider: identifica o provedor de computação *serverless* no qual se deseja realizar o teste, podendo tomar os seguintes valores: aws; azure; google; ow; all (o teste é executado em todas as plataformas).
  - test_number: identifica o número do teste, de acordo com os testes propostos na secção 6.2.1, que se pretende executar. Neste caso é o 1.
  - execution_time: Tempo durante o qual são efetuados pedidos de invocação sequenciais à função do teste.

- "python3 ServerlessBenchmarkAppInterface.py -c serverless_provider test_number min_concurrency max_concurrency concurrency_step level_concurrency_execution_time": Executa o teste de carga concorrente das plataformas *serverless* (T2). O teste realiza iterações de pedidos de invocação sequenciais, sendo que em cada uma das iterações

o número de invocações concorrentes é incrementado. Os parâmetros requeridos são os seguintes:

- serverless_provider: identifica o provedor de computação *serverless* no qual se deseja realizar o teste, podendo tomar os seguintes valores: aws; azure; google; ow; all (o teste é executado em todas as plataformas).
- test_number: identifica o teste 2 como sendo o que se pretende executar.
- min_concurrency: indica o valor mínimo de pedidos de invocação concorrentes.
- max_concurrency: indica o valor máximo de pedidos de invocação concorrentes.
- concurrency_step: indica o valor de pedidos de invocação concorrentes que serão incrementados a cada iteração do teste.
- level_concurrency_execution_time: tempo durante o qual são efetuados pedidos de invocação sequenciais à função do teste para cada nível de concorrência.

- "python3 ServerlessBenchmarkAppInterface.py -b serverless_provider test_number min_wait_time max_wait_time time_step pre_exec_time": Executa o teste de reutilização de *containers* das plataformas *serverless* (T3). O teste realiza uma pré-execução de pedidos de invocação sequenciais, de forma a garantir que existam *warm containers* da função na plataforma de computação *serverless* do provedor, passando depois a realizar pedidos de invocação espaçados por um tempo incremental. Os parâmetros requeridos são os seguintes:

  - serverless_provider: identifica o provedor de computação *serverless* no qual se deseja realizar o teste, podendo tomar os seguintes valores: aws; azure; google; ow; all (o teste é executado em todas as plataformas).
  - test_number: identifica o teste 3 como sendo o que se pretende executar.
  - min_wait_time: tempo mínimo de espera antes da realização do pedido de invocação.
  - max_wait_time: tempo máximo de espera antes da realização do pedido de invocação.
  - time_step: valor incrementado ao tempo de espera entre pedidos de invocação.
  - pre_exec_time: tempo durante o qual na pré-execução são efetuados pedidos de invocação sequenciais à função do teste.

- "python3 ServerlessBenchmarkAppInterface.py -p serverless_provider test_number execution_time": Executa o teste de impacto do tamanho do *payload* das plataformas *serverless* (T4). Para tal, para cada valor de tamanho de *payload* distinto, declarado no ficheiro de configuração do sistema de execução de *bechmark*, são efetuados pedidos de invocação sequenciais durante um período de tempo a uma função que retorna uma mensagem com o tamanho respetivo. Os parâmetros requeridos são os seguintes:

  - serverless_provider: identifica o provedor de computação *serverless* no qual se deseja realizar o teste, podendo tomar os seguintes valores: aws; azure; google; ow; all (o teste é executado em todas as plataformas).
  - test_number: identifica o teste 4 como sendo o que se pretende executar.
  - execution_time: Tempo durante o qual são efetuados pedidos de invocação sequenciais à função do teste para cada valor distinto de tamanho de *payload*.

- "python3 ServerlessBenchmarkAppInterface.py -l serverless_provider test_number execution_time": Executa o teste de impacto da linguagem de programação na performances das plataformas *serverless* (T5). Para tal, para cada versão de uma mesma função, escrita em diferentes linguagens de programação são efetuados pedidos de invocação sequenciais durante um período de tempo. Os parâmetros requeridos são os seguintes:
  - serverless_provider: identifica o provedor de computação *serverless* no qual se deseja realizar o teste, podendo tomar os seguintes valores: aws; ow.
  - test_number: identifica o teste 5 como sendo o que se pretende executar.
  - execution_time: Tempo durante o qual são efetuados pedidos de invocação sequenciais às funções do teste.

- "python3 ServerlessBenchmarkAppInterface.py -m serverless_provider test_number execution_time": Executa o teste de performance em função da memória alocada à função (T6). Para tal, para uma cada versão de uma mesma função, para a qual se alocam diferentes níveis de memória, são efetuados pedidos de invocação sequenciais durante um período de tempo. Os parâmetros requeridos são os seguintes:
  - serverless_provider: identifica o provedor de computação *serverless* no qual se deseja realizar o teste, podendo tomar os seguintes valores: aws; google; ow; all (o teste é executado em todas as plataformas).
  - test_number: identifica o teste 6 como sendo o que se pretende executar.
  - execution_time: Tempo durante o qual são efetuados pedidos de invocação sequenciais à função do teste com diferentes níveis de memória alocada.

- "python3 ServerlessBenchmarkAppInterface.py -w serverless_provider test_number execution_time": Realiza o teste de performance para execução de funções computacionalmente pesadas (T7). Para tal são efetuados pedidos de invocação sequenciais durante um período de tempo a uma função deste tipo. Os parâmetros requeridos são os seguintes:
  - serverless_provider: identifica o provedor de computação *serverless* no qual se deseja realizar o teste, podendo tomar os seguintes valores: aws; azure; google; ow; all (o teste é executado em todas as plataformas).
  - test_number: identifica o teste 7 como sendo o que se pretende executar.

  - execution_time: Tempo durante o qual são efetuados pedidos de invocação sequenciais à função do teste.

- "python3 ServerlessBenchmarkAppInterface.py -r serverless_provider test_number": Permite remover, do provedor de computação *serverless*, a função, ou as funções, que foram *deployed* para um determinado teste. Os parâmetros requeridos são os seguintes:
  - serverless_provider: identifica o provedor de computação *serverless* do qual se deseja realizar a remoção, podendo tomar os seguintes valores: aws; azure; google; ow.
  - test_number: identifica o número do teste, de acordo com os testes propostos na secção 6.2.1, do qual se pretende realizar a remoção das funções correspondentes.

7.3.3 Controlador de *deploy*

O controlador de *deploy* é o componente que permite o *deploy* e a remoção de funções e dos respetivos eventos de *triggers* das plataformas de computação *serverless*. Para tal, são utilizados serviços criados através da *Framework Serverless* e é utilizada a própria *framework*.

Quando é requerido o *deploy* das funções de um determinado teste num dos provedores, o controlador obtém, do ficheiro de configuração "conf.json", os caminhos para a diretoria dos serviços das funções correspondentes ao provedor e ao teste. Após obtê-los, é utilizado o módulo de *python* "subprocess"[7], que permite a geração de processos para a execução de comandos de terminal, para a invocação nas diretorias obtidas do comando da *Serverless Framework* que permite o *deploy* de funções: "serverless deploy". Findo o passo anterior, são registados no ficheiro de configurações, no parâmetro referente ao provedor e ao teste, os *urls* das funções que foram *deployed*.

Caso seja pretendida a remoção, de um dos provedores, das funções referentes a um determinado teste, o processo é idêntico ao realizado para o *deploy*, à exceção do comando da *Serverless Framework* a ser executado, que neste caso é o que permita a remoção de funções: "serverless remove". Ao ser realizada uma remoção, no ficheiro de configurações são eliminados os *urls* correspondentes as funções que foram removidas.

7.3.4 Controlador de testes

O controlador de testes é a componente do sistema de execução de *benchmark* que gere e executa os testes de performance nas várias plataformas de computação *serverless*. Para tal, através do *JMeter* são realizadas invocações a funções que foram *deployed* nessas plataformas e são registadas as métricas abordadas na sub-secção 7.2.3.

[7] `https://docs.python.org/2/library/subprocess.html`

Ao ser requerida a realização de um determinado testes num dos provedores, o controlador obtém do ficheiro de configuração o *template* do plano de teste correspondente, assim como os *urls* das funções necessárias na execução do teste. Depois é atualizado o *template* de acordo com o teste pretendido, modificando-se o tempo durante o qual serão realizados pedidos de invocação e o *url* da função a invocar. Em alguns dos testes é alterado também o número de *threads*, o que permite definir o número de pedidos concorrentes. Após terminada a atualização do plano de teste, este é executado no *JMeter*, recorrendo-se ao módulo "subprocess" para a execução do comando apresentado na sub-secção 7.2.1. As métricas obtidas pelo *JMeter* são guardadas no local definido no ficheiro de configuração para o respetivo teste.

Este controlador é então responsável por controlar toda a lógica inerente à realização de cada um dos testes, tal como, controlo das funções a serem invocadas, controlo da duração dos pedidos de invocação, controlo do nível de concorrência e controlo do tempo de espera entre pedidos de invocação subsequentes.

Após finalizadas as invocações às funções de um teste de performance, ou de um conjunto de testes, o controlador passa a informação dos que foram efetuados e dos ficheiros de métricas, gerados pela realização das invocações de funções no *JMeter*, ao controlador de resultados, para que este possa gerar os resultados finais da execução do *benchmark*.

7.3.5 Controlador de resultados

O controlador de resultados é o componente do sistema de execução de *benchmark* que gera os dados finais e os gráficos relativos aos testes efetuados. Para tal, através das métricas recolhidas pelo *JMeter* aquando da realização das invocações às funções, e com a utilização do módulo de *python* "Pandas"[8] (que é uma biblioteca para análise e tratamento de dados) são calculados e apresentados em formato de texto os seguintes dados: Latência média; Latência máxima; Latência mínima; Desvio padrão das latências; Percentagem de invocações que foram concluídas com sucesso; Percentagem de invocações que foram concluídas com falha; Número de invocações realizadas durante a execução do teste; E, em alguns dos testes, o *throughput* por segundo.

Para além dos dados apresentados são também gerados, utilizando-se o *matplotlib*[9], gráficos que permitem representar o resultados de performance obtidos para cada um dos testes. Estes gráficos são armazenados em formato "png" na diretoria definida no ficheiro de configurações para cada um dos testes.

[8] `https://pandas.pydata.org/`
[9] `https://matplotlib.org/`

7.4 Resumo do Capítulo

Com a implementação e a integração de cada uma das componentes apresentadas neste capítulo, e configurando-se de acordo com o também referido, foi possível implementar o sistema de execução do *bechmark*, seguindo-se a arquitetura proposta no capítulo anterior. Este sistema permite uma total automatização do processo de *benchmarking* de plataformas *serverless*, desde o *deploy* das funções nas respetivos provedores, à realização das invocações dos testes implementados, geração de resultados e remoção das funções. O sistema após estar completamente terminado e operacional foi utilizado na execução do *bechmark*, descrito no capítulo seguinte.

Capítulo 8

Execução do *Benchmark* e Resultados

Com a utilização do sistema de execução implementado foi realizado o *benchmarking*, tendo sido efetuados os sete testes propostos para avaliação e comparação da performance, nas vertentes abrangidas. Foram avaliadas as plataformas de computação *serverless AWS Lambda*, *Azure Funtions*, *Google Cloud Functions* e *IBM Cloud Functions/OpenWhisk*. Em todos os testes foram invocadas funções criadas recorrendo-se à utilização de *templates* da *Serverless Framework*, como descrito no capítulo anterior, sendo estas *deployed*, em todas as plataformas de computação *serverless*, utilizando o sistema de execução do *bechmark*, na região de Londres dos respetivos provedores. Foi utilizada esta localização geográfica por nela estarem disponíveis todas as plataformas testadas e por ser a mais próxima, tendo-se desta forma tempos de latência referentes à rede similares entre as diferentes plataformas. Para todas as funções foi ainda definido um *trigger* HTTP utilizado nas suas invocações através de pedidos deste tipo.

Todas as funções utilizadas nos testes, à exceção das do teste T6, por neste serem utilizadas versões de uma mesma função com diferentes valores de memória alocada, e das referentes à *Azure Functions*, por esta plataforma atribuir dinamicamente a memória às funções [18] [45], foram alocadas com 256 MB de memória.

Os testes foram efetuados localmente, ou seja, o sistema de execução do *benchmark* foi utilizada localmente, e foram repetidos várias vezes, de forma a verificar-se padrões nos comportamentos obtidos nos resultados da execução dos testes, e assim tentar eliminar comportamentos originados por perturbações na rede.

Nas sub-secções seguintes são apresentados os testes realizados assim como os resultados obtidos. De salientar que os testes efetuados foram limitados, por exemplo, na duração do período em que são realizadas invocações e no nível de concorrência utilizado, por estarem a ser testadas plataformas comerciais, com um custo associado. Por este motivo foram executados testes que permitiram manter o custo no nível gratuito, de um milhão de invocações por mês, para cada uma das plataformas.

8.1 Termos de serviço e licenças para a realização de *benchmarks* e divulgação de resultados

Antes da realização dos *benchmarks* nas diversas plataformas de computação *serverless*, foram consultados os acordos de licença do utilizador final (EULAs) assim como os termos de serviço dos respetivos provedores, de forma a verificar a existência de algum impedimento para a realização do *benchmarking*.

Nos termos de serviço da *Amazon Web Services (AWS)*[1] é referido que é permitida a realização de *benchmarks* ou de testes comparativos. Para a divulgação de resultados é necessário que seja fornecida informação que permita que os testes de *benchmkmark* sejam replicados por terceiros de forma completa e precisa.

Nos termos de serviço da *Google Cloud*[2] é dito que a divulgação de resultados é possível, se esta incluir informações que permitem à *Google* ou a um terceiro replicar os testes.

Relativamente a *Azure* e à *IBM Bluemix*, não foi encontrada, nos seus termos de serviço, qualquer restrição relativamente à execução e divulgação de resultados de um *benchmark*.

8.2 *T1 - Teste de overhead das plataformas serverless*

O teste T1, de *overhead* das plataformas *serverless*, foi projetado com o objetivo de obter e comparar o *overhead* de cada uma das plataformas. Foi utilizada uma função simples, que apenas retorna um "Hello, World!" e termina a execução, para que o tempo de computação não tenha impacto nos resultados. Foi implementada em *Node.js* por ser uma linguagem de programação comum a todas as plataformas testadas. A função foi *deployed* recorrendo-se à execução do comando "python3 ServerlessBenchmarkAppInterface.py -d serverless_provider 1" para cada um dos provedores das plataformas de computação *serverless*.

Foi realizada uma experiência, em que se procedeu à execução do teste nas 4 plataformas de computação *serverless* comerciais através do sistema de execução do *benchmark*, tendo sido utilizado o comando "python3 ServerlessBenchmarkAppInterface.py -o all 1 20". O sistema realiza invocações sequenciais, isto é, reemitindo um pedido de execução da função imediatamente após ser recebida a resposta do anterior, durante 20 segundos, em cada um dos provedores. Os resultados do teste, calculados pelo sistema a partir das latências das invocações, são apresentados na Tabela 8.1 e também mostrados sob a forma de gráfico na Figura 8.1.

Os resultados apresentados permitem observar que a *Azure Functions*, a *Google Cloud Functions* e a *AWS Lambda* apresentam latências médias por invocação próximas entre si, de cerca de 50 ms, sendo que na *OpenWhisk* este valor é consideravelmente superior,

[1] `https://aws.amazon.com/pt/aispl/service-terms/`
[2] `https://cloud.google.com/terms/`

Provedor:	**OW**	**Azure**	**Google**	**AWS**
Latência Máxima (ms):	385	314	168	103
Latência Mínima (ms):	147	39	40	48
Latência Média (ms):	187.34	48.96	52.70	52.77
Desvio Padrão Latência (ms):	34.16	18.99	22.44	4.72
% de Invocações Com Sucesso:	100	100	100	100
% de Invocações Com Falha:	0	0	0	0
Número de execuções:	98	351	361	332

Tabela 8.1: Resultado da execução do teste T1 por plataforma de computação *serverless*, durante 20 segundos

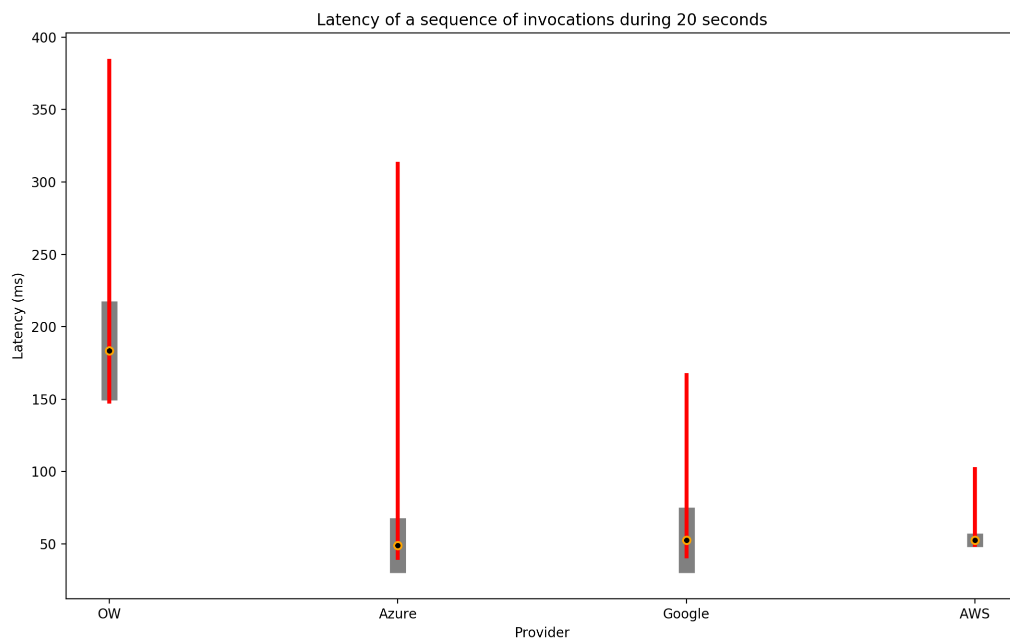


Figura 8.1: Resultado da execução do teste de latência T1 por plataforma de computação *serverless*, durante 20 segundos: Média (círculo a laranja), desvio padrão (retângulo cinzento) e valores máximos e mínimos (barra a vermelho)

de cerca de 187 ms. Significa então que nas três primeiras plataformas obtemos em média tempos de resposta às invocações semelhantes, enquanto que na *OpenWhisk* estes são superiores. Na *AWS Lambda* os tempos de latência entre diferentes invocações são bastante uniformes e estáveis, apresentando um desvio padrão em relação à média de apenas 4.72 ms. Nas restantes plataformas existe uma variação acentuada, sendo este efeito ainda mais visível na *OpenWhisk*, em que o desvio padrão é de 34.16 ms. Todas as invocações realizadas à função da *OpenWhisk* apresentam latências superiores à pior invocação na *AWS Lambda*. Na *Azure* e na *Google*, em algumas das invocações foram obtidas latências equiparáveis às da *OpenWhisk*, sendo que os piores tempos na *Azure* são bastante superiores aos da *Google* e da *AWS* e muito próximos dos piores da *OpenWhisk*. Durante os 20 segundos a *Google* realizou 361 execuções, enquanto que a *OpenWhisk* apenas realizou 98.

De referir ainda que todos as invocações do teste foram efetuados com sucesso, ou seja, as funções foram executadas pelas respetivas plataformas sem qualquer falha. De forma geral podemos dizer que obtivemos, com maior probabilidade, tempos de latência mais baixos na *AWS Lambda* e mais altos na *OpenWhisk* o que permite concluir que tivemos um menor *overhead* na primeira plataforma e um maior na segunda.

8.3 T2 - Teste de carga concorrente

O teste T2, de carga concorrente, tem como objetivo perceber a escalabilidade das plataformas de computação *serverless*. Foi utilizada uma função simples escrita em *Node.js*, que apenas retorna um "Hello, World!". A função foi *deployed* recorrendo-se à execução do comando "python3 ServerlessBenchmarkAppInterface.py -d serverless_provider 2" para cada um dos provedores das plataformas de computação *serverless*.

Procedeu-se à execução do teste nas 4 plataformas de computação *serverless* comerciais, tendo sido utilizado o comando "python3 ServerlessBenchmarkAppInterface.py -c all 2 1 15 1 20". O sistema inicia o teste com apenas um pedido de invocação concorrente e após 20 segundos esse valor é incrementado em uma unidade, até ao máximo de 15. Para cada valor de pedidos concorrentes são efetuadas invocações à função de forma sequencial durante 20 segundos. O teste é repetido, na totalidade, para cada um dos provedores. Os resultados do teste, calculados pelo sistema a partir das latências das invocações e do *throughput*, para cada provedor pode ser visualizado na Figura 8.2. Na Tabela 8.2 encontram-se os dados relativos ao resultado do teste para 15 pedidos concorrentes.

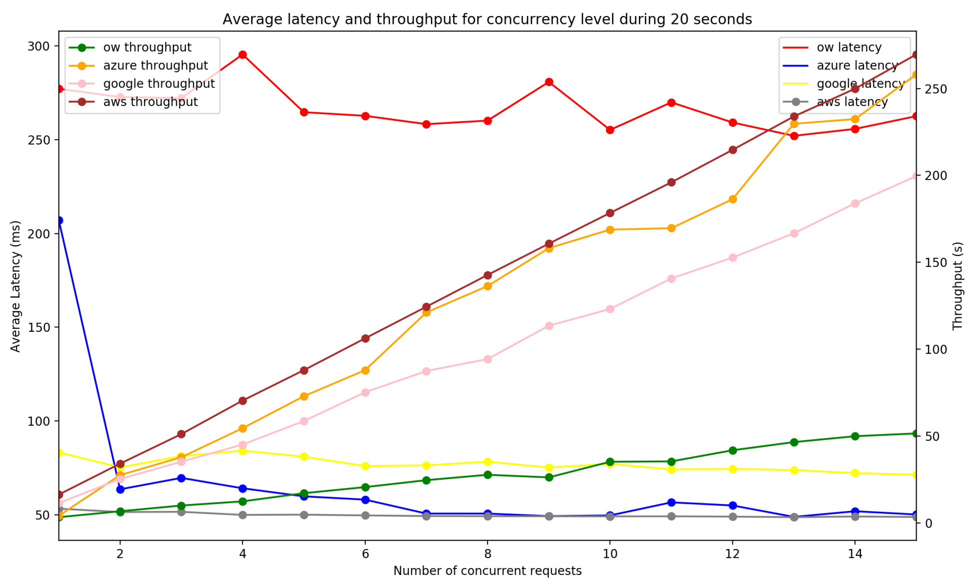


Figura 8.2: Resultado da execução do teste T2: *Throughput* por segundo e latência média por invocação em função do número de pedidos concorrentes efetuados durante 20 segundos

Pela análise do gráfico e da tabela percebe-se que a *AWS Lambda* apresenta uma esca-

Provedor:	**OW**	**Azure**	**Google**	**AWS**
Latência Máxima (ms):	1879	2609	260	131
Latência Mínima (ms):	208	37	47	43
Latência Média (ms):	262.61	50.10	71.30	48.78
Desvio Padrão Latência (ms):	88.79	67.47	38.34	4.45
% de Invocações Com Sucesso:	100	100	100	100
% de Invocações Com Falha:	0	0	0	0
Número de execuções:	1049	5200	4024	5417
Throughput (s):	51.5	258.1	199.6	269.6

Tabela 8.2: Resultado da execução do teste T2 por plataforma de computação *serverless*, durante 20 segundos com 15 pedidos concorrentes

labilidade linear, em que a performance não é afetada pelo crescimento da concorrência, sendo esta a plataforma que apresenta maior *throughput* para todos os níveis de invocações concorrentes. Neste provedor a latência média por invocação mantém-se constante com o aumento da concorrência, sendo a mais baixa ao longo de todo o teste. A performance mantida na *AWS Lambda* face ao aumento da concorrência pode, possivelmente, ser explicada com baixos valores de *overhead* nos *cold containers*, utilização de *containers* pré alocados (pre warmed containers) e uma grande otimização na utilização de *warm/hot containers*.

A *Azure Functions* apresenta uma escalabilidade igualmente linear, muito próxima da conseguida com a *AWS Lambda*, existindo contudo algumas diferenças na observação, como por exemplo, para 7 pedidos concorrentes em que o *throughput* apresenta um crescimento superior e para 11 pedidos concorrentes em que acontece um abrandamento. Neste provedor a latência média por invocação sofre um grande decréscimo ao passar de 1 para 2 pedidos concorrentes, facto que pode dever-se possivelmente a uma maior probabilidade de reutilização de *containers* com o aumento da concorrência ou à utilização de algum tipo de *containers* pré-alocados. Esta é a plataforma que mais se aproxima com a performance conseguida neste teste pela *AWS Lambda*.

A *Google Cloud Functions* exibe uma escalabilidade muito próxima de linear, sofrendo menos desvios do que a *Azure Functions*, embora com uma taxa de crescimento do *throughput* significativamente inferior à das duas plataformas já abordadas. Este facto deve-se a tempos de latência superiores, quando comparados com os da *AWS Lambda* e da *Azure Functions*. Com o aumento do número de pedidos concorrentes, estes tempos tendem a baixar, ainda que de forma bastante lenta.

A *OpenWhisk* mostra uma escalabilidade quase linear, embora com uma taxa de crescimento bastante baixa. A latência média por invocação neste plataforma é bastante alta, quando comparada com as restantes. Dos provedores em que foi realizado o teste, este é a que apresenta uma pior performance face ao aumento da concorrência. Este resultado pode ser explicado pela existência de um maior *overhead* em cada uma das invocações, como se pode verificar pelo teste apresentado na sub-secção anterior, que se mantêm mesmo com a possível reutilização de um maior número de *containers*, indicando assim que pode não existir uma tão boa otimização na utilização de *warm conatiners* se comparando com as

restantes plataformas.

8.4 T3 - Teste de reutilização de *containers*

O teste T3, de reutilização de *containers*, foi projetado com o propósito de se estabelecerem padrões de reutilização de *containers* por parte das plataformas de computação *serverless*, verificando-se o impacto da utilização de *cold containers* nas invocações. Foi utilizada uma função simples, que apenas retorna um "Hello, World!". A função foi *deployed* recorrendo-se à execução do comando "python3 ServerlessBenchmarkAppInterface.py -d serverless_provider 3" para cada um dos provedores das plataformas de computação *serverless*.

Executou-se o teste nas 4 plataformas de computação *serverless* comerciais, tendo-se utilizado o comando "python3 ServerlessBenchmarkAppInterface.py -b all 3 30 600 30 20". O sistema, para cada um dos provedores, inicia o teste realizando uma pré-execução de invocações sequenciais durante 20 segundos com o objetivo de garantir que existam *warm containers* da função nas plataformas de computação *serverless*. Após o passo anterior, são repetidas invocações da respetiva função espaçadas por um tempo incremental de 30 segundos desde a última invocação, iniciando-se com uma espera de 30 segundos e terminando-se em 600 segundos. Para cada invocação, é registada a latência para cada tempo de espera. Esta é utilizada pelo sistema para a geração do gráfico que pode ser visto na Figura 8.3.

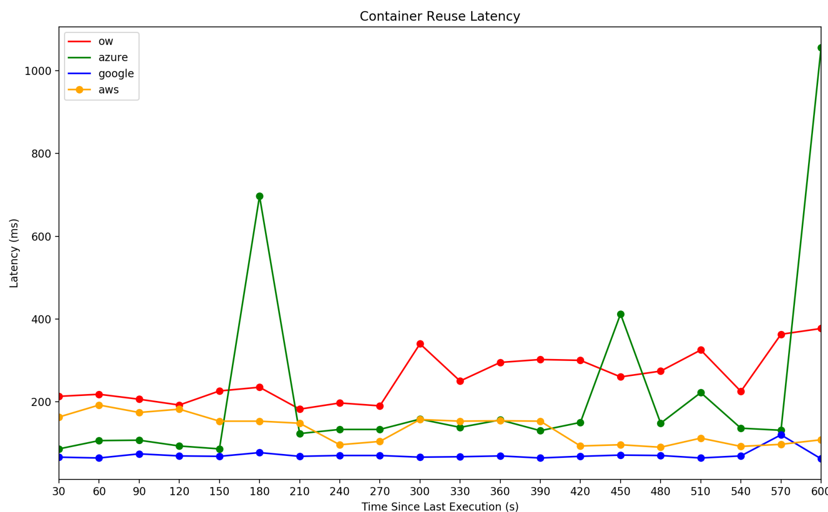


Figura 8.3: Resultado da execução do teste T3: Latência da invocação em função do tempo ocorrido desde a ultima invocação

As latências apresentadas no gráfico em função do tempo ocorrido desde a ultima inovação, para a *AWS Lambda* e para a *Google Cloud Functions* não permitem tirar qualquer informação relativamente ao tempo durante o qual ficam ativos *containers* da função e ao

overhead associado à utilização de um *cold container*. As invocações em ambas as plataformas não parecem ser afetadas pela utilização de funções inativas, ou seja, que não possuam um *hot/warm container* ativo. Possíveis explicações para este comportamento podem ser tempos de instanciação e inicialização de novos *containers* (*cold containers*) extremamente rápidos ou pré-alocação de *containers* por parte dos provedores. Outra possibilidade poderá ser a manutenção dos *warm containers* por períodos superiores aos 10 minutos que foram utilizados como valor máximo no teste efetuado.
Para a *Azure Functions* são visíveis ocorrências de picos, que indica a existência de invocações realizadas com *cold containers*, nos tempos desde a última execução de 180, 450 e 600 segundos. Contudo, não é possível definir o tempo durante o qual a plataforma mantém os *containers* ativos, uma vez que o esperado seria a invocação a *cold containers* sempre a partir do valor em que esta ocorresse a primeira vez. Este comportamento pode indicar que a *Azure Functions* não utiliza um tempo fixo para libertar os *containers*, sendo estes eliminados quando a plataforma necessita dos recursos para executar uma outra função. Neste teste, o *overhead* introduzido da execução da função em *cold containers* varia entre cerca de 300ms e 950ms, sendo esta a plataformas que apresenta uma pior performance neste cenário de execução.
Na *OpenWhisk*, a transição do cenário de reutilização de *containers* para o de criação de um novo, para a execução da função, ocorre aos 300 segundos, ou seja, a plataforma mantém a função ativa durante 5 minutos. O *overhead* introduzido neste processo face à utilização de uma função ativa é de cerca de 150 ms, sendo este valor bastante inferior ao obtido na plataforma da*Azure*.

Em situações em que se utilize uma das plataformas para executar uma função raras vezes e espaçadas por algum tempo, situações que levam a utilização de *cold containers*, deverá ser obtida uma melhor performance na *Google Cloud Funtions* e na *AWS Lambda* e uma pior na *Azure Functions*.

8.5 T4 - Teste de impacto do tamanho do *payload*

Com o teste T4, pretende-se perceber o impacto do tamanho do *payload* na performance das plataformas de computação *serverless*. Utilizou-se neste teste uma função, escrita em *Node.js*, que retorna frases com tamanhos pré-definidos. Foi utilizada esta estratégia ao invés de, por exemplo, uma função que gerasse uma frase com um tamanho pretendido pois, desta forma, ficamos imunes ao tempo de computação da geração das mesmas nas latências obtidas. Foram utilizados tamanhos de *payload* de 0, 32, 64, 96, 128, 160, 192, 224 e 256 Kb, tendo estes sido especificados no ficheiro de configuração do sistema de *benchmark*. Limitou-se a 256 Kb por este ser o tamanho máximo que a *AWS Lambda* pode retornar em uma execução de função [46]. Foi realizado o *deploy*, nas quatro plataformas de computação *serverless* comerciais, recorrendo-se ao comando "python3 ServerlessBenchmarkAppInterface.py -d serverless_provider 4".

Procedeu-se à execução do teste utilizando-se o comando "python3 ServerlessBenchmar-

kAppInterface.py -p all 4 20". Em cada uma das plataformas é realizada uma sequência de invocações à função, durante 20 segundos, para cada tamanho de *payload* definido. As médias de latências por invocação, em função do tamanho do *payload*, são apresentadas no gráfico da Figura 8.4.

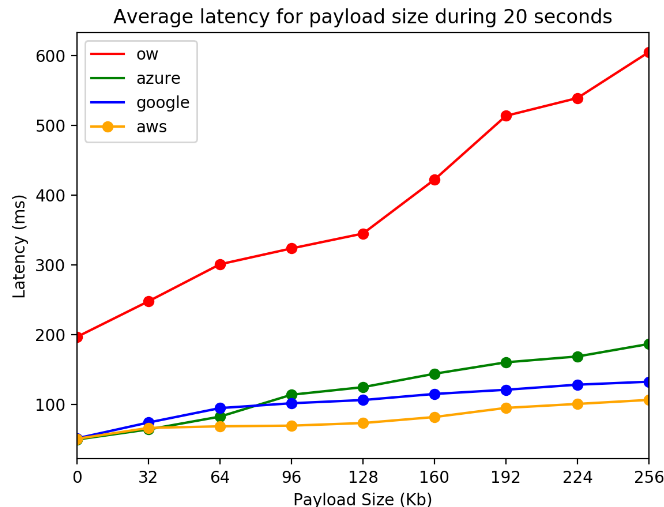


Figura 8.4: Resultado da execução do teste T4: Latência média em função do tamanho do *payload*

Os resultados demonstram que a performance das plataformas de computação *serverless* não parece ser diretamente afetada pelo aumento do *payload*, para os valores de tamanho que foram utilizados, sendo a variação das latências influenciada apenas pelo tempo de transmissão na rede. Esta afirmação é suportada no facto do crescimento das latências obtidas em função do aumento do *payload* seguirem aproximadamente o crescimento linear esperada pela evolução do tempo de transmissão na rede em função do tamanho de *payload* utilizando ligações TCP [47]. A maior taxa de crescimento apresentada na *OpenWhisk* pode, possivelmente, ser justificada por uma menor largura de banda até à infraestrutura do provedor *IBM*, ou dentro da própria infraestrutura entre as componentes que constituem a plataforma de computação *serverless*.

8.6 T5 - Teste de impacto da linguagem de programação na performance

Com o teste T5 pretende-se comparar a latência obtida com a execução de uma função em diferentes linguagens de programação, de forma a identificar se estas têm impacto na performance das plataformas de computação *serverless*.
No momento da realização do presente teste, a *Google Cloud Funtions* apenas suporta *Node.js* como linguagem de programação e a *Framework Serverless* apenas permite o *deploy* de funções escritas também em *Node.js* na *Azure Functions*. Desta forma, as duas plataformas não foram utilizadas na execução do teste.

Na *AWS Lambda* e na *OpenWhisk* foram utilizadas versões da função que apenas retorna um "Hello, World!", escritas em diferentes linguagens de programação. Na primeira plataforma utilizou-se *Node.js, Python, Go* e *Java*. Já na segunda, utilizou-se *Node.js, Python, Swift, Ruby* e *PHP*. O comando utilizado para o *deploy* das diversas funções, nas plataformas da *AWS* e *Azure* foi correspondentemente "python3 ServerlessBenchmarkAppInterface.py -d aws 5" e "python3 ServerlessBenchmarkAppInterface.py -d azure 5".

O teste foi executado, tendo-se utilizado respetivamente os comandos "python3 ServerlessBenchmarkAppInterface.py -l aws 5 20" e "python3 ServerlessBenchmarkAppInterface.py -l ow 5 20". Para cada uma das plataformas, o sistema de execução do *benchmark* realiza uma sequência de invocações, durante 20 segundos, de cada função escrita numa linguagem de programação diferente. Os resultados calculados para a *AWS Lambda* podem ser consultados na Figura 8.5 e na Tabela 8.3. Para a *OpenWhisk* podem ser observados na Figura 8.6 e na Tabela 8.4.

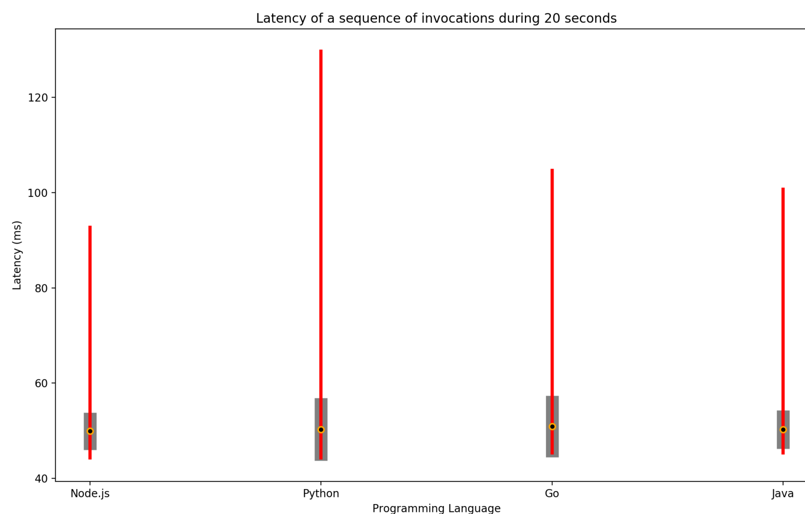


Figura 8.5: Resultado da execução do teste T5 na *AWS Lambda*, durante 20 segundos: Média (círculo a laranja), desvio padrão (retângulo cinzento) e valores máximos e mínimos (barra a vermelho) de latência em função da linguagem de programação

Linguagem de programação:	**Node.js**	**Python**	**Go**	**Java**
Latência Máxima (ms):	93	130	105	101
Latência Mínima (ms):	44	44	45	45
Latência Média (ms):	49.92	50.30	50.95	50.28
Desvio Padrão Latência (ms):	3.90	6.60	6.45	4.04
% de Invocações Com Sucesso:	100	100	100	100
% de Invocações Com Falha:	0	0	0	0
Número de execuções:	350	350	346	350

Tabela 8.3: Resultado da execução do teste T5 na *AWS Lambda*, durante 20 segundos

A análise das latências, obtidas dos resultados da *AWS Lambda* permitem concluir que a utilização das diferentes linguagens de programação não apresenta diferenças de perfor-

mance significativas, uma vez que a média de latências para cada uma das linguagens de programação é bastante próxima, cerca de 50 ms. Nesta plataforma, *Node.js* e *Java* parecem ser as linguagens mais consistentes em termos de latência entre diferentes invocações, uma vez que apresentam um menor valor para o desvio padrão. A função escrita em *Go* apresenta uma maior latência média, um desvio padrão alto e foi executada menos vezes, no mesmo período de tempo, por isso, *Go* pode ser considerada a pior em performance de entre as 4 testadas na *AWS Lambda*.

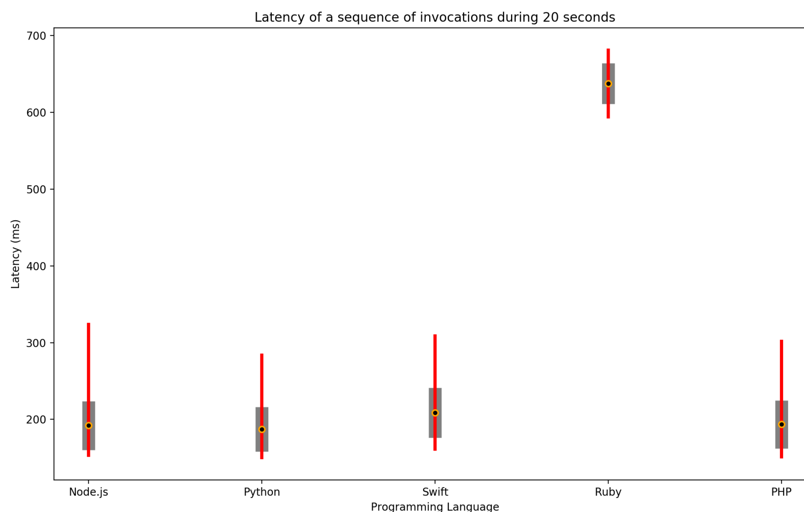


Figura 8.6: Resultado da execução do teste T5 na *OpenWhisk*: Média (círculo a laranja), desvio padrão (retângulo cinzento) e valores máximos e mínimos (barra a vermelho) de latência em função da linguagem de programação

Linguagem de programação:	**Node.js**	**Python**	**Swift**	**Ruby**	**PHP**
Latência Máxima (ms):	326	286	311	683	304
Latência Mínima (ms):	151	148	159	592	149
Latência Média (ms):	192.10	187.29	208.64	637.17	193.60
Desvio Padrão Latência (ms):	31.80	28.98	32.42	26.52	31.16
% de Invocações Com Sucesso:	100	100	100	100	100
% de Invocações Com Falha:	0	0	0	0	0
Número de execuções:	94	98	86	30	97

Tabela 8.4: Resultado da execução do teste T5 na *OpenWhisk*, durante 20 segundos

Na *OpenWhisk* é possível identificar claramente que a linguagem de programação *Ruby* apresenta a pior performance de entra as testadas, mostrando um grande *overhead*, de cerca de 450ms, em relação as outras linguagens de programação. Este facto pode indicar que há uma pobre otimização no ambiente de execução desta linguagem de programação. Das restantes linguagens *Swift* é a que apresenta pior performance, embora bastante melhor que a obtida com *Ruby*, e uma maior variação entre invocações distintas, como se pode comprovar respetivamente pela latência média e pelo desvio padrão. *Node.js*, *Python* e *PHP* apresentam performances bastante idênticas. Na escrita de funções para *OpenWhisk*,

de acordo com este teste, deve ser evitada ao máximo a utilização de *Ruby*.

8.7 T6 - Teste de performance em função da memória alocada

Ao realizar-se o *deploy* de uma função num dos provedores *serverless* poderá ser possível alocar memória à mesma, isto é, definir uma quantidade de memória *RAM* que a função terá disponível durante o seu processo de execução. Em algumas plataformas a capacidade de processamento, CPU, e largura de banda são atribuídas de acordo com a quantidade de memória .

Com o teste T6 pretende-se perceber de que forma a memória atribuída a uma função altera a sua performance. A plataforma *Azure Functions* foi excluída deste teste por atribuir dinamicamente a memória às funções, não permitindo esse controlo pelo utilizador. Nas restantes três plataformas foram utilizadas funções com diferentes valores de memória alocada. O comando utilizado para o *deploy* de todas as funções em cada um dos provedores foi "python3 ServerlessBenchmarkAppInterface.py -d serverless_provider 6".

Numa primeira experiência foi realizado o teste na *AWS Lambda*, na *OpenWhisk* e na *Google Cloud Funtions*, recorrendo-se à utilização de uma função que apenas retorna um "Hello, World!". O comando usado para a execução do teste, nas 3 plataformas, foi respetivamente "python3 ServerlessBenchmarkAppInterface.py -m aws 6 10", "python3 ServerlessBenchmarkAppInterface.py -m ow 6 10" e "python3 ServerlessBenchmarkAppInterface.py -m google 6 10". Para cada uma das plataformas de computação *serverless*, o sistema de execução do *benchmark* realiza uma sequência de invocações, durante 10 segundos, a cada uma das funções com diferentes valores de memória alocada. Os gráficos gerados pelo sistema apresentam para cada função, com diferente valor de memória, a latência de cada uma das invocações efetuadas durante os 10 segundos. Para a *OpenWhisk*, para a *AWS Lambda* e para a *Google Cloud Functions* os resultados podem ser visualizados respetivamente nas Figuras 8.7, 8.8 e 8.9

O gráfico da *OpenWhisk*, Figura 8.7, não permite tirar qualquer conclusão relativa a performance em função da memória alocada, uma vez que não existe nenhum padrão nas latências. Por exemplo, em algumas das invocações, com uma função com 128 MB de memória conseguimos ter a mesma performance que uma com 512MB. Um facto interessante é que as melhores latências são obtidas com a função alocada com menor memória.

Relativamente aos resultados da *AWS Lambda*, gráfico da Figura 8.8, mais uma vez não se consegue ver o impacto que a memória alocada tem na performance da plataforma *serverless* na execução de uma função. Também nesta plataforma conseguimos obter igual performance para os diferentes valores de memória alocada. Algumas das piores latências são obtidos com os dois valores mais altos de memória atribuída, podendo este facto indicar que existe um maior *overhead* nos ambientes de execução que oferecem mais

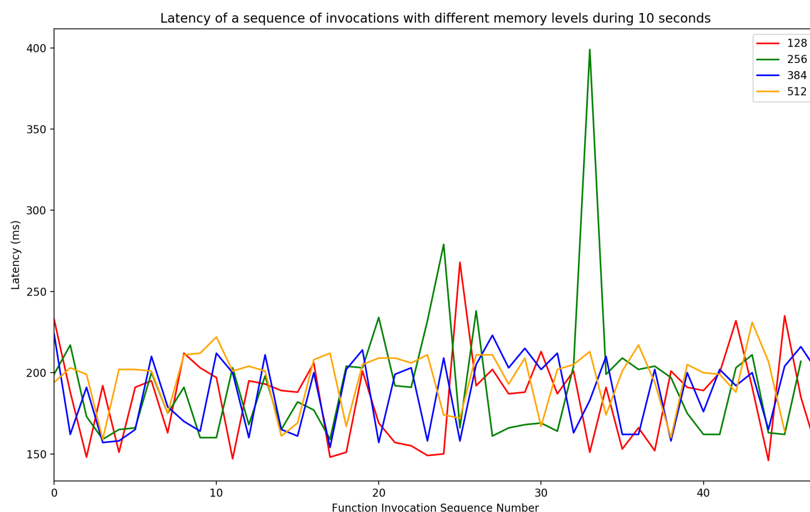


Figura 8.7: Resultado da execução do teste T6 na *OpenWhisk*, durante 10 segundos, utilizando uma função leve computacionalmente: Latência por invocação para cada valor de memória alocada em MB

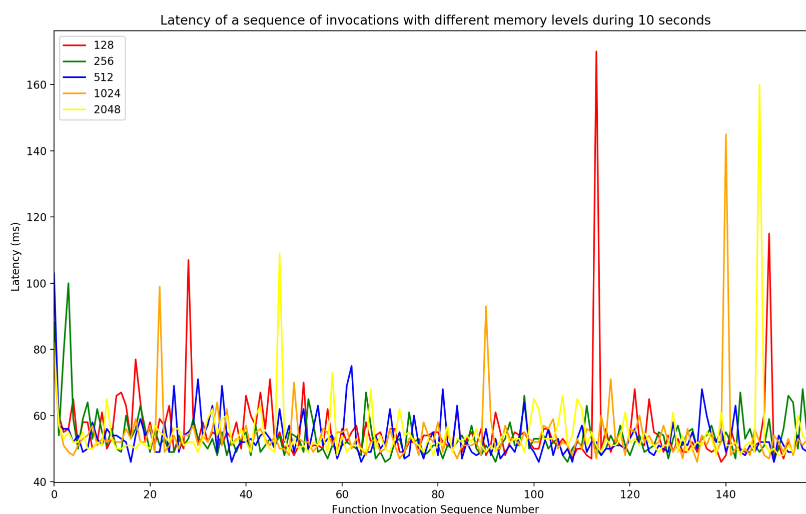


Figura 8.8: Resultado da execução do teste T6 na *AWS Lambda*, durante 10 segundos, utilizando uma função leve computacionalmente: Latência por invocação para cada valor de memória alocada em MB

memória.

Os resultados para a *Google Cloud Funtions*, apresentados na Figura 8.9, não demonstram também nenhum impacto da memória alocada, uma vez que conseguimos obter a mesma performance para uma função definida com o mínimo de memória, 128MB, que a que obtemos com uma atribuída com o máximo de memória, 2048 MB.

Para uma função simples, com pouco peso computacional, não é possível perceber o impacto da memória alocada na performance da sua execução, o que indica que para este tipo

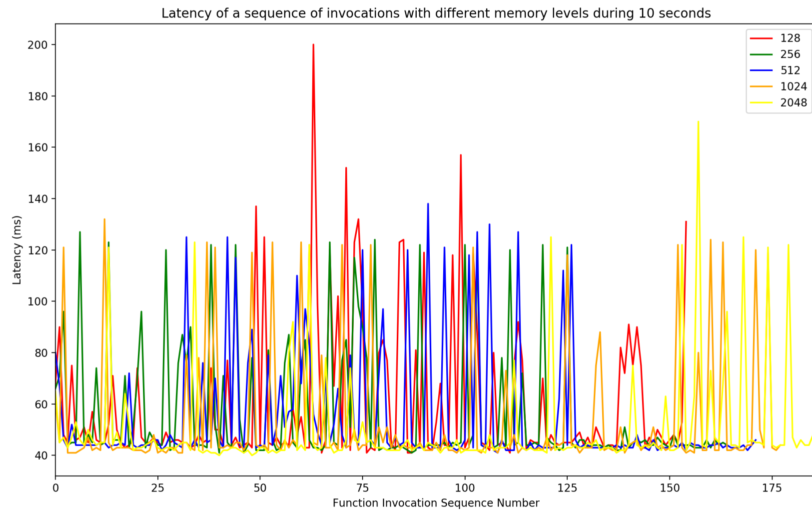


Figura 8.9: Resultado da execução do teste T6 na *Google Cloud Functions*, durante 10 segundos, utilizando uma função leve computacionalmente: Latência por invocação para cada valor de memória alocada em MB

o aumento de memória não leva necessariamente à obtenção de latências mais baixas e que podemos estar a ocupar e a pagar por recursos dos quais não tiramos qualquer vantagem. Tendo em consideração este aspeto, foi executado um novo teste, recorrendo-se desta vez à utilização de uma função recursiva que calcula a sequência de *Fibonacci*. Por ser mais pesada computacionalmente, pode beneficiar de um maior poder computacional, associado ao aumento de memória alocada, que leve a melhorias substâncias na performance.

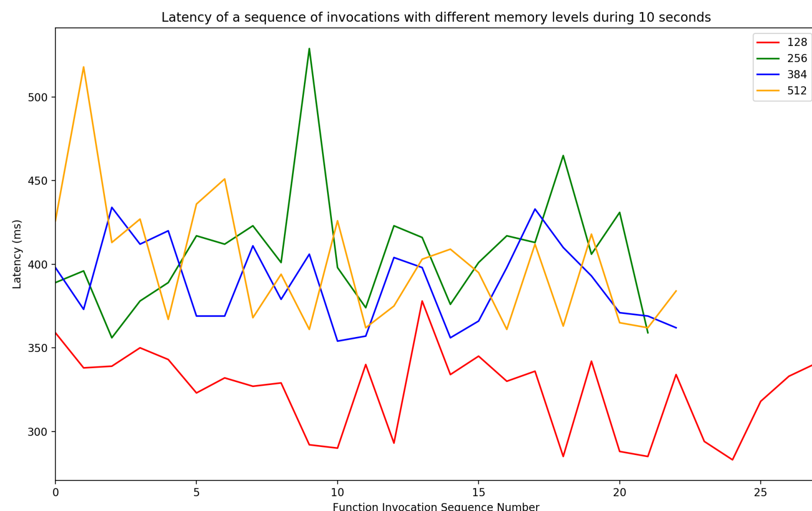


Figura 8.10: Resultado da execução do teste T6 na *OpenWhisk*, durante 10 segundos, utilizando a função de Fibonacci com n=35: Latência por invocação para cada valor de memória alocada em MB

O gráfico da Figura 8.10 apresenta os resultados obtidos para a *OpenWhisk*. A sua análise

permite dizer que nesta plataforma a memória alocada não influencia o poder computacional da plataforma, e que, por isso, um aumento da memória não leva a melhorias na performance obtida. Os resultados mostram ainda que as menores latências são obtidas com a função com menor memória, o que não deixa de ser estranho, podendo significar que há uma menor otimização nos ambientes de execução com maior memória, ou que existe um grande *overhead* na utilização dos mesmos. Outra das possibilidades é a presença de algum tipo de *bug* na plataforma.

Relativamente aos resultados obtidos para a *AWS Lambda* e para a *Google Cloud Funtions*, Figuras 8.11 e 8.12, respetivamente, podemos afirmar que, de facto, o aumento da memória leva efetivamente a uma melhoria na performance. Nestas duas plataformas a memória associada a uma função influência o poder computacional subjacente, como por exemplo, o da capacidade de processamento e de largura de banda que é disponibilizada, como é descrito no capítulo 4 para a *AWS Lambda*. Outra observação que se pode retirar é que nestas duas plataformas, diferentes invocações com o mesmo valor de memória alocada apresentam tempos de latência bastante uniformes entre si, não existindo nelas degradações de performance substanciais.

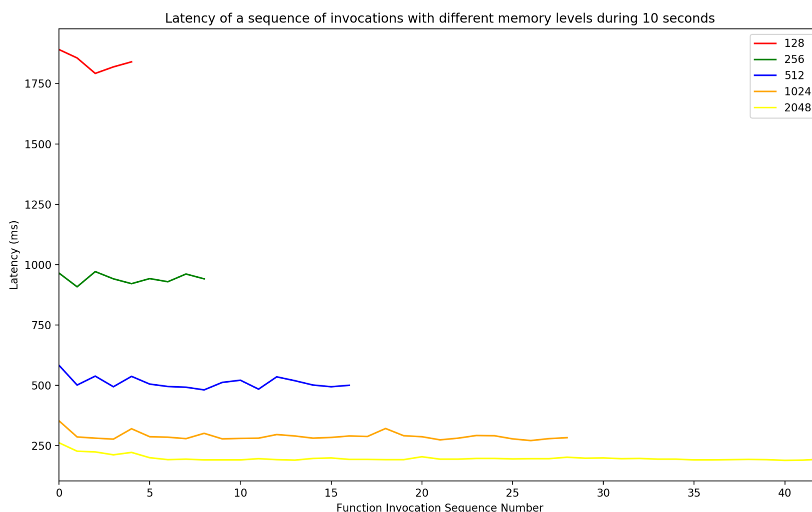


Figura 8.11: Resultado da execução do teste T6 na *AWS Lambda*, durante 10 segundos, utilizando a função de Fibonacci com n=35: Latência por invocação para cada valor de memória alocada em MB

A realização destes dois testes permitiu concluir que na *AWS Lambda* e na *Google Cloud Functions* a capacidade computacional (capacidade de processamento e largura de banda) é atribuída de acordo com a memória alocada à função a ser executada, contudo, a melhoria na performance só será conseguida para funções que necessitem de facto da capacidade computacional que lhe foi atribuída. Por isso, para cada caso deve ser realizado um teste com os diferentes níveis de memória, de forma a verificar-se o custo e o benéfico da alocação de um determinado nível de memória compensa ou não. No caso da *OpenWhisk*, de acordo com os dois testes realizados, deve ser sempre utilizado o menor valor de memória, 128

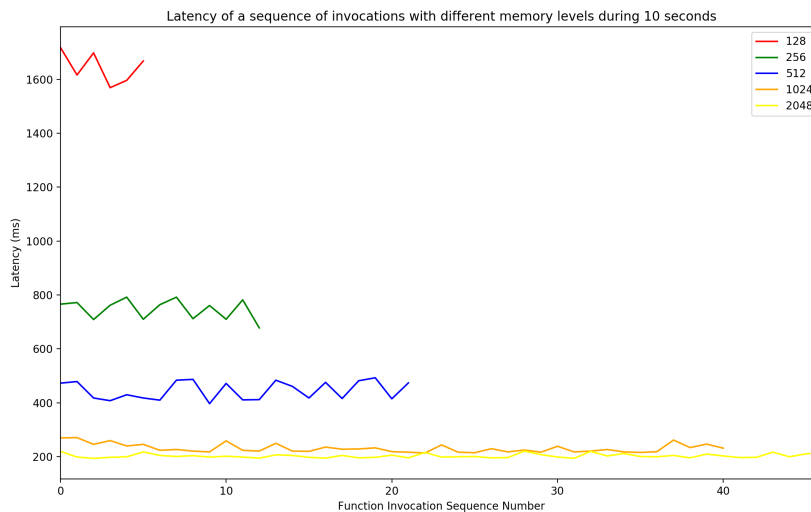


Figura 8.12: Resultado da execução do teste T6 na *Google Cloud Functions*, durante 10 segundos, utilizando a função de Fibonacci com n=35: Latência por invocação para cada valor de memória alocada em MB

MB, pois foi o que apresentou menores latências e é o que têm o menor custo.

8.8 T7 - Teste de performance para execução de funções computacionalmente pesadas

O teste T7 pretende verificar e comparar como se comportam as plataformas de computação *serverless*, em termos de performance, quando executam funções computacionalmente pesadas. Foi utilizada uma função recursiva que calcula a sequência de *Fibonacci* e que recebe como parâmetro o "n" da sequência que se pretende calcular. Para a implementação da função foi utilizada a linguagem de programação *Node.js*. Para o seu *deploy* em cada uma das plataformas de computação *serverless* foi usado o comando"python3 ServerlessBenchmarkAppInterface.py -d serverless_provider 7".

Foi realizada a execução do teste nas 4 plataformas de computação *serverless* comerciais através do comando "python3 ServerlessBenchmarkAppInterface.py -w all 7 20": Para cada um dos provedores, o sistema realiza invocações sequenciais, isto é, reemitindo um pedido de execução da função imediatamente após ser recebida a resposta do anterior, durante 20 segundos, para cada um dos valores de "n" da sequência de *Fibonacci* definidos no ficheiro de configuração do sistema de execução do *benchmark*. Utilizaram-se neste teste os seguintes valores: 0; 5; 10; 15; 20; 25; 30; 35; 40. Os resultados do teste, calculados pelo sistema a partir das médias das latências das invocações para cada valor de "n", são apresentados sob a forma de gráfico na Figura 8.13.

Pela análise do gráfico verifica-se que as latências obtidas seguem aproximadamente o esperado para um algoritmo com complexidade temporal O(2^n), logo, as plataformas de

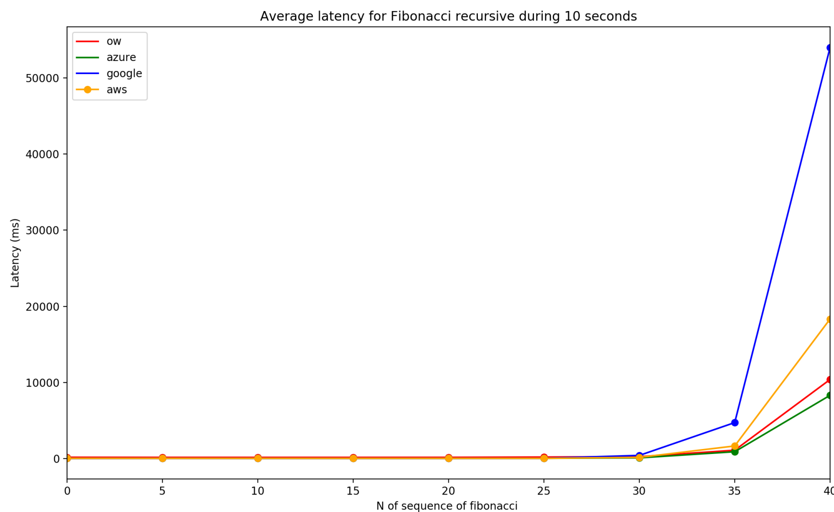


Figura 8.13: Resultado da execução do teste T7: Média da latência por invocação em função do N da sequência de *Fibonacci* utilizado

computação *serverless*, para funções computacionalmente pesadas, reagem de acordo com a respetiva complexidade temporal. É ainda possível visualizar que o ritmo do crescimento em função do aumento do "n" é bastante superior na *Google Cloud Funtions*, o que pode indicar que o poder computacional em termos de capacidade de processamento é inferior nesta plataforma em relação às restantes. No caso da *OpenWhisk* e da *Azure Functions*, o crescimento é bem mais suave, possivelmente devido a uma maior capacidade computacional. Para o "n" de *Fibonacci* 40, valor mais alto utilizado no teste, a *Azure Functions* é a plataforma que apresenta uma menor latência. Uma vez que esta plataforma aloca dinamicamente a memória à função, poderá também alocar capacidade computacional, o que explica esta alta performance. No caso da *OpenWhisk*, que demonstrou nos restantes testes possuir tempos de latência superiores às restantes plataformas, neste consegue ter latências muito próximas da *Azure Functions*, o que poderá dever-se a uma alta capacidade computacional, que permite que com o tempo ganho no tempo de execução atenue-se o *overhead* inerente à própria plataforma, na execução de funções computacionalmente pesadas. Outra das possibilidades é a *OpenWhisk* possuir uma rede com capacidade inferior à das restantes plataformas, o que explica a obtenção de piores resultados nos testes anteriores e de resultados bastante positivos neste teste, uma vez que os resultados deste teste são mais influenciados pelo tempo de computação do que pelo de transmissão na rede.

8.9 Comparação de performance entre uma instalação local da *OpenWhisk* e a *OpenWhisk* da IBM

Uma vez que a *OpenWhisk* é *open source*, é possível instalá-la localmente, de forma a realizar alguns testes para que se perceba o impacto que a infraestrutura, sobre a qual a plataforma corre, apresenta na sua performance. Os resultados obtidos foram comparados com os obtidos na execução, desses mesmos testes, na plataforma *OpenWhisk* da IBM. Procedeu-se a instalação da *OpenWhisk* numa máquina virtual *Linux*. Esta possui um ambiente com 4 cores de CPU, 4GB de memória RAM e 64GB de disco. Os passos seguidos no processo de instalação da plataformas podem ser vistos em `https://github.com/apache/incubator-openwhisk#native-development`. De forma a perceber-se qual o número máximo de *containers* de execução de funções suportados pela instalação local, realizaram-se invocações com um alto valor de concorrência, e visualizaram-se , repetidas vezes, os *containers* existentes na máquina da instalação. O máximo obtido foi de 4 *containers* ativos para a função em execução e 2 *containers* pré alocados para a linguagem de programação utilizada, *Node.js*, como se pode ver na Figura 8.14.

```
root@hjmart:~# docker -H unix:///var/run/docker.sock ps
CONTAINER ID    IMAGE                                COMMAND                CREATED          STATUS
                  NAMES
a29d021d45e9    openwhisk/nodejs6action:latest       "/bin/sh -c 'node --…"  2 seconds ago    Up Less than a second
                  wsk0_1723_guest_hellojs
9fe008d9a108    openwhisk/nodejs6action:latest       "/bin/sh -c 'node --…"  2 seconds ago    Up Less than a second
                  wsk0_1722_guest_hellojs
f20bba664391    openwhisk/nodejs6action:latest       "/bin/sh -c 'node --…"  3 seconds ago    Up Less than a second
                  wsk0_1721_guest_hellojs
1f9710da174e    openwhisk/nodejs6action:latest       "/bin/sh -c 'node --…"  3 seconds ago    Up 1 second
                  wsk0_1720_guest_hellojs
025a0202f5d9    openwhisk/nodejs6action:latest       "/bin/sh -c 'node --…"  3 seconds ago    Up 1 second
                  wsk0_1718_prewarm_nodejs6
72c09a339cfe    openwhisk/nodejs6action:latest       "/bin/sh -c 'node --…"  3 seconds ago    Up 1 second
```

Figura 8.14: *Containers* ativos de uma função e pré alocados para a linguagem de programação na instalação local da *OpenWhisk*

Foi executado o teste T1 nas duas plataformas, seguindo-se exatamente o mesmo processo que o descrito na sub-secção 8.2. Os resultados calculados pelo sistema a partir das latências das invocações realizadas de forma sequencial, num período de 20 segundos, em ambos os provedores, IBM e local, são apresentados na Tabela 8.5 e também mostrados sob a forma de gráfico na Figura 8.15.

Provedor:	**OW IBM**	**OW Local**
Latência Máxima (ms):	345	1061
Latência Mínima (ms):	237	109
Latência Média (ms):	269.93	510.59
Desvio Padrão Latência (ms):	26.20	354.27
% de Invocações Com Sucesso:	100	100
% de Invocações Com Falha:	0	0
Número de execuções:	68	39

Tabela 8.5: Resultado da execução do teste T1 na *OpenWhisk* da IBM e na local, durante 20 segundos

A análise dos resultadas, do teste T1, permite perceber que a latência média por invocação

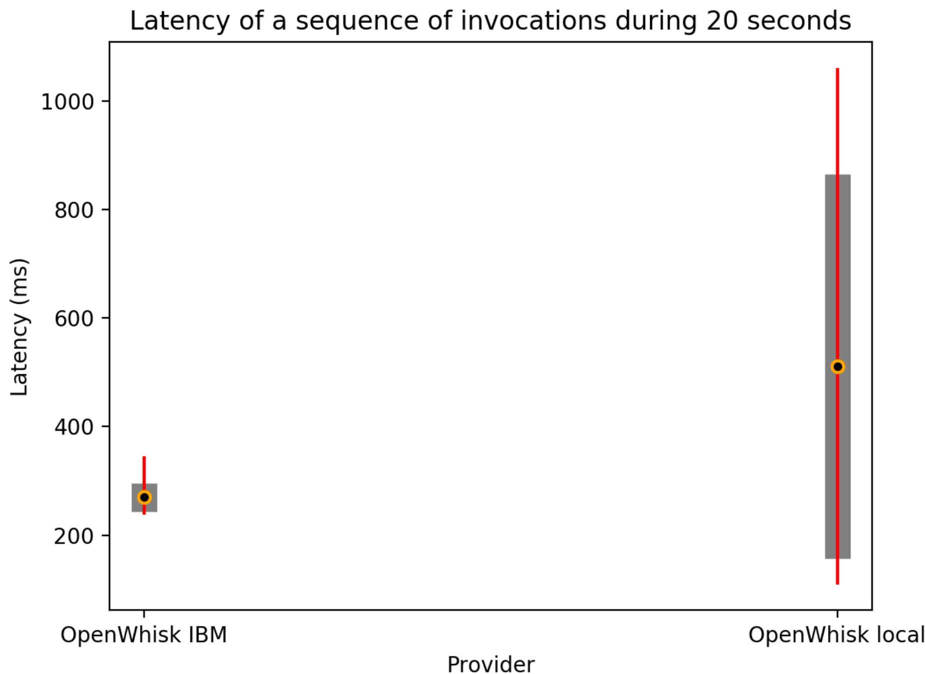


Figura 8.15: Resultado da execução do teste T1 na *OpenWhisk* da IBM e na local, durante 20 segundos: Média (círculo a laranja), desvio padrão (retângulo cinzento) e valores máximos e mínimos (barra a vermelho) de latência

na *OpenWhisk* local é aproximadamente o dobro da apresentada na plataforma da *IBM*. Outro dos aspetos a retirar é que na instalação local a diferença entre a latência de diferentes invocações é bastante dispersa, uma vez que apresenta um alto valor de desvio padrão. Isto significa que entre diferentes invocações podemos obter tempos de latência bastante variáveis. Uma observação interessante é que em algumas das invocações conseguimos obter valores de latência bastante mais baixos na nossa plataforma do que na da *IBM*. Este facto pode dever-se às latências ocorridas aquando da execução da função utilizando um *hot container*. No geral, a degradação de memória na plataforma local pode dever-se a constante destruição dos *containers*, devido aos poucos recursos computacionais que possui, uma vez que a plataforma liberta com frequência os que tem ativos, para que possa ter capacidade disponível para responder a possíveis invocações a outras funções. Esta ação explica também o alto desvio padrão já abordado.

Numa tentativa de se tentar perceber como a instalação local se comporta em escala, com o aumento da carga concorrente, foi executado o teste T2 nas duas plataformas, seguindo-se exatamente o mesmo processo que o descrito na sub-secção 8.3. Os resultados do teste, calculados pelo sistema a partir da latência média e do *throughput* das invocações realizadas de forma sequencial, num período de 20 segundos, para cada valor de carga concorrente, são apresentados para a instalação local da *OpenWhisk* na Figura 8.16 e para a da *IBM* na Figura 8.17.

Analisando os resultados do teste T2, podemos verificar que para a *OpenWhisk* da IBM

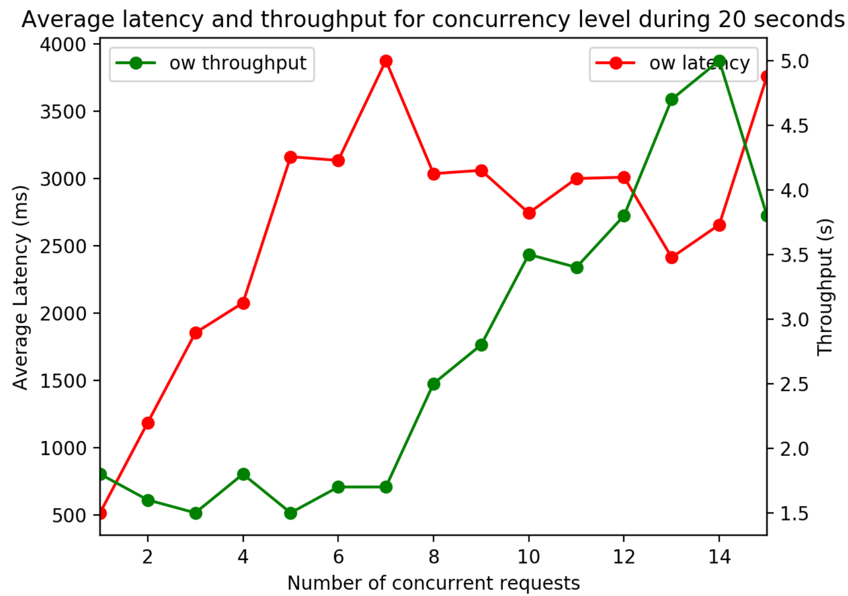


Figura 8.16: Resultado da execução do teste T2 na instalação local da *OpenWhisk*: *Throughput* por segundo e latência média por invocação em função do número de pedidos concorrentes efetuados durante 20 segundos

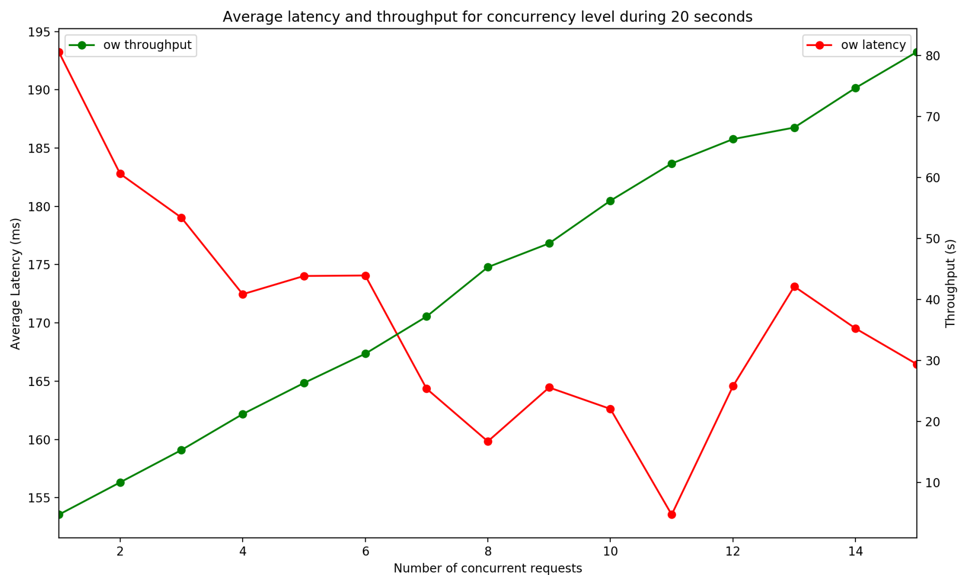


Figura 8.17: Resultado da execução do teste T2 na *OpenWhisk* da IBM: *Throughput* por segundo e latência média por invocação em função do número de pedidos concorrentes efetuados durante 20 segundos

mostra uma escalabilidade quase linear, em função do aumento da concorrência, sendo que a latência média por invocação tende a baixar à medida que a concorrência aumenta. Este comportamento é explicado pelo facto de quanto for maior o nível de concorrência, maior será a probabilidade de existir um *hot container* da função ativo, o que permite que o *overhead* relativo à utilização de um *cold container* seja eliminado. Relativamente

aos dados relativos à instalação local, podemos observar um baixo *throughput* até ao nível de concorrência 8 e um forte aumento nos tempos de latência. Este comportamento pode dever-se ao facto da plataforma lançar múltiplos novos *containers* antes de começar a reutiliza-los, e desta forma existe em cada uma das invocações o *overhead* da criação dos *containers*. A partir deste valor de concorrência, o *throughput* escala de forma bastante rápida e a latência estabiliza, sendo que para este comportamento contribui a reutilização de *containers* para a realização da execução das funções. Como já foi explicado, a plataforma só suporta 6 *containers* ativos, 4 da função e dois pré criados para a linguagem de programação. Uma vez que o máximo de *throughput* conseguido foi de 5 respostas por segundo para 14 invocações concorrentes, pode-se ter atingido o limite da plataforma para este valor de pedidos concorrentes. A diminuição do *throughput* e o aumento da latência média no nível 15 de concorrência pode dever-se à libertação de *containers* ativos por parte da plataforma, por esta ter atingido o seu limite, necessitando de ter recursos disponíveis para o caso de ter de executar uma outra função.

Como se pode verificar por estes dois testes realizados na instalação local da *OpenWhisk*, a capacidade computacional que as plataformas de computação *serverless* têm afeta bastante a performance que podem oferecer. Este efeito é bastante visível a nível da reutilização de *containers*, que é um mecanismo bastante importante para a obtenção de uma melhor performance nestas plataformas.

8.10 Conclusões finais da execução do *benchmark*

Da realização do *benchmarking*, através da execução do conjunto de testes nas plataformas de computação *serverless AWS Lambda*, *OpenWhisk*, *Azure Functions* e *Google Cloud Functions* é possível retirar algumas conclusões. As plataformas *AWS*, *Azure* e *Google* apresentam *overheads* baixos, ao contrário da *OpenWhisk*, sendo que na primeira há uma maior consistência entre diferentes invocações. A plataforma que melhor lida com o aumento de carga concorrente é a *AWS Lambda* e a pior é a *OpenWhisk*, mantendo sempre altas latências, se comparada com as restantes plataformas. Nos testes efetuados, a *AWS Lambda* e a *Google Cloud Funtions* não parecem sofrer degradação substancial de performance com a utilização de *cold containers*. A *OpenWhisk* elimina os *warm containers* após 5 minutos de inatividade, sofrendo de um aumento de cerca de 150 ms na latência com a utilização de um *cold container*. Na *Azure Functions* não foi possível identificar o tempo durante o qual os *containers* permanecem ativos contudo, é a plataforma que mais perda de performance apresenta em situação de *cold container*. Em todas as plataformas, o tamanho de *payload* não parece afetar diretamente as suas performances. Relativamente às linguagens de programação, apenas a utilização de *Ruby* na *OpenWhisk* afetou de forma considerável a performance obtida na execução da função. A memória alocada a uma função computacionalmente leve pode não levar a uma melhoria de performance na *AWS Lambda* e na *Google Funtions*. Já para funções computacionalmente pesadas a memória alocada influencia diretamente a performance, o que permite concluir

que a quantidade de memória associada a uma função, nestas duas plataformas, define capacidade de processamento computacional. Já para a *OpenWhisk*, os resultados obtidos são bastante estranhos, uma vez que a melhor performance é conseguida com o menor valor de memória alocada, o que indica claramente que este valor não tem impacto nos recursos computacionais disponíveis para execução das funções.

Portanto, pelos testes realizados, a *OpenWhisk* foi a que apresentou pior performance. Contudo, tendo em conta os resultados do teste T6, caso se tivessem usado funções definidas com 128 MB nesta plataforma, ao invés dos 256 MB, teríamos possivelmente obtido uma performance mais próxima da obtida nas restantes plataformas.

Outro aspeto a considerar é que as funções foram executadas na região de Londres dos respetivos provedores, sendo que seria interessante avaliar a performance entre diferentes regiões, para que num cenário de utilização real se utilize a melhor. Apesar desta avaliação não ter sido realizada, o nosso sistema de execução de *benchmark* permite efetua-la, sendo apenas necessário alterar a região no ficheiro configuração das respetivas funções, realizar os respetivos *deploys* e realizar os testes pretendidos, sendo que serão utilizadas para invocação as funções implementadas nas novas regiões.

Relativamente ao tempo de execução dos testes realizados, o teste T3, por ter pausas entre cada uma das invocações, demora um tempo considerável a ser realizado, tendo para os valores temporais utilizados demorado cerca de 8 horas a estar completo. Cada um dos restantes testes, com os períodos de invocação utilizados, demorou tempo na ordem de alguns minutos para ser efetuado.

Esta página foi propositadamente deixada em branco.

Capítulo 9

Demonstração da utilização da plataforma *serverless* no contexto do MobiWise

No contexto do projeto *MobiWise*[1] é pretendida a utilização de uma plataforma de computação *serverless* para a criação de serviços e aplicações que fornecerão soluções inovadoras de mobilidade para as pessoas, a partir de dados recolhidos por sensores, em cidades inteligentes (*smart cities*). A plataforma deve permitir aos programadores desenvolver aplicações que interajam com uma vasta quantidade de sensores e atuadores, com um esforço mínimo, através da criação de *workflows* que reutilizem serviços, aplicações e funções. Para além disto, a plataforma de computação e *workflows serverless* deve permitir que se escrevam programas simples e compreensíveis, capazes de serem executados de formas transparente, a partir de vários locais.

Tendo em conta as considerações acima apresentadas e pelo estudo realizado as várias plataformas de computação *serverless*, optou-se pela utilização da *OpenWhisk* por ser uma plataforma *open source*, que pode ser instalada numa *cloud* privada. Para além disto, suporta *workflows serverless* com a utilização da sua poderosa ferramenta *Composer*[2], que conta com uma interface gráfica que, para além de permitir a criação e gestão de *workflows*, permite também a criação, edição e gestão das funções *serverless*. Suporta ainda uma vasta gama de linguagens de programação para a escrita de funções. Relativamente às questões de performance, apesar de ter sido a plataforma que apresentou piores resultados no *bechmarking* executado, como foi mencionado no capítulo anterior, utilizando-se funções configuradas com 128 MB de memória pode-se obter uma performance mais próxima da apresentada pelas restantes plataformas comerciais.

Para demonstrar a utilização da *OpenWhisk*, no contexto da *MobiWise*, para a criação de um novo serviço de mobilidade para o ambiente de uma cidade inteligente, foi considerado

[1] `http://mobiwise.av.it.pt/`
[2] `https://github.com/ibm-functions/composer`

o problema de controlar fluxos de tráfego e encontrar rotas para os mesmos de acordo com vários critérios de otimização, levando em consideração o movimento de veículos presente nas ruas e as informações sobre poluentes emitidos por cada um dos veículos e presentes em cada rua. Os fluxos de tráfego podem ser controlados numa cidade, impactando positivamente nos tempos de trânsito, nas distâncias percorridas, na poluição emitida em cada uma das ruas e na segurança.

Tendo em conta o problema apresentado, procedeu-se à criação de um serviço de demonstração, que foi apresentado no *workshop* intermédio do projeto[3], e que tem como objetivo a otimização das rotas de acordo com um parâmetro para um fluxo de veículos que desejam ir de um ponto para outro numa cidade. No momento da criação desta demonstração, apenas se encontrava implementada, no algoritmo de otimização utilizado, que foi desenvolvido por outros participantes do projeto, a otimização por distância total percorrida por esse fluxo de veículos ou por tempo total gasto nas viagens. No futuro, será implementada a otimização da poluição. Para além disto, como ainda não se tinha acesso a uma infraestrutura de sensores e atuadores, que permitisse, para uma determinada rede rodoviária, obter dados de trânsito e de poluição, optou-se pela utilização do simulador de mobilidade urbana "SUMO"[4]. Este permitiu a definição de uma rede rodoviária, através do desenho de um conjunto de interseções e de vias de trânsito, que pode ser vista na figura 9.1 e sobre a qual se definiram fluxos de veículos que pretendem transitar de um ponto para outro ponto da rede. Ao ser corrida uma simulação é possível obter, posteriormente os dados relativos à rede, os dados relativos a cada um dos veículos que transitaram, bem como a posição e a emissão de poluentes em cada um dos momentos de simulação.

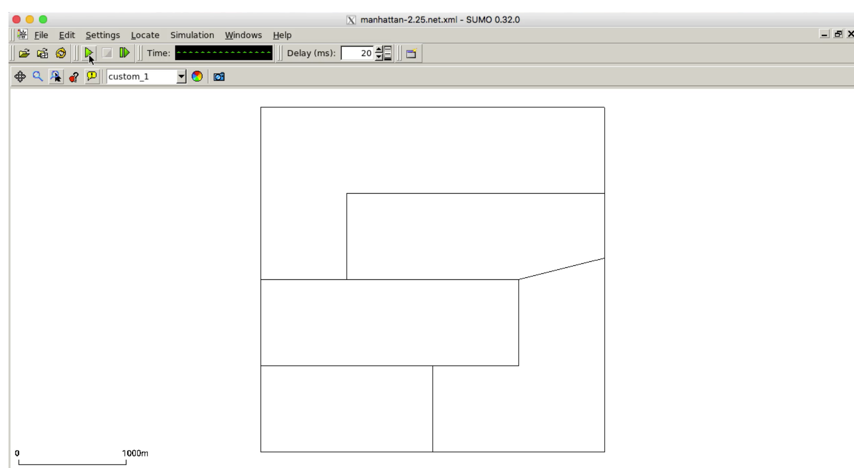


Figura 9.1: Exemplo de rede rodoviária criada no *software* SUMO

O serviço foi elaborado através da definição do *workflow* da figura 9.2, que foi criado com a utilização de funções *serverless* da *OpenWhisk* e da sua ferramenta *Composer*. O *Workflow* inicia recolhendo os dados da simulação efetuada no SUMO, utilizando o algoritmo de distribuição de tráfego do próprio simulador, simulando a obtenção de dados dos sensores, a sua filtragem e conversão para um formato aceite pelo algoritmo de otimização.

[3] `http://mobiwise.av.it.pt/workshop/`

[4] `http://sumo.dlr.de/userdoc/Networks/SUMO_Road_Networks.html`

Isto foi conseguido através da implementação de quatro funções *serverless*, sendo elas as seguintes: "NodesToCSV", que pode ser vista na figura 9.3: obtém, filtra e converte para *csv* os dados relativos aos nós que constituem a rede; "NetworkToCSV": obtém, filtra e converte para *csv* os dados relativos as ruas que constituem a rede; "TripInfoToCSV": obtém, filtra e converte para *csv* os dados relativos ao percurso realizado por cada um dos veículos; "EmissionToCSV": obtém, filtra e converte para *csv* os dados relativos aos poluentes emitido por cada veículo a cada momento da simulação. Através da função "RunOptimization", com os dados obtidos invoca-se o algoritmo de otimização, que os irá utilizar para calcular a melhor rota, para cada um dos veículos pertencentes ao fluxo, em função do parâmetro a otimizar, que pode ser a distância ou o tempo. Após o passo anterior, o *workflow* do serviço obtém as rotas otimizadas e executa uma nova simulação utilizando-as, como pode ser visto na figura 9.4 para cada um dos parâmetros, simulando assim, por exemplo, o envio de uma rota para um dispositivo GPS de um veículo. Caso seja pretendido, aquando da invocação do serviço da simulação, são obtidos dados de desempenho que permitem a avaliação do algoritmo de otimização, como os mostrados na figura 9.5, em que se pode verificar que para a variável a ser otimizada o valor é inferior se comparado com a outra opção.

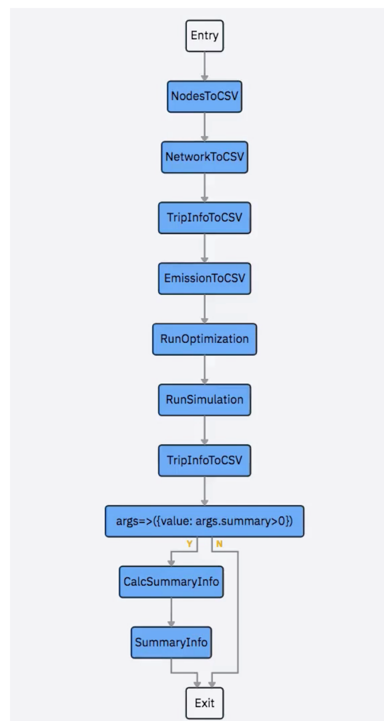


Figura 9.2: *Workflow serveless* do serviço, criado com a utilização da *Openwhisk* e da sua ferramenta de *Workflows Composer*

ACTION

NodesToCSV

v0.0.25

WEB ACCESSIBLE

This is a python:2 action

```python
import xml.etree.ElementTree as ET
import requests
import json


def main(args):
    net_name = args.get('netname')
    simulationname = args.get('initialsimulationname')
    payload = {'name': net_name}
    r = requests.get('http://hjmart2.lis.ipn.pt:8080/api/v1/network', params=payload)
    response = json.loads(r.text)
    xms = response.get('networkxml')

    root = ET.fromstring(xms)


    stringr = "node_id\tx\ty\n"
    for child in root:
        if child.tag == "junction" and child.attrib.get("type") != "internal":
            stringr += "" + child.attrib.get("id") + "\t" + child.attrib.get("x") +
"\t" + child.attrib.get("y")
            stringr +='\n'

    payload = {'filename': simulationname+".nod.csv", 'data': stringr}
    headers = {'content-type': 'application/json'}

    r = requests.post('http://hjmart2.lis.ipn.pt:8080/api/v1/data',
data=json.dumps(payload), headers=headers)
    response = json.loads(r.text)
    filename = response.get('filename')
    return args
```

Figura 9.3: Exemplo de função *serverless* da *OpenWhisk* utilizada no serviço

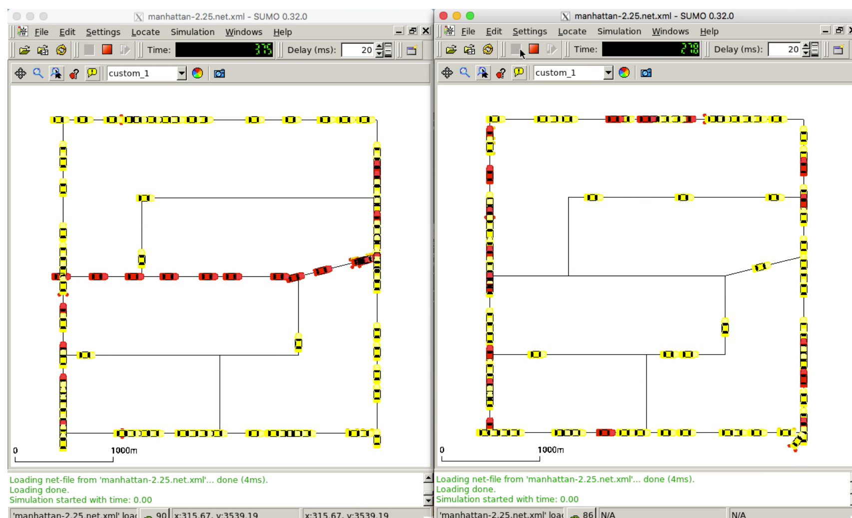


Figura 9.4: Execução da simulação com as rotas otimizadas pelo serviço, de acordo com a distância (à esquerda) e com o tempo (à direita). Veículos amarelos possuem percursos fixos e os vermelhos seguem as rotas otimizadas pelo serviço

Simulated		
Variable	**Time**	**Length**
Length	319440,9	**310398,5**
Time	**13974**	14823
PMx	5357,17	5411,15
CO	722303,31	722443,76
CO2	60134451,66	60792621,48
Fuel	23974,79	24237,18
NOx	123090,84	124025,68

Figura 9.5: Comparação dos dados de distância, tempo e poluentes obtidos das simulações das rotas geradas pela otimização em função da distância total e do tempo total gasto em viagem

A demonstração realizada, recorrendo-se a utilização da plataforma *OpenWhisk*, permitiu perceber que a plataforma pode ser usada para implementar serviços reais de soluções de mobilidade, que necessitem de recolher informação de várias fontes, como de um conjunto de sensores, centralizando a computação na *cloud*, permitindo também integração e orquestração de serviços externos.

Esta página foi propositadamente deixada em branco.

Capítulo 10

Conclusão

Com o trabalho desenvolvido durante o estágio, pretendeu-se numa primeira fase elucidar o leitor acerca daquilo que é a computação *serverless* também conhecida por *Function-as-a-Service (FaaS)*. Sendo esta um modelo de serviço de computação em *cloud*, este paradigma é também apresentado, assim como a tecnologia de *software* de *containers*, essencial para o modelo *serverless*.

Foi de seguida, realizado um estudo sobre as plataformas *serverless* existentes, ou seja, plataformas que oferecem o modelo de computação *serverless*, sendo estas apresentadas, assim como algumas das suas características de operacionalidade, funcionalidades e limitações. Propôs-se um conjunto de parâmetros que permitiu a comparação funcional destas plataformas, tendo sido apresentadas algumas conclusões.

Posteriormente, foi feita uma análise aos estudos de performance e *benchmarkings* existentes, os quais se revelaram lacunares, nomeadamente, a nível de completude e abrangência. Tendo em consideração estas limitações, foi efetuada uma proposta de *benchmarking* modular. Esta contempla um conjunto de testes que irão permitir avaliar e comparar a performance das plataformas em múltiplas vertentes, especialmente, medindo-se o *overhead* introduzido, verificando como se comportam com o aumento da carga concorrente, como é utilizada a otimização a nível de utilização e reutilização de *containers* e de que forma o tamanho de *payload*, a linguagem de programação e a memória alocada afeta a performance. De forma a automatizar o processo de execução do *benchmark* é apresentada uma arquitetura que serviu como base ao desenvolvimento do sistema.

O sistema de execução do *benchmark* implementado permite o *deploy* de funções e a execução do conjunto de testes proposto nas plataformas de computação *serverless AWS Lambda*, *Google Cloud Functions*, *Azure Functions* e *OpenWhisk*. Recorrendo-se à sua utilização efetuou-se o *benchmarking* nas quatro plataformas, tendo-se ainda efetuado uma comparação entre a *OpenWhisk* da IBM e uma instalação local da mesma. Os resultados obtidos permitiram tirar algumas conclusões, nomeadamente, a *AWS Lambda* para além de apresentar *overheads* baixos apresenta uma grande consistência entre invocações, sendo também a plataforma que melhor lida com o aumento da carga concorrente, mantendo

uma escalabilidade linear. A *OpenWhisk* é a plataforma que apresenta um maior *overhead*, contudo consegue lidar bem com o aumento do peso computacional de uma função. Já a *Azure Funtions* apresenta tempos de *cold containers* bastantes altos se comparada com a *OpenWhisk*. Por fim, na plataforma da *Google* e na da *AWS* a memória alocada parece influenciar diretamente o poder computacional disponível para a execução da função.

No contexto do projeto *MobiWise* e como demonstração de uma das tarefas para o *workshop* intermédio foi efetuada a criação de um serviço de mobilidade, que otimiza rotas para um fluxo de veículos, recorrendo-se à utilização da plataforma *OpenWhisk* e a respetiva ferramenta de *workflows "Composer"*.

No geral, considero que todos os objetivos traçados para este estágio foram concretizados. Os testes de *benchmark* propostos permitiram verificar e comparar a performance das plataformas de computação *serverless* em várias vertentes, como era pretendido, e espera-se que o sistema de execução implementado possa ser usado por utilizadores deste tipo de plataformas, podendo ser expandido no futuro.

10.1 Trabalho futuro

Na continuação do trabalho realizado durante o estágio e descrito neste documento, penso que há espaço para algum trabalho futuro. Poderá ser criada uma interface gráfica, mais amigável para os utilizadores que a atual linha de comandos. Deve ser explorada a expansão do conjunto de testes de forma a incluir testes que permitam ter em consideração também a performance de outros sistemas que normalmente são usados em conjunto com as plataformas de computação *serverless*, como é o caso das ferramentas de *workflows*, de bases de dados, de sistemas de armazenamento de ficheiros, entre outros. Outro aspeto que não foi possível ser explorado, por questões de tempo, mas que pode ser bastante interessante é o consumo de energia por parte destas plataformas. Este tópico pode ser explorado especialmente em instalações locais através da comparação com outro tipo de sistemas de computação em *cloud*, como por exemplo os serviços tradicionais hospedados em máquinas virtuais.

Referências

[1] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.

[2] Claus Pahl. Containerization and the paas cloud. *IEEE Cloud Computing*, 2(3):24–31, 2015.

[3] Shashank Rastogi. Introduction to docker. Disponível em `https://dzone.com/articles/introduction-to-docker-1`, 2018. Acedido a 08-06-2018.

[4] Christopher M. Judd. Getting started with docker. Disponível em `https://dzone.com/refcardz/getting-started-with-docker-1`, 2018. Acedido a 09-06-2018.

[5] Arun Gupta and Christopher M. Judd. Getting started with kubernetes. Disponível em `https://dzone.com/refcardz/kubernetes-essentials`, 2018. Acedido a 09-06-2018.

[6] Steven Haines. Serverless computing with aws lambda, part 1 — javaworld. Disponível em `https://www.javaworld.com/article/3210726/application-development/serverless-computing-with-aws-lambda.html`, 2018. Acedido a 23-05-2018.

[7] Andrew Baird, George Huang, Chris Munns, and Orr Weinstein. Serverless architectures with aws lambda. Disponível em `https://d1.awsstatic.com/whitepapers/serverless-architectures-with-aws-lambda.pdf`, 2018. Acedido a 23-05-2018.

[8] IBM. Ibm cloud functions. Disponível em `https://console.bluemix.net/docs/openwhisk/index.html#getting-started`, 2018. Acedido a 05-06-2018.

[9] Rajdeep Dua, A Reddy Raja, and Dharmesh Kakadia. Virtualization vs containerization to support paas. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 610–614. IEEE, 2014.

[10] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[11] Pankaj Deep Kaur and Inderveer Chana. Unfolding the distributed computing paradigms. In *Advances in Computer Engineering (ACE), 2010 International Conference on*, pages 339–342. IEEE, 2010.

[12] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[13] Zaigham Mahmood and Richard Hill. *Cloud Computing for enterprise architectures*. Springer Science & Business Media, 2011.

[14] Preethi Kasireddy. A beginner-friendly introduction to containers, vms and docker. Disponível em `https://medium.freecodecamp.org/a-beginner-friendly-introduction-to-containers-vms-and-docker-79a9e3e119b`, 2016. Acedido a 09-06-2018.

[15] Docker Docs. Swarm mode key concepts. Disponível em `https://docs.docker.com/engine/swarm/key-concepts/`, 2018. Acedido a 11-06-2018.

[16] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.

[17] Garrett McGrath and Paul R Brenner. Serverless computing: Design, implementation, and performance. In *Distributed Computing Systems Workshops (ICDCSW), 2017 IEEE 37th International Conference on*, pages 405–410. IEEE, 2017.

[18] Garrett McGrath. *Serverless Computing: Applications, Implementation, and Performance*. PhD thesis, University Of Notre Dame, 2017.

[19] Theo Lynn, Pierangelo Rosati, Arnaud Lejeune, and Vincent Emeakaroha. A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 162–169. IEEE, 2017.

[20] Microsoft Azure. Azure functions documentation. Disponível em `https://docs.microsoft.com/azure/azure-functions/`, 2018. Acedido a 04-06-2018.

[21] Microsoft Azure. Microsoft azure functions. Disponível em `https://azure.microsoft.com/services/functions/`, 2018. Acedido a 04-06-2018.

[22] Google Cloud. Google cloud functions. Disponível em `https://cloud.google.com/functions/`, 2018. Acedido a 03-06-2018.

[23] IBM. Getting started with ibm cloud functions. Disponível em `https://console.bluemix.net/openwhisk/`, 2018. Acedido a 05-06-2018.

[24] Janakiram MSV. An architectural view of apache openwhisk. Disponível em `https://thenewstack.io/behind-scenes-apache-openwhisk-serverless-platform/`, 2017. Acedido a 06-06-2018.

[25] Markus Thömmes. Squeezing the milliseconds: How to make serverless platforms blazing fast! Disponível em `https://medium.com/openwhisk/squeezing-the-milliseconds-how-to-make-serverless-platforms-blazing-fast-aea0e9951bd0`, 2017. Acedido a 06-06-2018.

[26] IBM. Introducing serverless composition for ibm cloud functions. Disponível em `https://www.ibm.com/blogs/bluemix/2017/10/serverless-composition-ibm-cloud-functions/`, 2017. Acedido a 05-06-2018.

[27] IBM. Ibm cloud functions. Disponível em `https://www.ibm.com/cloud/functions`, 2018. Acedido a 05-06-2018.

[28] Openstack. Qinling's documentation. Disponível em `https://docs.openstack.org/qinling/latest/index.html`, 2018. Acedido a 25-05-2018.

[29] OpenFaaS. Openfaas - serverless functions made simple for docker & kubernetes. Disponível em `https://github.com/openfaas/faas`, 2017. Acedido a 01-06-2018.

[30] Alex Ellis. Openfaas. Disponível em `https://docs.openfaas.com/`, 2017. Acedido a 01-06-2018.

[31] Platform9. Fission: Serverless with kubernetes. Disponível em `https://platform9.com/fission/`, 2018. Acedido a 26-05-2018.

[32] Fission. Serverless functions for kubernetes. Disponível em `https://fission.io/`, 2018. Acedido a 26-05-2018.

[33] Fission. Fission: Serverless functions for kubernetes. Disponível em `https://docs.fission.io/0.7.2/`, 2018. Acedido a 26-05-2018.

[34] Matt Santamaria. Sd times github project of the week: Fission. Disponível em `https://sdtimes.com/github/sd-times-github-project-week-fission/`, 2018. Acedido a 26-05-2018.

[35] Ran Ribenzaft. How to make lambda faster: memory performance benchmark. Disponível em `https://medium.com/epsagon/how-to-make-lambda-faster-memory-performance-benchmark-be6ebc41f0fc`, 2018. Acedido a 15-06-2018.

[36] John Chapin. The occasional chaos of aws lambda runtime performance. Disponível em `https://blog.symphonia.io/the-occasional-chaos-of-aws-lambda-runtime-performance-880773620a7e`, 2017. Acedido a 16-06-2018.

[37] Nima Kaviani and Michael Maximilien. Cf serverless. Disponível em `http://schd.ws/hosted_files/cfsummiteu2016/87/CF%20Serverless.pdf`, 2016. Acedido a 16-06-2018.

[38] Simon Shillaker. A provider-friendly serverless framework for latency-critical applications. *12th Eurosys Doctoral Workshop*, 2018.

[39] Alex Casalboni. Google cloud functions vs. aws lambda: Fight for serverless cloud domination begins. Disponível em `https://cloudacademy.com/blog/google-cloud-functions-serverless/`, 2017. Acedido a 16-06-2018.

[40] Marco Parenzan. Microsoft azure functions vs. google cloud functions vs. aws lambda: fight for serverless cloud domination continues. Disponível em https://cloudacademy.com/blog/microsoft-azure-functions-vs-google-cloud-functions-fight-for-serverless-cloud-domination-continues/, 2017. Acedido a 16-06-2018.

[41] Yan Cui. Comparing aws lambda performance when using node.js, java, c# or python. Disponível em https://read.acloud.guru/comparing-aws-lambda-performance-when-using-node-js-java-c-or-python-281bef2c740f, 2017. Acedido a 16-06-2018.

[42] Tim Nolet. Aws lambda go vs. node.js performance benchmark: updated. Disponível em https://hackernoon.com/aws-lambda-go-vs-node-js-performance-benchmark-1c8898341982, 2018. Acedido a 17-06-2018.

[43] Matt Billock. Serverless performance shootout. Disponível em http://blog.backand.com/serverless-shootout/, 2017. Acedido a 17-06-2018.

[44] Serverless. The way cloud should be. Disponível em https://serverless.com/, 2018. Acedido a 18-06-2018.

[45] Maciej Malawski, Kamil Figiela, Adam Gajek, and Adam Zima. Benchmarking heterogeneous cloud functions. In *European Conference on Parallel Processing*, pages 415–426. Springer, 2017.

[46] AWS. Aws lambda limits. Disponível em https://docs.amazonaws.cn/en_us/lambda/latest/dg/limits.html, 2018. Acedido a 06-11-2018.

[47] Alessandro Ludovici, Pol Moreno, and Anna Calveras. Tinycoap: A novel constrained application protocol (coap) implementation for embedding restful web services in wireless sensor networks based on tinyos. *Journal of Sensor and Actuator Networks*, 2:288–315, 06 2013.