INSTITUTO
UNIVERSITÁRIO
DE LISBOA

Developing Self-adaptive Microservices

João Duarte Silva Figueira

Master in Computer Engineering

Supervisor:
Dr. Carlos Eduardo Dias Coutinho, Assistant Professor,
ISCTE – IUL

September, 2023

Department of Information Science and Technology

Developing Self-adaptive Microservices

João Duarte Silva Figueira

Master in Computer Engineering

Supervisor:
Dr. Carlos Eduardo Dias Coutinho, Assistant Professor,
ISCTE – IUL

September, 2023

To everyone that crossed my way and made me who I am,

My family, friends, teachers, and colleagues,

Thank you.

Acknowledgments

I would first like to thank my thesis' supervisor Professor Carlos Coutinho for his time and guidance through each stage of this dissertation. He gave me the confidence that I needed to close this chapter.

To my parents, Olga Silva e Miguel Figueira, for always being there for me since day one.

To the rest of my family, especially my grandparents, that kept pushing for me along this journey.

To my girlfriend, Liliana Moreira, for the encouragement and all the motivation that she gave me.

To Bruno Moscão, João Vieira and Eduardo Pinto, my superiors, for giving me the time and space to grow and invest in myself.

To all my friends who believed in me.

Thank you.

Resumo

As abordagens de desenvolvimento mais recentes estão a estabelecer os microsserviços e a computação em nuvem como tendências importantes para benefício da comunidade tecnológica. No entanto, estas tecnologias são frequentemente propensas a vários problemas relacionados com o desenvolvimento paralelo por várias partes, estratégias de entrega de software desenvolvido e afetação de recursos. Este artigo propõe uma nova arquitetura para o desenvolvimento de microsserviços autoadaptativos, utilizando Kubernetes através do Azure Container Apps, incluindo uma estratégia que complementará a arquitetura para melhorar o seu desenvolvimento, visando alcançar uma solução que permita aos leitores entregar software mais rapidamente, com mais resiliência, mais escalável e mais económico, dependendo o menos possível da intervenção humana para manter e escalar. O autor irá aplicar os conhecimentos adquiridos para propor e testar uma arquitetura para um caso de uso real, construindo um serviço de notificações integrado com um sistema complexo de aplicações web hospedado na nuvem.

Palavras-chave: Arquitetura de microsserviços; Computação em nuvem; Virtualização; Sistemas autoadaptativos; Kubernetes; Azure Container Apps;

Abstract

The modern development approaches are establishing microservices and cloud computing as major trends to benefit the technological community. However, these technologies are often prone to multiple issues regarding parallel development by numerous parties, delivery strategies and resource allocation. This paper proposes a novel architecture for developing self-adaptive microservices, using Kubernetes through the Azure Container Apps, including a strategy that will complement the architecture to enhance the development of microservices and aiming to achieve a solution that allows the readers to deliver software faster, with more resilience, more scalable, and more cost-effective, depending as low as possible from human intervention to maintain and scale. The author will apply the acquired knowledge to propose and test an architecture for a real use case scenario, building a notifications service integrated with a complex cloud-based web application system.

Keywords: Microservices architecture; Cloud computing; Virtualization; Self-adaptive systems; Kubernetes; Azure Container Apps;

Index

List of figures and equations

CHAPTER 1

Introduction

Developing and delivering software code to the real world can be challenging. Software applications are more complex than ever, and the users are extremely demanding. In 2018, the author of this thesis started his development career on a small startup, with the task of growing a cloud-based-platform used by multiple clients all over the world. As a junior developer, his main role was to deliver code, developing new features and maintaining the existing ones. Although this task was apparently simple, it didn't take long to apprehend the complexity that was involved.

The first big difference is the number of people working in the same code base. During their academic development, students are used to participate in group projects, where the group size is very limited, and most of the times they share the code sending it on a compressed file by email or through a cloud drive when part of the project is done. This doesn’t work on a real industrial project, because the number of persons involved is higher, and the tasks should be done on a parallel way, to deliver faster. This leads to problems like merge conflicts or dependencies between tasks from different developers. Of course, there are ways to solve these, but it’s complicated and time consuming. These problems were also boosted by another point: the company’s application was composed all in one piece. Meaning that all the functionalities and components were tightly coupled, making it harder to split work across multiple teams or individuals, which is a common pattern in older software.

The second divergence: multiple environments. Software companies that make use of a continuous delivery approach normally have multiple stages before dropping new code into production, to try to protect its customers from errors and unexpected behaviours. In this case, the code is first deployed on a development environment, that is only accessible by the developers and the quality assurance team, that performs the first tests. If the tests are accepted, the code is moved into a test environment, with different configuration and more stable than the previous one, and another round of tests is performed here. Finally, the code reaches production, distributed on different regions to improve latency across the world. But obviously, the workload on a production environment is completely different from the workload on a test or development environment. Even in production, there are differences between regions and the product performance is very dynamic throughout the day. When the software users are more active, the workload grows, and when they are sleeping, the workload is relative insignificant. Meaning that if we have the same power of resources all day long, we are not taking full advantage of it and we are overpaying.

The difficulty to manage many people working on the same code base is not the only problem faced when having the software composed all in one piece. When the team wants to deliver a new change into production, all the application needs to be deployed, which represents a high-risk operation. Very often the need to be sure that the version is stable enough, forces to delay the delivery and staying weeks without releasing into production.

These problems, found in the referred startup company, are common and applicable to the general case and represent a major short come to development activities that needs to be tackled. The company, following the global trend started splitting its applications into multiple microservices, taking advantage of all the benefits that it comes with. By having an important role in this change for the company, the author was pushed to the need to cope with the latest technologies and techniques on this topic and was able to research and develop in a real business a novel approach regarding the architecture for microservice development, namely for self-adaptive microservices.

1.1. Objectives

Taking in consideration the challenges that were elicited, the main objective of this research is to propose an innovative approach and architecture of a self-adaptive system, using the latest technologies and best practices known at the time of this writing, that allows developers to deliver code faster, with more resilience, which is more scalable and cost-effective, and depending as least as possible from human intervention to maintain and scale.
To exemplify and demonstrate these techniques, the author implemented a notifications service, which was developed for real production use in a real industrial scenario.

1.2. Research Questions

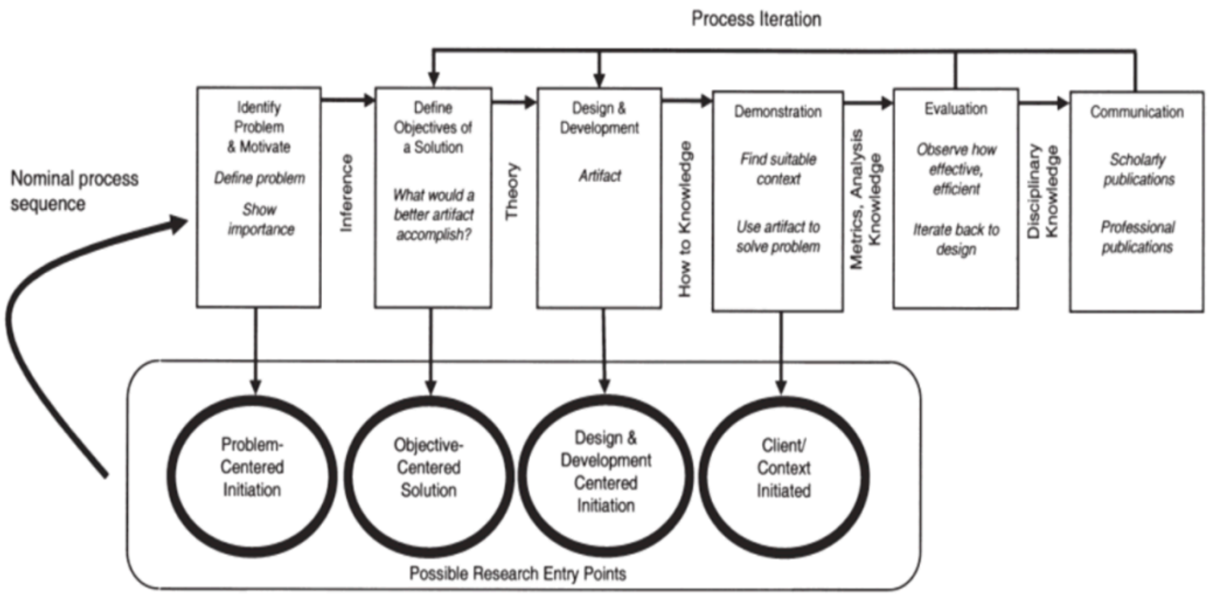
The methodology that was selected for this research (see section 1.3) advocates that the chased objectives should be tied to a set of research questions. In this case, a set of research questions for this research can be proposed:

- How can a software system benefit from the inclusion of a microservices architecture combined with cloud and virtualization techniques such as Virtual Machines and Containers, in its development?
- How can a system use self-adaptive techniques to adapt and optimize itself on different workload conditions to be cost effective and keep the desired performance?
- How can we abstract the microservices from the underlying infrastructure and deal with the challenges of a distributed system running on the cloud?

1.3. Research Methodology

The methodology used in this research was the Design Science Research (DSR). The DSR methodology aims to achieve "a new purposeful artefact to address a generalised type of problem and evaluates its utility for solving problems of that type" [1]. The main goal is to generate knowledge about relevant problems in the real-world environment and propose a way of how things should be done to achieve a desired set of goals [2]. In [2] the authors proposed 6 different sequential activities to follow when applying this type of methodology.

The steps that concern this methodology are visually represented in Figure 1, that represents the Design Science Research process model proposed by Jan vom Brocke, Alan Hevner and Alexander Maedche in [2].



Process Iteration
Nominal process sequence
Identify Problem & Motivate
Define problem
Show importance
Inference
Define Objectives of a Solution
What would a better artifact accomplish?
Theory
Design & Development
Artifact
How to Knowledge
Demonstration
Find suitable context
Use artifact to solve problem
Metrics, Analysis Knowledge
Evaluation
Observe how effective, efficient
Iterate back to design
Disciplinary Knowledge
Communication
Scholarly publications
Professional publications
Problem-Centered Initiation
Objective-Centered Solution
Design & Development Centered Initiation
Client/ Context Initiated
Possible Research Entry Points

Figure 1 - Design Science Research process model [2]

In the first phase of this methodology the objective is to identify the problem and the motivations that justifies the value and relevance of the solution. In this research, this step was accomplished on chapter 1, by enumerating the problems that motivated the research.

The second phase aims to define the objectives of a solution. After identifying the problem on the previous step, it's important to understand what a better artefact should accomplish. These objectives can be quantitative or qualitative and can be inferred from the phase one. In this case, the objectives were clearly defined in chapter 1 on the "Objectives" section.

In the third phase, the objective is to design and create an artifact. This includes the determination of the desired functionalities and its architecture. This step was accomplished on chapter 3, after using chapter 2 to understand the state of the art.

On the fourth phase the researcher should demonstrate that his produced artifact solves one or more instances of the problem defined on the first phase, finding a suitable context, and using his solution. This was done by simulating a real-world environment, in the section “Load tests and metrics analysis” of chapter 3.

The fifth phase is the evaluation. Here the research should observe how well the artifact can be used as a solution to the problem. At this point, if the researcher is not satisfied with the effectiveness of the artifact, he can return to the phase two or three to improve the outcome. This evaluation took place on chapter 4 - Results Discussion.

The sixth and last phase of the process is the communication, where the problem and the designed artifact is communicated to the relevant stakeholders. This has been accomplished by the publication of an article in the International Conference on Industry 4.0 and Smart Manufacturing (ISM 2023[1]).

1.4. Structure

This thesis is structured in the following way: The current chapter (1) introduces this research, presenting the main motivations, objectives, and research questions, as well as the methodology that conducted this research. Chapter 2 performs a literature review, analysing the most common software architectures to better understand the advantages and disadvantages of using Microservices, as well as understand complex concepts such as self-adaptive systems and cloud computing. Chapter 3 proposes an implementation of a self-adaptive microservice. It presents multiple diagrams that will help the reader to understand the architecture and implementation of the microservice and the integration with the overall system. Chapter 4 presents a discussion on the obtained results, mapping them to the research questions elicited. Finally, Chapter 5 presents the draw conclusions and presents future work.

[1] https://www.msc-les.org/ism2023

CHAPTER 2

Literature Review

This chapter elaborates a Literature review over the topics that are most relevant for this research, aligned with the methodology presented in 1.3.

2.1. Software Architectures

2.1.1. Overview

A software architecture refers to the way that an application is built, including the code and the way that each functionality or component interacts and connects with each other. The software architecture will influence the way that a software evolves, as well as the performance, scalability, and security. Each software has different requirements and concerns, and choosing the proper architecture is a key factor to the project's success, meaning that there's no "one-size-fits-all", and each case should be analysed and sometimes tested before commit to one.

2.1.2. Monolithic Architecture (MA)

The monolithic architecture (MA) has been for many years the main choice of the developers for modelling the design of software programs. The Cambridge Dictionary defines the word "monolithic" as "very large, united, and difficult to change", which is appropriate to understand this type of software design. In a MA, all the functionalities and business logic are encapsulated in a single application that runs in a single process and uses a single code base for all [3], [4]. Figure 2 shows a conceptual model of the monolithic architecture and illustrates the coupling between the User Interface, the business logic, and the data access layer.

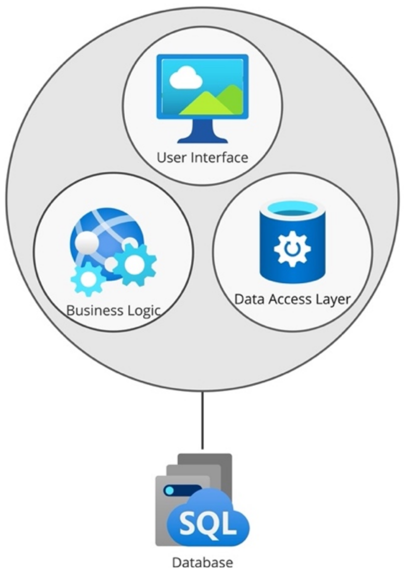


Figure 2 - Conceptual model of the monolithic architecture

2.1.3. Microservices Architecture (MSA)

Over the last years, the Microservices architecture (MSA) has increased importance to become the most popular way of developing applications [5]. Large companies like Netflix, Amazon, eBay, and many others have migrated their software systems from the old patterns (MA) to this architecture type [3].

In an MSA, the key idea involves isolating business functionalities into microservices that interact through standardized interfaces [6] based on lightweight mechanisms such as HTTP or gRPC. Each microservice runs on its own process and should be responsible for a single business logic with well-defined boundaries, normally managed and developed by a single team, hence it is independent from the other system components. Each microservice should be designed, developed, deployed, and administrated independently [7]. Figure 3 shows a conceptual model of a system composed for multiple independent microservices that communicate with each other and may or may not depend on a database component.

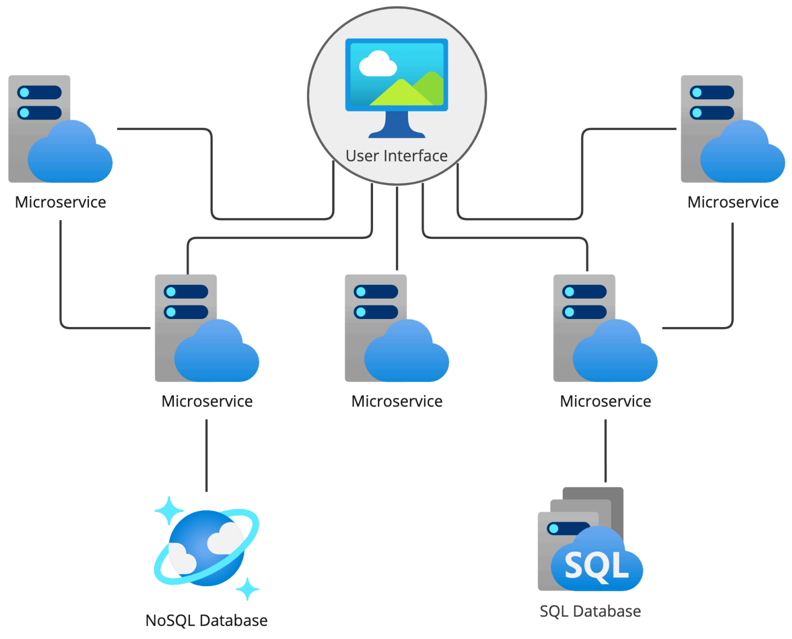


Figure 3 - Conceptual model of the microservices architecture

2.1.4. Monolithic vs Microservices

Each architecture has its advantages and disadvantages. The MA is simpler to design and has a better performance when we're talking about a small scale, since all internal communications is done via intra-process mechanisms [8]. Multiple studies have shown that monolithic systems normally outperform the microservices architecture on small and medium systems [3], [8], [9]. But along the time, a monolithic software tends to evolve and increase its size and number of functionalities, and it gradually becomes too large for any developer to fully understand. In addition, these functionalities may require different types of resources, making it difficult to choose the right server configuration [10]. There are some more advantages of choosing the monolithic as software design, such as the simplicity of testing and deploying. On the other hand, microservices are emerging due to multiple reasons. In [3], the authors enumerated five main advantages:

- **Technology heterogeneity -** each microservice can use its technology stack, accordingly to its needs and the characteristics that better suit him (e.g., in terms of performance), while in the monolith approach the larger the systems get, the harder it is to change [11].
- **Resilience -** having in mind the fact that each microservice is an independent component, the failure of one of them will not affect the whole system, since the other services can still handle new requests. The MA lacks fault isolation, because all the application runs on a single process, and an unexpected error would crash the entire software [10].
- **Scaling -** while in the MA scalability is difficult to achieve since it requires to be scaled as a whole, the MSA offers the possibility to only scale the components that need more resources.
- **Ease of deployment -** in a monolith application, any change requires the whole application to be deployed, and this could represent a high-impact and a high-risk for the organizations. Microservices are deployed independently, allowing us to get our change in production faster and safer, since the rollback can be achieved more easily if anything goes wrong.
- **Organizational Alignment -** microservices minimizes the number of people working on the same code base. This allows a better organization inside the development team, leading to more productive environments.

In [12], Sam Newman also brings to the table two more key advantages of using microservices. One of them is “Composability”, meaning that some functionality should be easily reused in different ways and perhaps for different consumers (e.g., Web and Mobile). The other one is mentioned as “Optimizing for Replaceability” and relies on the fact that microservices facilitates the task of replacing, refactoring, and even remove old system parts. On monoliths, this task is risky, and the effort is sometimes too high. However, designing a microservice architecture can be really challenging since there isn’t a well-defined algorithm for decomposing a system into multiple pieces. MA also increases the complexity of developing the software, introducing challenges such as distributed systems [10]. Beside of that, microservices also increase the complexity of the service governance, including monitoring, testing, discovery, and others [7].

2.2. Cloud Computing

2.2.1. Overview

Cloud computing can be seen as a business model. In the cloud it’s possible to store and access data and programs over the internet, instead of using our own local machines and hardware [13]. In fact, cloud computing clients can take advantage of the pay-per-use model, that is a payment model where you pay for what you use which fits the needs and allows to scale gradually, quickly, and as needed. A cloud client pays to use a small set of resources from a shared pool, and its billed considering its utilization. Cloud can be defined as a pool of virtualized and configurable computing resources, including hardware, storage, networks, interfaces, and services [13]–[15]. Its main characteristics include on demand self-service, scalability, and elasticity to increase or decrease usage demands by adding or removing resource power. The ability to “adapt to workload changes by provisioning and deprovisioning resources in an automatic manner” [16], avoiding under and over-provisioning with real-time monitoring. All of these are translated in a cost-effective system, more reliable and highly scalable.

2.2.2. Cloud Service Models

Cloud services can be provided in different model types and each one targets a different type of users and meets different requirements.

- **Infrastructure as a Service (IaaS) -** delivers the physical hardware and infrastructure that supports cloud computing, including servers, storage, network components, etc. The providers are responsible for all the maintenance and the physical place where the components are stored (data centres).

- **Platform as a Service (PaaS) –** an application platform that allows clients to build, deploy and manage software, without the complexity of building and maintaining the underlying infrastructure.
- **Software as a Service (SaaS) –** delivers software applications through the internet. Providers are responsible to host and maintain the software.

When the user requires more control and flexibility over the infrastructure, the IaaS is the most suitable model. However, it requires more management and maintenance concerns that the other types. In the other hand, SaaS offers the least control over the underlying infrastructure. Figure 4 visually represents this hierarchy.

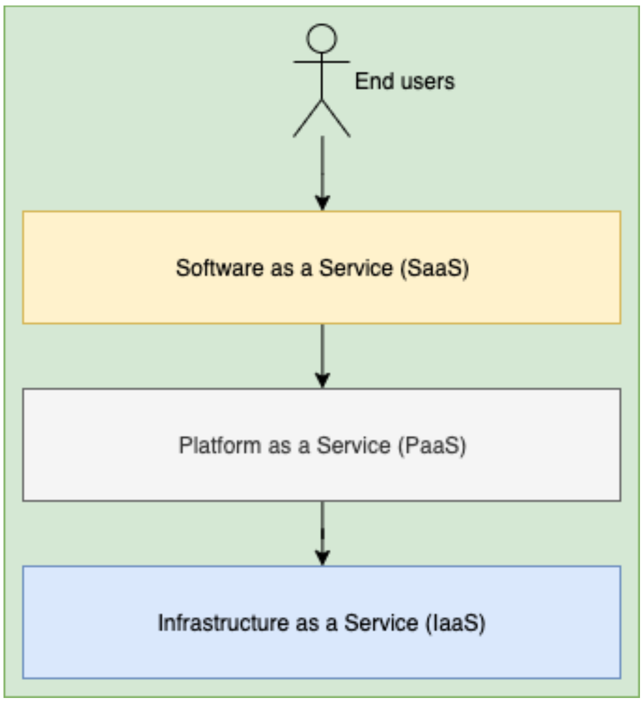

End users
Software as a Service (SaaS)
Platform as a Service (PaaS)
Infrastructure as a Service (IaaS)

Figure 4 - Cloud service models

2.2.3. Virtualization

It's the backbone technique of cloud computing [15]. Virtualization allows the system to run as an isolated system, but in a shared environment [17]. It creates an abstract layer of system resources, decoupling the software from the hardware. The three main characteristics of virtualization are [13], [14]:

- **Partitioning -** the available physical system resources are shared by portioning it in multiple parts so it can be used by various applications and operating systems.
- **Isolation -** each virtualized environment is independent from the host and other environments, meaning that they can't share data and a failure on one element won't affect the others.

- **Encapsulation -** stored or represented as a single file.

Virtualization can be applied on different levels: Operating System (OS), Application-Server, Application, Administrative, Network, Hardware, and Storage.

2.2.4. Virtual machines (VMs)

Virtual machines are one of the most famous types of virtualizations. On the traditional architecture, the software application runs on top of an operating system (OS) that operates the hardware. The next figure represents a diagram of a traditional host architecture, with 3 different layers (as shown on Figure 5):

- **Application layer –** the programs that are executed on the OS (e.g., Microsoft Word)
- **Operating System Layer –** the OS installed on the host (e.g., Windows, Linux…)
- **Infrastructure/Hardware Layer –** the physical resources from the host (e.g., Memory, Disk)

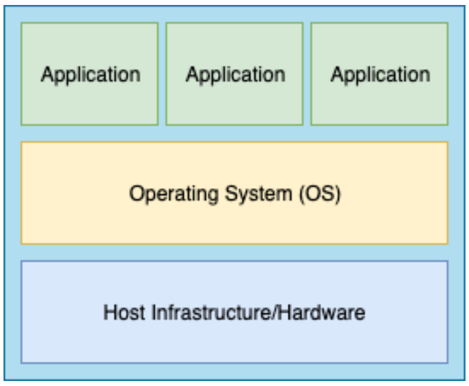


Figure 5 - Traditional host architecture

However, most of the times the programs and applications running on a host machine don't take advantage of all the physical resources available, using only a small amount of the available power. Instead of having multiple machines with unutilized resources, it's possible to take advantage of virtualization to distribute those physical resources for multiple virtual environments, using for example virtual machines.

A virtual machine has its own operating system and shares a pool of resources (from the "host" machine) with another virtual environments. All the virtual machines running on a host are managed by the Virtual Machine Manager (VMM) also known as Hypervisor, which is a "program or combination of software, hardware or firmware that creates and executes various virtual machines" [15].

The hypervisor can run directly on the host machine, and these are called native hypervisors or bare metal hypervisors (Figure 6). This type (Type I) is generally more efficiently and secure, considering that it works without the attack-prone operating system layer.

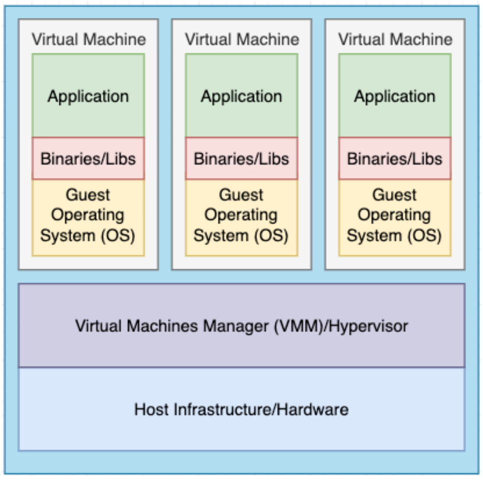

Virtual Machine
Application
Binaries/Libs
Guest Operating System (OS)
Virtual Machine
Application
Binaries/Libs
Guest Operating System (OS)
Virtual Machine
Application
Binaries/Libs
Guest Operating System (OS)
Virtual Machines Manager (VMM)/Hypervisor
Host Infrastructure/Hardware

Figure 6 – Virtual Machine architecture running on a host machine (Type I / Bare-metal / Native Hypervisor)

On the other hand, the hypervisor can also run on top of the host machine operating system, just like other programs do. However, the guest operating systems (from the VMs) are abstracted (isolated) from the host OS, and for this reason they can be different. This type of hypervisor is called Embedded or Hosted Hypervisor. Having this extra layer, the latency is bigger when compared to the Type I (Figure 7).

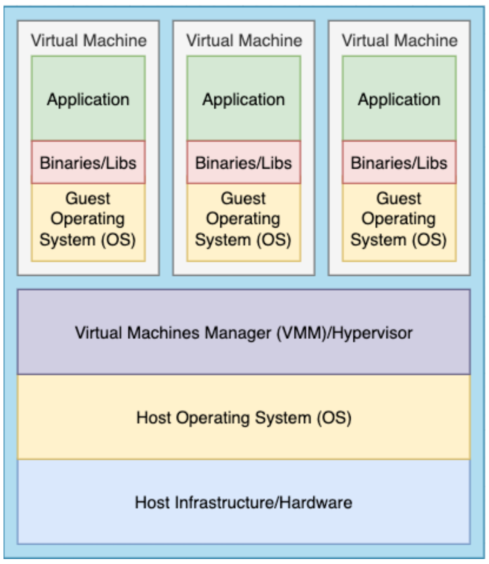

Virtual Machine
Application
Binaries/Libs
Guest Operating System (OS)
Virtual Machine
Application
Binaries/Libs
Guest Operating System (OS)
Virtual Machine
Application
Binaries/Libs
Guest Operating System (OS)
Virtual Machines Manager (VMM)/Hypervisor
Host Operating System (OS)
Host Infrastructure/Hardware

Figure 7 - Virtual Machine architecture running on a host machine (Type II / Embedded / Hosted Hypervisor)

The use of virtual machines has many benefits, such as:

- **Cost effectiveness –** it lets you share the same infrastructure to run multiple virtual environments, reducing maintenance and electricity costs, contributing to a green computing practice.

- **Portability –** A virtual machine is represented on a single file called "image". This file encapsulates all the VM layers, including the programs, binaries/libraries, and operating system. For this reason, it's very easy and fast to provide entire new environments, improving the availability and performance of applications.
- **Security –** Being a virtualized environment, it is isolated from the host OS, protecting the host from potential treats as viruses and attacks.

2.2.5. Containers

Containers are also a virtualization technology that allows multiple applications to run on a single host machine. It virtualizes the operating system and they are more lightweight because, unlike virtual machines, containers do not run their own operating system. Instead, the container image contains only the application and its dependencies (libraries). It runs on top of a container engine, like Docker, which allows its users to create, maintain and deploy containers, providing an API to communicate with its core engine - Figure 8.

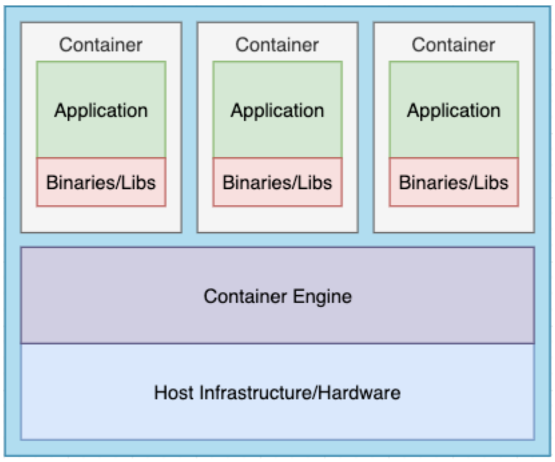


Figure 8 - Container architecture

2.2.6. Docker

Docker is one of the most well-known and widely used platforms for developing, shipping, and running container applications with fastest and more consistent deliveries [18]. The docker architecture is built around three main components: the docker client, the docker host and the docker registry. The docker client allows the user interaction with the Docker daemon, through the Docker API. The Docker daemon is the core of the Docker platform, responsible for building, running, and distributing the docker containers, as well as manage other docker objects such as images, networks, and volumes. The registry is where the docker images are stored and it can be public, like the Docker Hub (used by default), or private. When the user makes a request to push, pull or run a container image, the docker daemon uses the configured repository to upload or download the requested image. Figure 9 is a simple diagram of the Docker architecture that shows some of the interactions between the Docker Client, the Docker Host (where the Docker Daemon is running) and the different registries.

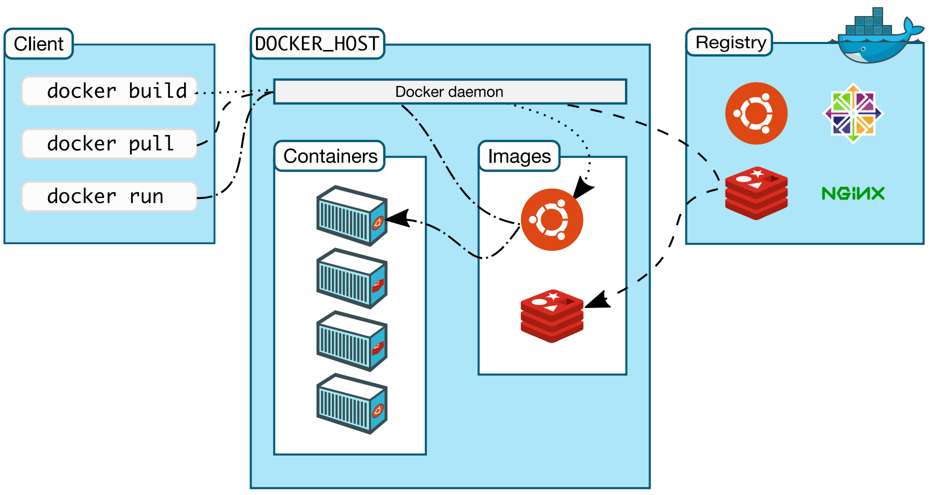
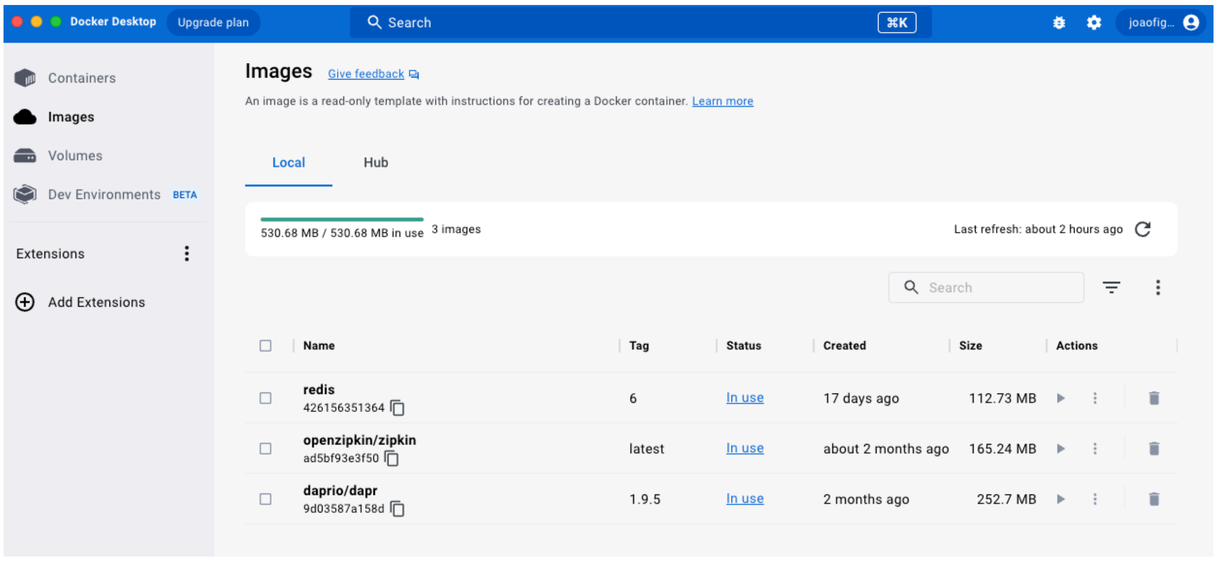


Figure 9 – Docker architecture [18]

The Docker containers are created from the Docker images pulled from the repository. Users can build up containers from public images from public repositories, or create their own using a "Dockerfile", that contains simple instructions on how to build it. These instructions can be used to specify the base image to use as the starting point for the build (like an operating system or a framework), the files to be copied from the host, the working directory for subsequent instructions, commands to be executed, environment variables, the exposed ports that the applications listen on, etc. The Figure 10 and Figure 11 are snapshots from the Docker Desktop, an integrated development environment (IDE) application for Windows, Linux and macOS, which allows to manage images, containers, and applications through a Graphical User Interface (GUI). It shows three images locally stored, that were previously downloaded from the Docker Hub. Each of these images was used to launch a container that is running and reachable on a specific port.



Images

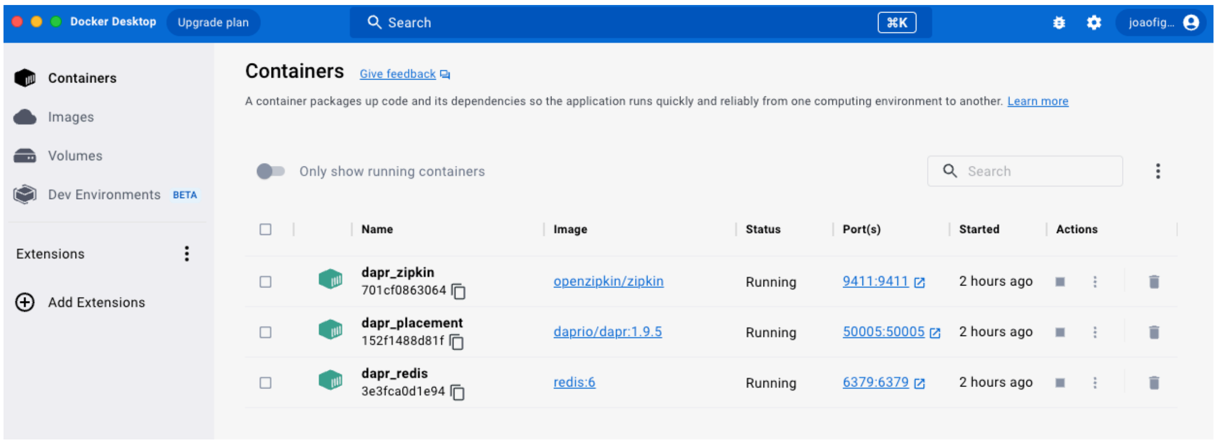
An image is a read-only template with instructions for creating a Docker container. Learn more

530.68 MB / 530.68 MB in use 3 images

Last refresh: about 2 hours ago

Name	Tag	Status	Created	Size
redis 426156351364	6	In use	17 days ago	112.73 MB
openzipkin/zipkin ad5bf93e3f50	latest	In use	about 2 months ago	165.24 MB
daprio/dapr 9d03587a158d	1.9.5	In use	2 months ago	252.7 MB

Figure 10 – Docker images managed on the Docker Desktop (v4.16.2 on macOS)



Containers

A container packages up code and its dependencies so the application runs quickly and reliably from one computing environment to another. Learn more

Name	Image	Status	Port(s)	Started
dapr_zipkin 701cf0863064	openzipkin/zipkin	Running	9411:9411	2 hours ago
dapr_placement 152f1488d81f	daprio/dapr:1.9.5	Running	50005:50005	2 hours ago
dapr_redis 3e3fca0d1e94	redis:6	Running	6379:6379	2 hours ago

Figure 11 – Docker containers running on the docker platform and managed on the Docker Desktop (v4.16.2 on macOS)

2.2.7. Virtual machines vs. Containers

Both technologies allow their users to take advantage of the virtualization benefits. However, each one has its own advantages and more suitable tasks. Virtual machines use hardware virtualization, meaning that it is fully independent from the other virtual machines and host. On the other hand, a container uses operating system-level virtualization, sharing the host's kernel and even some libraries. Having this in mind, we can consider that VMs are more isolated than containers, however it comes with the cost of requiring more resource power to run this full virtual copy of a normal host machine. Meaning that a virtual machine will normally use more memory, storage, and CPU than a container. In [19], the authors performed some tests to compare the performance between docker containers and virtual machines, considering many aspects such as CPU performance, memory throughput, storage read and writes measurements, load tests and operations speed. They observed that docker containers performed much better over virtual machines in every test. For example, in the load test the authors used an Apache Benchmarking tool to evaluate the throughput of each one, and they concluded that a docker container can handle more than the double of the requests per second, as shown on the Figure 12 (retrieved from the paper).

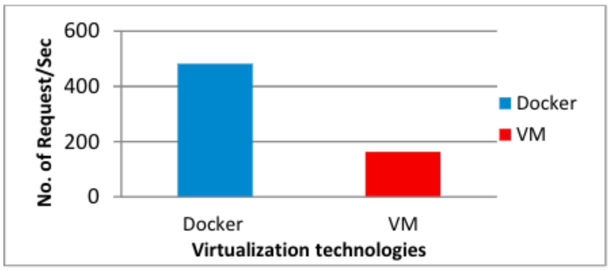


Figure 12 - Load tests to evaluate the No. of requests per second processed by each technology. Source: [19]

In terms of CPU and memory levels, the tests were also very conclusive. The memory tests were executed using different operations to measure multiple aspects, and they have shown that it is more than twice as fast when using docker in every type of operation. These memory speed results are shown on Figure 13.

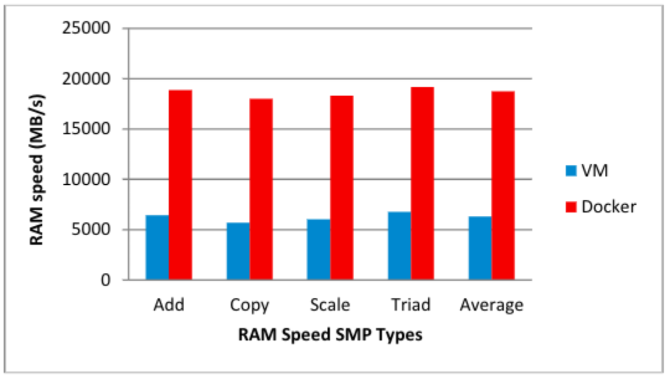


Figure 13 - Memory speed tests results. Source: [19]

The container's CPU has taken approximately half of the time to resolve different computational problems. The following figure shows the completion times that each virtualization technology took to resolve the classic problem in computer science and mathematics: "the eight queens". It's a chess puzzle, where the objective is to place eight queens on a chess board, in a way that no two queens can attack each other. In other words, no two queens can be on the same row, column, or diagonal. One more time, the container's benchmark was much better, because the execution time was much lower, as shown in Figure 14.

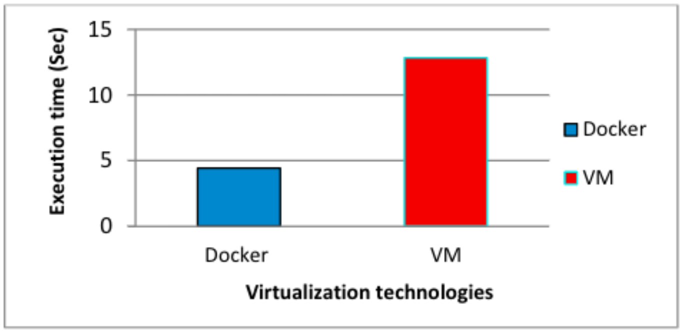


Figure 14 - Eight Queen problem resolution. Source: [19]

Summing up, containers are more effective on the resource usage, which normally helps to improve performance and reduce costs even more than virtual machines. Considering that containers do not run their own operation system, they're also highly portable and the start-up time is much faster when compared to VMs, making it well-suited to scale applications.

2.3. Self-adaptive microservices

2.3.1. Overview

The scale of the system or system faults are some of the uncertainties that are not possible to anticipate before deployment. The only viable architecture management solution was proposed by Kephart and Chess: self-management [20]. Self-adaptive microservices should be able to autonomously adapt themselves to the current unpredictable circumstances to reach certain high-level goals defined by the designers and administrators. In [21] the authors described self-adaptive systems as systems that are constantly monitoring their behaviour to modify themselves at run time to preserve or enhance their quality attributes.

The Figure 15 is a conceptual model of a self-adaptive system presented by Danny Weyns [22]. The main actors are the "Managed System", "Managing System" "Environment" and "Adaptation Goals".

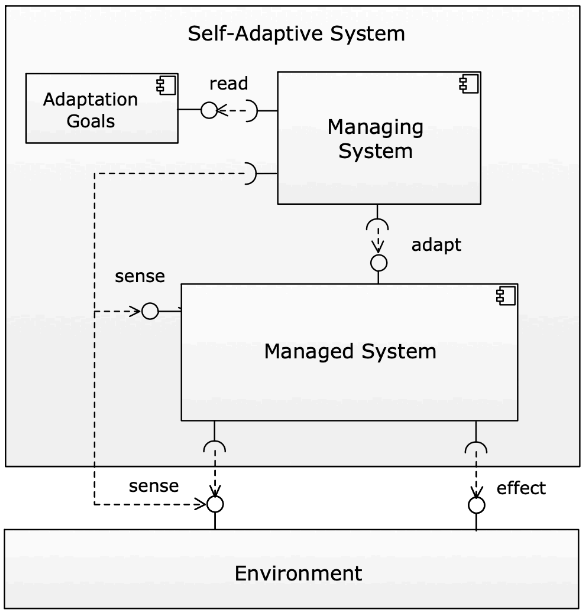


Figure 15 - Conceptual model of a self-adaptive system. Source: [22]

The managing system manages the managed element accordingly to the effects produced by the environment and felt on the managed system, to reach a group of adaptation goals [22].

To make runtime changes on the system, the managing element usually has a control loop that consists in four different stages: Monitor, Analyse, Plan, and Execute. These elements share common Knowledge, and this is called MAPE-K [21], [22], represented on Figure 16. This loop monitors the managed system and gathers information on the environment (where the conditions are dynamically changing), analyses it to identify possible problems and divergencies from the desired goals, plans on how to act if necessary, and executes that plan [11], [22].

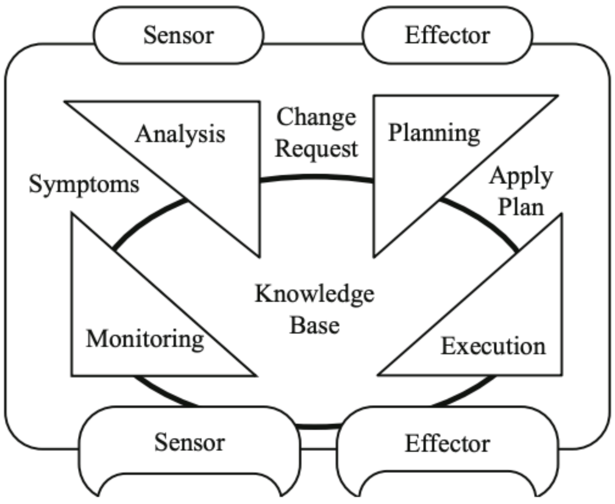


Figure 16 - The MAPE-K loop [20]

2.3.2. Self-management types

There are four types of self-management objectives [11], [22], [23]:

- **Self-configuration -** systems can configure and readjust themselves to meet the agreed objectives.
- **Self-optimization -** continuously monitor themselves and seek opportunities to improve performance and costs.
- **Self-healing -** ability to autonomously recover from failures and even predict them.
- **Self-protecting -** against malicious attacks or cascading failures.

2.3.3. Achieving self-adaptation: The Rainbow framework

In [17], [24], the authors developed a framework that uses externalized control mechanisms. They called it the "Rainbow Framework" and the main objective is to achieve self-adaptation, monitoring, and adapting at run time, using an external "managing system". This principle facilitates the task of reuse self-adaptive mechanisms across systems and its components and makes it possible to apply it to legacy systems with a low-cost approach.

The framework's infrastructure is divided in three layers [24], as shown in Figure 17:

- **System-layer infrastructure –** is where the executing system is allocated, in other words, is the target of self-adaptation (managed system), the component that will be affected by the environment and changed at runtime. It must be equipped with a measurement mechanism – *probes*, to observe and monitor the system's state – and *effector* mechanism to carry out the modification if needed.
- **Architecture-layer infrastructure –** this is the layer responsible for receiving the information sent by the probes and constantly evaluate it to find possible constraint violations (*Constraint evaluator*).
- **Translation-layer infrastructure –** mediates the mapping between the system and the architectural layers, keeping the abstraction between them to reuse as many components as possible.

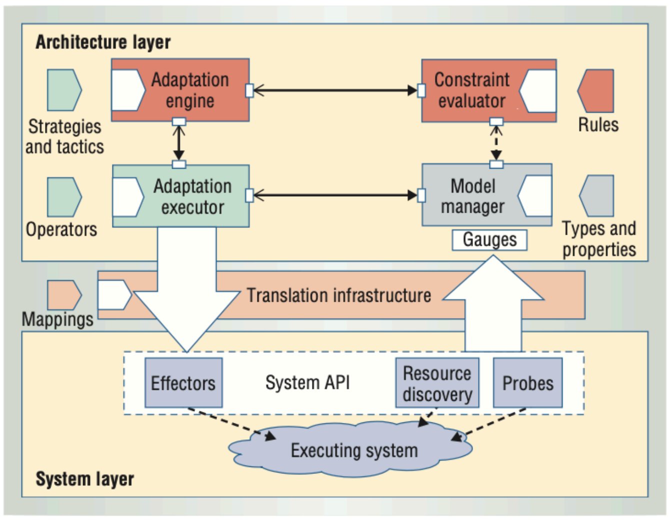


Figure 17 - Rainbow framework [24]

2.3.4. Kubernetes and container orchestration

Kubernetes is an open-source project. It’s an orchestration system for automating the deployment, scaling, and management of containerized applications. This automated orchestration allows to achieve high levels of availability. It is capable of detect unhealthy containers, restarting or replacing them with new ones and detect host failures [25]. Kubernetes also provides load balance and scaling capabilities, to self-adapt on different workload conditions.

A Kubernetes cluster is composed by two different types of nodes: the master node, and the worker nodes. Nodes are physical or virtual machines, with the necessary services to run Pods, “the smallest deployable unit in Kubernetes” [26] where the application container is running. Each Pod can be seen as an execution environment that can run multiple containers and share the same resources, such as the network stack, memory, and storage. All containers in a Pod run on the same node and are immutable – updating a Pod configuration requires replacing it with a new one [27].

The master node is the control plane of the cluster and where multiple necessary Kubernetes processes are running. It's responsible to monitor, scale and replace pods and nodes, to match the defined deployment configuration. The control plane is composed by an API Server, the "kube-apiserver", responsible for validating and forwarding of all the requests to the appropriate component. It allows the user to query and manipulate the state of Kubernetes objects, such as Pods. For example, if a user makes a request to create a new pod, he would use this API, which would forward that request to the "kube-controller-manager". The controller manager is another component built-in the control plane, that runs multiple processes, each one representing a different controller to perform different tasks and jobs. These background processes are constantly running and trying to adjust the cluster to match as far as possible the desired state configuration (e.g., the minimum number of pods). After creating a new Pod, the scheduler component would be noticed and responsible to assign it to a node, considering many factors such as the hardware requirements or software and policy constraints.

The API server also communicates directly with the "etcd", another component that represents a distributed key-value store used to store the cluster's configuration, as well as the current state.

On the other hand, the worker nodes are composed by the "kubelet" and the "k-proxy". The "kubelet" is responsible for monitoring the containers within the pods and ensuring they are running and healthy. It communicates with the control plane and receives instructions on what actions to take, while also reporting the overall status of the node. The Kube Proxy is responsible for providing network access to the pods within and from outside the cluster and acts as a load balancer [26], [27]. Figure 18 shows the Kubernetes cluster architecture with all the components previously exposed.

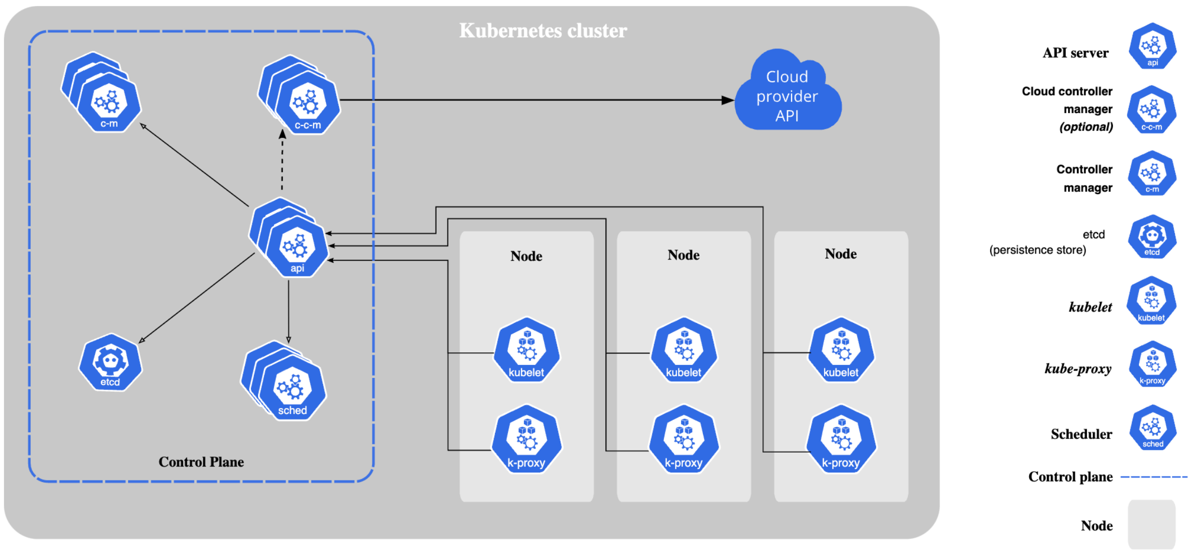


Figure 18 - Kubernetes cluster architecture. Source: [26]

However, in [25] the authors preformed some availability tests on Kubernetes services and concluded that its healing system by itself has some limitations when specific components start failing, especially the master node, which makes total sense considering that this node is responsible for maintaining all the cluster. To guarantee high Service Level Agreements (SLA), the Kubernetes' healing capabilities should be reinforced with redundancy models, including configure the deployment configuration to have multiple replicas of the master node.

2.3.5. Dapr (Distributed Application Runtime)

Dapr (Distributed Application Runtime) simplifies the development, deployment, and management of microservices and cloud-native apps with its open-source, event-driven portable runtime. It provides a set of building blocks and eliminates (or at least reduces) the need for managing infrastructure, allowing developers to create scalable and portable distributed apps that can run in multiple environments. With a simple and consistent programming model, Dapr enables developers to concentrate on writing business logic rather than dealing with infrastructure. It can be used with any developing language or any framework and run on any cloud or edge infrastructure. The building blocks APIs allows its users' code to remain simple, portable, and agnostic to any specific infrastructure implementation, encapsulating the industry's best practices [28].

At the time of this writing, Dapr offers 9 different types of building blocks [28]:

- **Service-to-service invocation –** to eliminate the need of a service discovery to find out the other remote services locations, by simply call it by its unique name and communicating through encrypted TLS connection.
- **State management –** pluggable component for storing and querying key/value pairs in a language-agnostic way and regardless of the underlying data store, including SQL databases, NoSQL databases or key-value store. It also provides the ability to manage the concurrency control and data consistency settings. This component has integrations with (but not only) Azure SQL Database, Azure CosmosDB, AWS DynamoDB, MongoDB, Redis, among many others.
- **Publish and subscribe –** to send and receive messages using topics and queues regardless of the underlying message broker. It integrates with different message brokers and queuing systems, such as Azure Service Bus, RabbitMQ, AWS SNS/SQS, and others. This type of component is used on event-driven architectures, and it enables microservices communication with each other by sending (publisher) and subscribing (subscriber) messages on a message broker.

- **Resource bindings –** a component to be used on event-driven architectures to send and receive events from external resources. The incoming events can be listened on a HTTP endpoint or a gRPC proto library configured on the component's YAML file and trigger code on the application when an event is received.
- **Actors –** to implement the actor pattern, where actors are encapsulated units of code, that can receive and send messages to each other asynchronously. Each actor has its own private state, and each instance can only process one message at a time.
- **Observability –** a built-in component to provide insight information that is not possible to collect on the infrastructure layer about the behaviour and performance of a distributed application. It allows to gather information such as traces, logs, or metrics, and be easily integrated with monitoring tools such as Zipkin, Jaeger or Datadog.
- **Secrets –** to store sensitive application information like connections string, keys, or tokens, on dedicated secret stores such as AWS Secrets Manager or Azure Key Vault. The main objective of this pluggable building block is like the ones referenced previously for other building blocks: being agnostic to the underlying infrastructure.
- **Configuration –** the configuration API allows to access application configuration items from state stores and databases returned as a read-only key/value pairs. An application can subscribe those items, to be notified of updates and load the new configuration at runtime.
- **Distributed lock –** a distributed lock across all instances of an application, used to provide exclusive access to shared resources. It's usually used on update operations (not reads), when only one instance should update a state at the same time, to avoid concurrency and consistency problems.

2.3.6. Dapr and Kubernetes on a Cloud Platform

On Kubernetes, the containers are deployed inside of pods, and each pod can have multiple containers. When we deploy an application or a service with Dapr enabled, it deploys a secondary container on the same pod, called "sidecar". This sidecar container is used to communicate directly with other sidecars (from another services) and with the Dapr components. The application container and the sidecar can communicate with each other by HTTP and gRPC. For example, if an application wants to save a state using the building block that Dapr provides, it would make a HTTP or gRPC call to the side car, which would forward that the request to the building block API.

In Figure 19, we can have a look on the Dapr architecture on Kubernetes. When Dapr initializes, it starts by deploying four main services, each running on its own pod: the Dapr sidecar injector, the Dapr operator, the Dapr placement and the Dapr Sentry. The Dapr sidecar injector is the service responsible for watching for new pods on the cluster, checking if it contains the required Dapr annotations and inject the sidecar when needed. The Dapr Sentry is the certificate authority that manages the encrypted communication between services, using the mutual authentication TLS (mTLS) mechanism to build an encrypted channel after two-way authentication is established. The Dapr Placement stores the different locations of each actor's instance, so it can be reached when required.

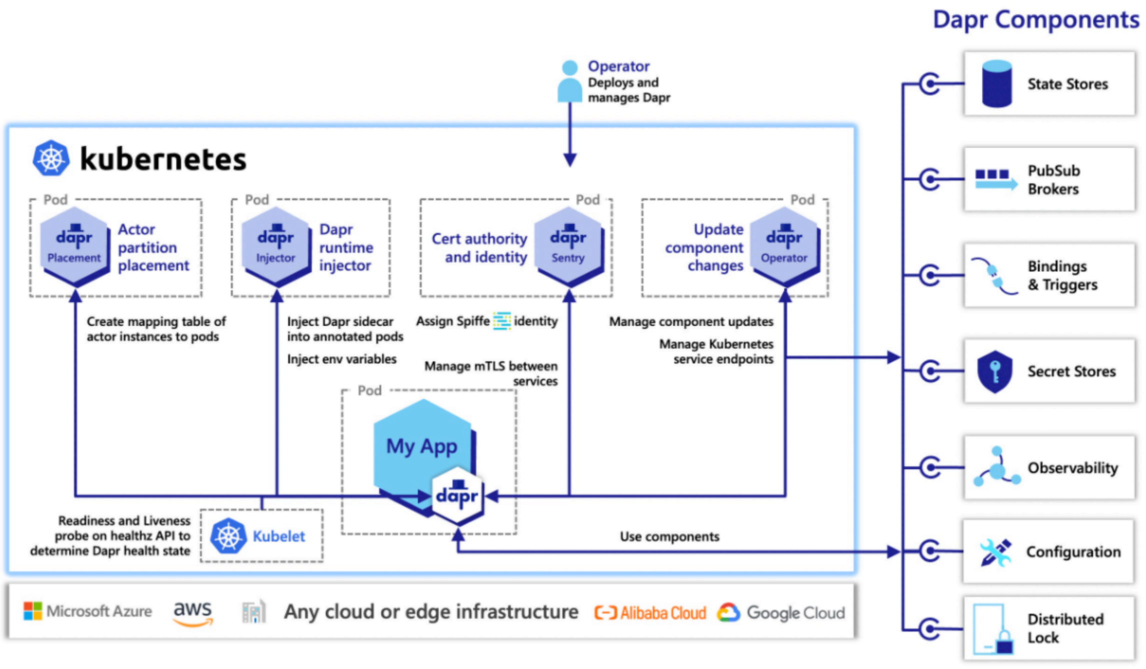


Figure 19 – Dapr architecture when running on a Kubernetes hosting environment. Source: [28]

2.3.7. KEDA

KEDA (Kubernetes Event-Driven Autoscaling) allows the Kubernetes cluster's containers scaling, based on event triggers and states. It scales horizontally (adding or removing pods), to and from zero (to be as much cost-efficient as possible) in conjunction with the Kubernetes Horizontal Pod Autoscaler (HPA) [29]. HPA scales based on memory and CPU metrics, but sometimes those are not the best metrics to base the scaling option. For example, a service that consumes messages from a queue may be very memory and CPU efficient and Kubernetes will assume that a single replica (with only one pod) is enough based on those metrics. However, if the message producer produces faster that the consumer consumes, the time that the messages spend waiting for processing will start to grow, and that can be a problem on some latency critical scenarios.

KEDA has a rich catalogue of scalers that can detect if the Kubernetes deployment should be activated or deactivated, including not only memory and CPU metrics, but also Cron schedules, SQL query results, queue lengths, topic subscriptions, custom metrics and many others specific event sources that can be found on the official website [29]. Figure 20 shows how KEDA works in conjunction with Kubernetes, the KEDA extends the HPA capabilities. While the HPA is responsible for scaling the cluster from 1 to N, and N to 1, the KEDA is responsible for scaling it from 1 to 0, and 0 to 1.

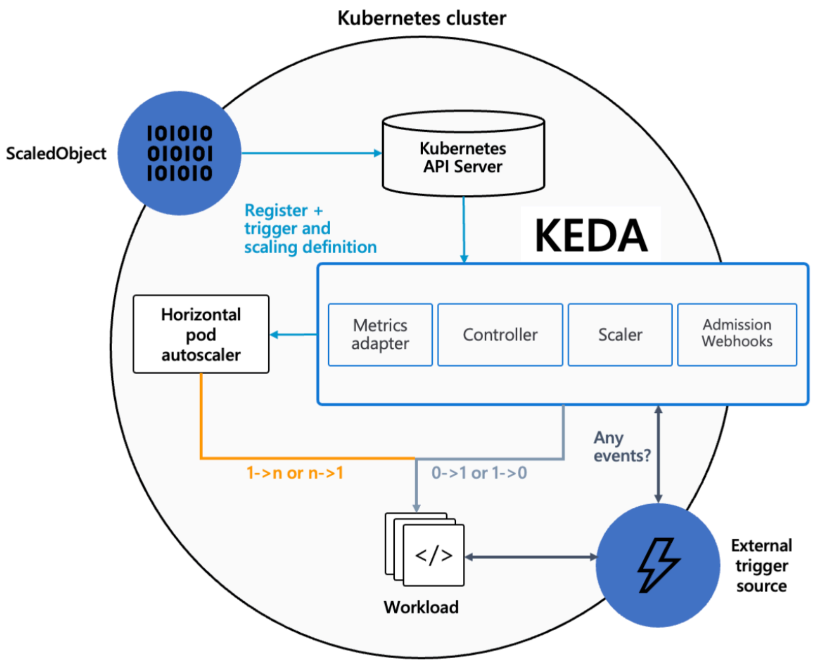


Figure 20 – KEDA works in conjunction with the Kubernetes [29]

2.4. Cloud providers and container services

2.4.1. Overview

A cloud provider is a company that provides computing services and infrastructure over the internet, including storage, servers, databases, networking, software, and others.

2.4.2. Microsoft Azure as cloud provider

At the time of this writing the 2 most popular cloud service providers (CSP) are the Microsoft Azure and the Amazon Web Services (AWS) [30]. As the Figure 21 shows, these 2 cloud providers combined are holding 55% of the market. Each CSP provides different services and platforms. As previously said, the author will implement a solution for a real problem, that will be used in a production scenario. The system where this solution will be integrated is already hosted on the Azure cloud. For this reason, to have a more natural integration, this solution will also be deployed and implemented using the Microsoft technologies. Azure provides multiple ways of running a containerized application on the cloud. Azure Container Apps (ACA) is one of those services and provides multiple advantages when compared with the other type of services available. These advantages will be explored on the Chapter 3 (section 3.5).

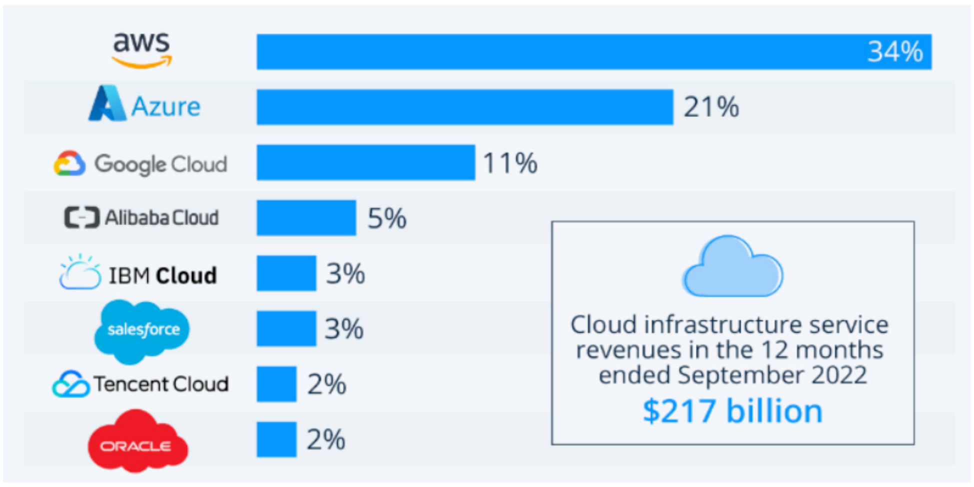


Figure 21 - Cloud market share company wise [30]

2.4.3. Other options from different cloud providers

In some cases, the different cloud providers have a direct one-to-one equivalent service. For example, the Azure Kubernetes Services (AKS) and the Amazon Elastic Kubernetes Service (EKS) are very similar services that allows to orchestrate Docker containerized application deployments with Kubernetes. In this match, the services have a close feature-to-feature parity. But in other cases, a service on a specific cloud provider may not have an equivalent service on another cloud provider. This is the case of the Azure Container Apps. On AWS there's no service that provides such abstraction over Kubernetes, neither the built-in integrations with Dapr and KEDA. To use a different cloud provider and implement this solution it would require more knowledge about container orchestration, and it would introduce the complexity of manually inject the Dapr framework in the cluster.

CHAPTER 3

Proposed solution

In this chapter the author proposes an implementation framework to be used to develop self-adaptive microservices. For a better understanding and for testing scenarios, the author chose to implement a real use case and develop a notifications service. The artifact produced in this chapter will be used by a real company that has a cloud-based-platform hosted on the Azure Cloud.

3.1. Notifications overview

The main core goal of this service is to deliver custom notifications to end-users. Notifications may have different types, and each type will probably have a different level of importance. For this reason, when the sender makes a request to create a new notification, the sender should be able to choose on what platforms the recipient should receive it (Web, Email, SMS, etc). In addition, each user should have total control to customize the type of notifications that he wants to receive, and to configure on what platforms he wants to receive them. These user preferences are important because most of the times the users are bothered with notifications that they have no interest on, which end up with the users ignoring most of them.

Usually, the notification types are well known by the client applications (such as web) and the template to present each type is on the source code. But one of the requirements of the system is to allow users to create their own notifications with custom types, based on triggers and business logics that they consider relevant. Therefore, it must be possibly to define the template that should be used to display each notification. Additionally, each channel has a different User Interface (UI) and needs to display the notification in a different way, so it might be a different template per channel. These templates are saved on another microservice (out of the scope of this thesis).

As previously said, this service is going to be used on a real production environment, and for that reason, it needs to address multiple functional and non-functional requirements.

Functional requirements:

- Create a new notification for a user or group of users.
- Update a notification to mark it as read.
- Get the unread notifications counter for a user since a specific datetime.
- Get the list of notifications for a specific user.
- Get the list of unread notifications for a specific user.
- Group notifications that are related (for each user).
- Support the sending for multiple platforms (Web Application, email, SMS, etc.)

- Each notification shall have a specific type, so each user can choose to receive it or not on the different platforms.
- It should be possible to easily create new notification types and show them on the several platforms (without code required).

Non-functional requirements:

- Multi-tenant support
- Decoupled from the main application.
- Highly scalable without human intervention.
- Easy extensible (easy to add integrations with new platforms).
- The microservice shall probe and try to heal itself.

3.2. System overview

The Figure 22 is the first diagram of the author's solution, and it shows the high-level architecture of a system composed by several microservices. Each micro-service can interact with another through endpoints exposed by each one. These APIs (Application Programming Interface) are private (or internal) and can only be reached by resources placed on the same Virtual Network – which means that the service does not have a static public IP address to be called over the public internet by external users or services. If any of these want to interact with an internal microservice, it must be done through the public API, which works as a gateway and knows how to discover the internal resources. Using a gateway as suggested on this diagram offers multiples benefits. It can be used as a protocol translator, allowing to use different communication protocols between different actors. In this case, the external subjects communicate with the public gateway through REST, and the public API communicates with the microservices through gRPC. This option relies on the fact that gRPC can be significantly faster than REST, with lower latency and higher throughput, since it uses binary serialization to exchange data between services, which is more memory efficient than other types of serialization, such as JSON and XML [31]. However, it still has low compatibility with the browsers making it more difficult to use, and for this reason, it makes sense to take advantage of both protocols, using the REST when exists interaction with users (through browsers) and the ease of use is a priority, and gRPC for internal communications, where the performance is a critical request. The gateway can also work as a security layer and include authentication and authorization mechanisms, as well as other features such as encryption and firewalls. In Figure 22 there are four microservices represented, but the Notifications Service is the focus of this project and is the one that the author will propose an architecture for, using basic implementations for the others. Beside of that microservice, there is also represented the API Gateway, that the author already explained its purpose, and the Users and Templates microservices.

These services are extremely simple, responsible for the CRUD operations for each entity and without any relevant logic beside of that. This diagram is a high-level view of the system and the interactions between them and the outside network. More ahead the author will introduce new concepts and technologies that will allow him to present a more complex and detailed view of the final solution.

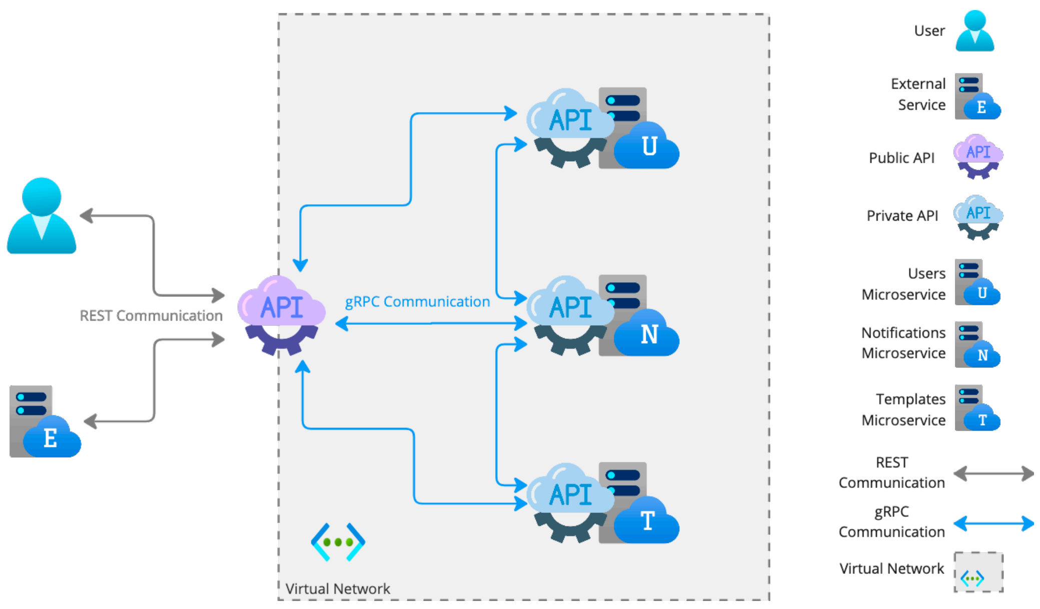


Figure 22 – Diagram of the system overview

3.3. Architecture proposal for the notifications service

The Figure 23 represents a high-level view over the notifications API service components. The main objective of this microservice it's to deliver notifications to the system end-users. These users shall be able to be updated on relevant events at real-time and on different channels, such as email, SMS, push notifications or just on the web application. It's also important to keep this service completely decoupled from the business logic of the system that is using it. This microservice relies on multiple Microsoft Azure services, since Azure it's the cloud service provider where the author's company is hosted. However, all the main providers have an equivalent offer with similar performance and usability, and the architecture can be adapted to fit on each one. Figure 23 is the first diagram of the Notification Microservice. Just as the previous Figure 22, this is a basic overview that will further be improved with more detail.

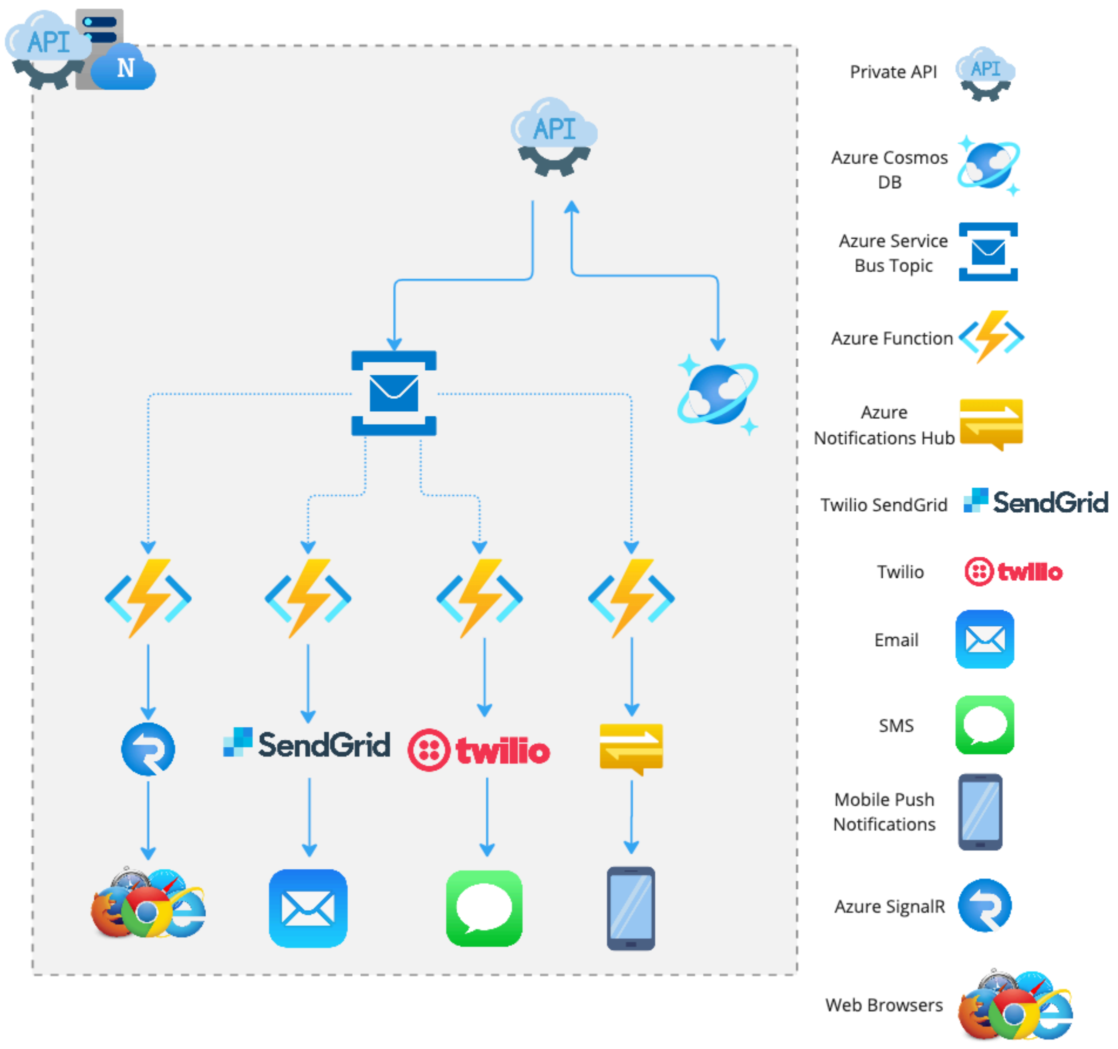


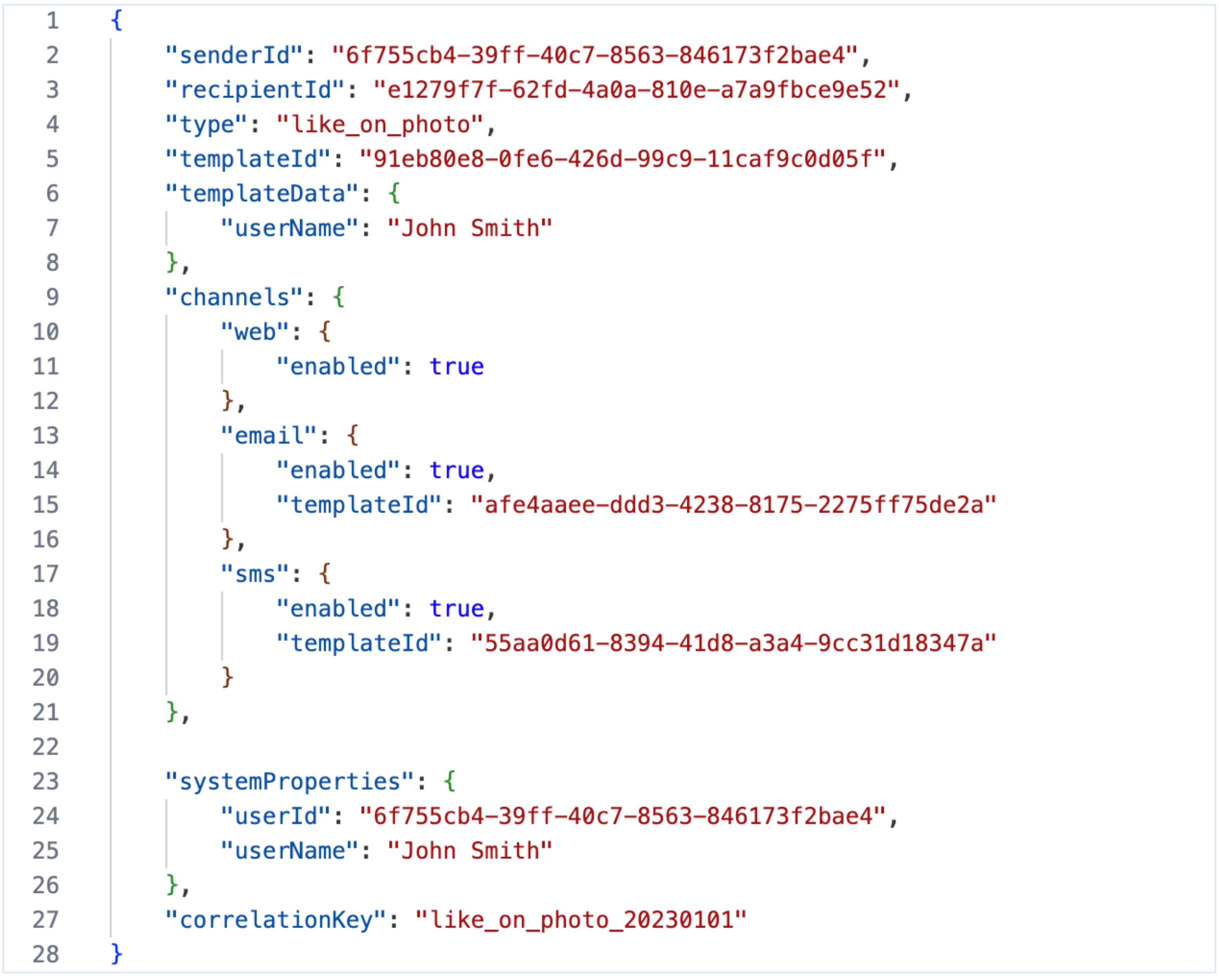
Figure 23 – First high-level diagram of the Notifications API microservice

The first important component of this architecture it's the API. This interface will be used to communicate with the service's external environment, to receive requests to create, update and retrieve notifications by other actors (such as other services or through the public API gateway, shown on the last figure and that will be called by end-users).

After applying the user's preferences, the notifications will be stored on a different Azure CosmosDB collection. Azure Cosmos DB is a NoSQL and relational database with high performance, high availability, and instant scalability. Cosmos is designed to handle large amounts of data and guarantees low latency and high throughput for read and write operations [32]. Then the notification is sent to an Azure Service Bus Topic. Azure Service Bus is a message broker with message queues and publish-subscribe topics [33]. While queues are used to deliver messages to a single consumer, topics allow a one-to-many form of communication in a publish and subscribe pattern [34]. When a message is added to a service bus topic, all the subscribers are notified. The author's proposal is that each platform has its own subscriber that will be responsible for send the notification for that channel. This way, the implementation of each subscriber is cleaner, independent, and easier to extend. In addition, if a specific channel has any problem and the sending fails, the users will still be receiving notifications on the remaining channels. On this architecture plan, the subscribers are Azure Functions, a serverless Functions-as-a-Service (FaaS) solution that allows to implement event-driven logic, called "functions" that will run for limited amount of time. These functions can be triggered by specific events such as HTTP requests, timers, service bus events, and others [35]. Being serverless means that the developers do not have to worry about server scaling, resource's computing power, maintenance, or availability, because all these concerns are handled by the cloud service provider [36]. We can see in the previous figure 4 different Azure Functions. When a notification is published on the topic, each required function (depending on the notification's configuration) will be triggered, and each one will be responsible to send it to a specific channel using different technologies. For in-app notifications, the author proposes to use Azure SignalR, a service that offers real-time communications between the server and clients, without the need to keep polling for updates, enabling instant messaging delivery, collaborative features, streaming (and others) on web and mobile applications [37]. Azure Notifications Hub is a cross-platform support service designed to send push notifications to mobile devices (including iOS, Android, Windows, etc.) from the back-end service, removing the overload of dealing with multiple platform-specific integrations [38]. Twilio is a cloud communications platform that offers multiple types of communications, including SMS and email (with SendGrid), with very developer-friendly APIs and extensive documentation.

3.4. Notification structure

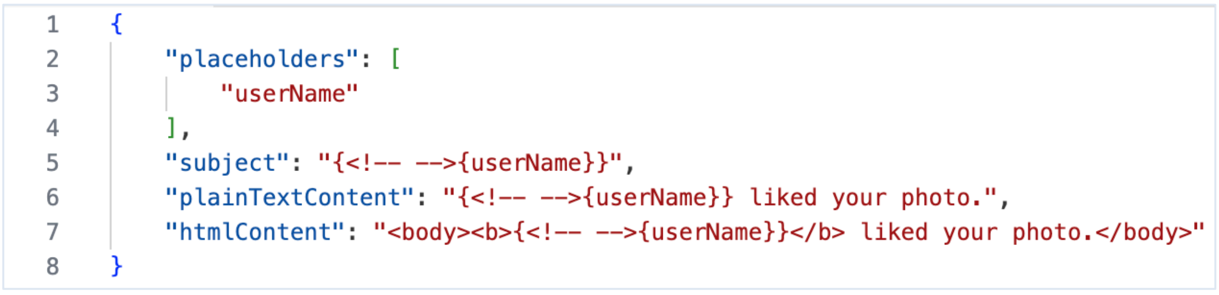
The Figure 24 represents an example of the model that the notifications API expects to receive to create a new notification. The “senderId” is the ID of the user that is sending the notification. On the other hand, the “recipientId” is the ID of the notification’s target. The “type” is self-explanatory, and the “channels” are the different platforms which the sender will try to use to deliver the notification. The “systemProperties” is metadata.



```json
{
    "senderId": "6f755cb4-39ff-40c7-8563-846173f2bae4",
    "recipientId": "e1279f7f-62fd-4a0a-810e-a7a9fbce9e52",
    "type": "like_on_photo",
    "templateId": "91eb80e8-0fe6-426d-99c9-11caf9c0d05f",
    "templateData": {
        "userName": "John Smith"
    },
    "channels": {
        "web": {
            "enabled": true
        },
        "email": {
            "enabled": true,
            "templateId": "afe4aaee-ddd3-4238-8175-2275ff75de2a"
        },
        "sms": {
            "enabled": true,
            "templateId": "55aa0d61-8394-41d8-a3a4-9cc31d18347a"
        }
    },

    "systemProperties": {
        "userId": "6f755cb4-39ff-40c7-8563-846173f2bae4",
        "userName": "John Smith"
    },
    "correlationKey": "like_on_photo_20230101"
}
```

Figure 24 – Notification POST model

The “templateId” is the reference ID of the template that should be used to display the notification. Each template may have multiple placeholders, that will be replaced by the values sent on the “templateData”. The template may be different for each channel, so it is possible to override the default template. An example of a possible template is presented on the Figure 25.



```
{
    "placeholders": [
        "userName"
    ],
    "subject": "{<!-- -->{userName}}",
    "plainTextContent": "{<!-- -->{userName}} liked your photo.",
    "htmlContent": "<body><b>{<!-- -->{userName}}</b> liked your photo.</body>"
}
```

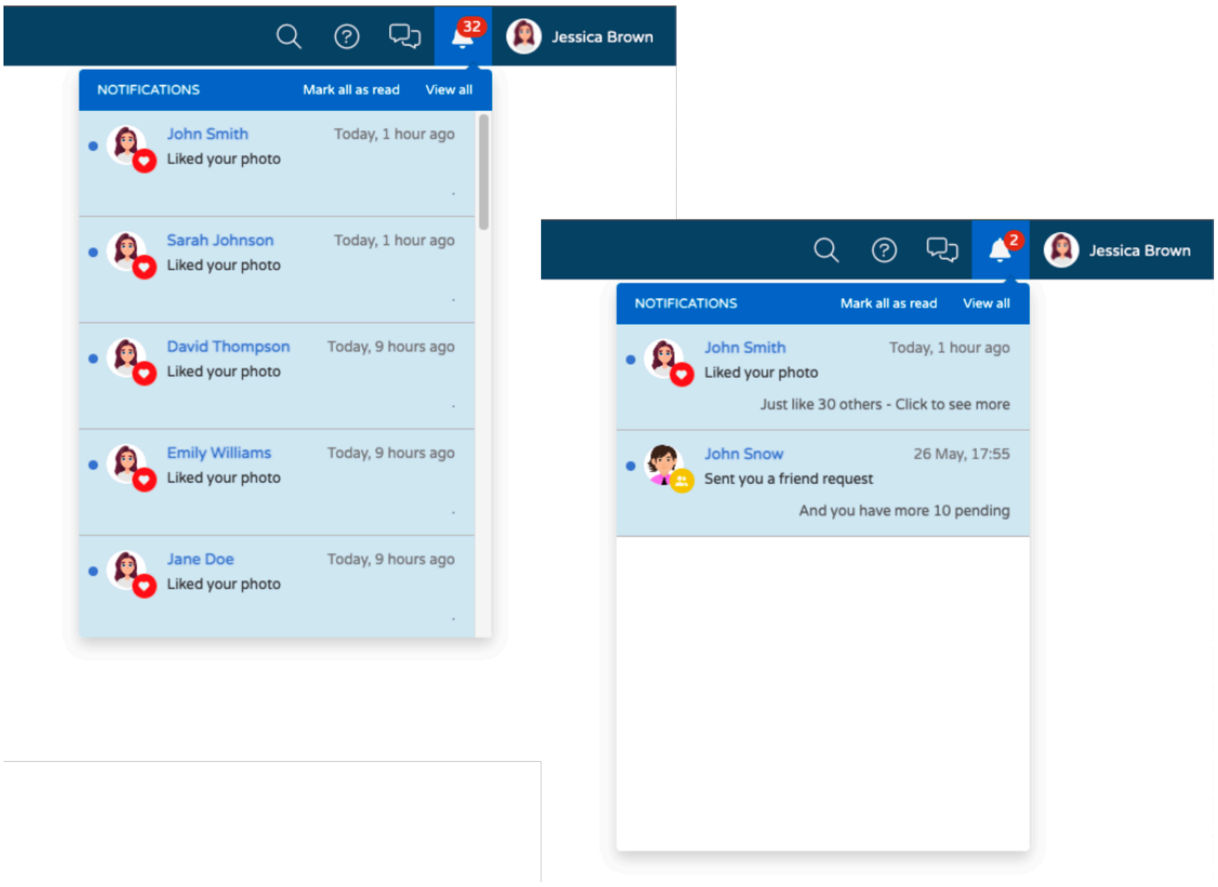
Figure 25 – Template POST model

Systems can generate a lot of notifications per second. Sometimes even a single action can generate multiple notifications for the same user. For example, if a popular user posts a photo on a social network, it will probably have a lot of engagement, resulting in multiple comments and reactions from other users. If this popular user receives a single notification for each user that commented or reacted, the user's notifications list will grow instantly, becoming unreadable and increasing the risk of missing something important. For this reason, it's usual to see the most famous social networks grouping the notifications by some context. People are used to see something like "John Doe and 30 others liked your photo." This is a lot more readable and useful that a list of 31 rows. Having this is mind, is critical for this service to have a way of correlate multiple notifications considering a given context. The author's proposal is that each notification should have a correlation key defined on the moment of its creation. For the previous example, a possible correlation key could be "like_on_photo_12345". Every notification that aims to notify the user from a new like on his photo should include this correlation key. In this manner, the service can understand that different notifications have the same context and that should be summarized on the same notification group information.

The notification group information works like a persistent state and holds relevant information about a group of notifications that have the same correlation key, including the total counter, the unread counter, the date of the last notification received and an ordered small set of the last ones. On creation, the service verifies if the correlation key belongs to a new notification group or if it already exists, to understand if it must be created or updated with the latest information. All these details could be computed at runtime, but it wouldn't be effective, and it would add a significant overload on the request performance.

With this approach the granularity of the context is also very flexible, and it transfers the onus of the definition to the notification sender, which is the best entity to define it considering that knows the context well. In the previous example, if the sender decides that the "like" notifications should also be grouped by day, it would be as simple as adding the date on the correlation key, to be something like "like_on_photo_12345_2023_01_01".

The Figure 26 helps to visually understand what the correlation key aims to achieve. On the left side, all the unread notifications are listed, without considering the correlation key. On the right side, each "row" represents a different notification group (which means a different correlation key). The service will support both versions, but it's clear that the second approach is much more user friendly and useful.



John Smith Today, 1 hour ago
Liked your photo

Sarah Johnson Today, 1 hour ago
Liked your photo

David Thompson Today, 9 hours ago
Liked your photo

Emily Williams Today, 9 hours ago
Liked your photo

Jane Doe Today, 9 hours ago
Liked your photo

John Smith Today, 1 hour ago
Liked your photo
Just like 30 others - Click to see more

John Snow 26 May, 17:55
Sent you a friend request
And you have more 10 pending

Figure 26 – Comparison between an extensive list of notifications and the listing of the notification groups

The Figure 27 is a UML Activity Diagram that helps to better visualize the process flow when the notification API service receives a request to create a new notification.

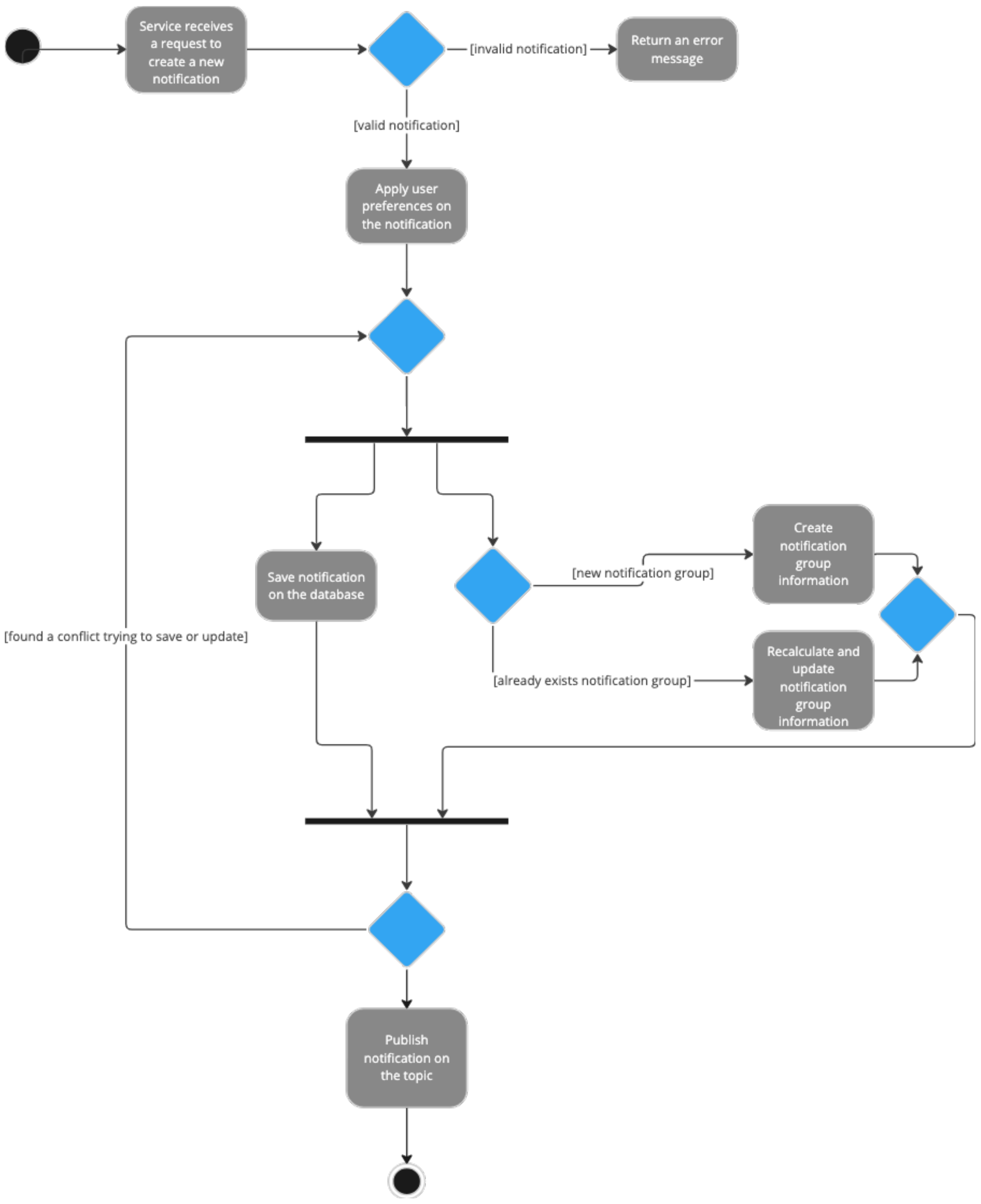

Service receives a request to create a new notification
[invalid notification]
Return an error message
[valid notification]
Apply user preferences on the notification
Save notification on the database
[new notification group]
Create notification group information
[already exists notification group]
Recalculate and update notification group information
[found a conflict trying to save or update]
Publish notification on the topic

Figure 27 – UML Activity Diagram for a notification creation

3.5. Azure Container Apps and Dapr integration

3.5.1. Overview

Azure Container Apps (ACA) provides an abstraction over Kubernetes. It is one of the many ways that Azure offers to run microservices and containerized applications on a serverless platform [39]. The main difference when compared to Azure Kubernetes Service (AKS) it's that Azure Container Apps (ACA) doesn't provide direct access to the Kubernetes API, meaning that it's less flexible. However, ACA it's easier to use and it encapsulates the best-practices to handle Kubernetes. As the reader will see forward, the ACA allows to set multiple declarative scaling rules so it can automatically scale horizontally. These rules may be driven from different source types, such as HTTP traffic (based on the number of concurrent HTTP requests), CPU and memory load, event-driven triggers with KEDA-supported scalers, and others. Each container app may have 1 or more active revisions (immutable snapshot of a container app version [40]), and each revision may have many replicas (or instances) running at the same time. Inside each replica it's possible to run one or more containers. When the application (revision) scales out, new replicas are created, and it can scale to a maximum of 300 replicas. On the other hand, using specific scaling rules, it can also scale to 0 replicas, which is an important feature since it won't be billed usage charges. The Figure 28 helps to understand the hierarchy between container apps, revisions, replicas, and containers.

ACA provides built-in integration with Dapr. When Dapr is enabled, the ACA launches a secondary container alongside the application, often called as "Dapr sidecar".

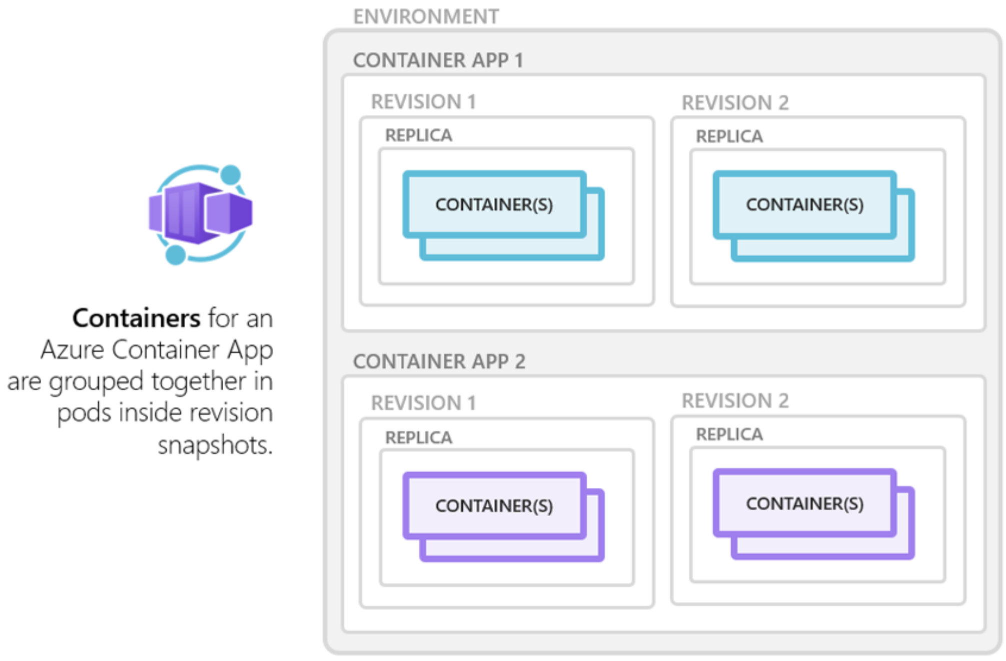


Figure 28 - Azure Container Apps [40]

3.5.2. Zero downtime deployment

Azure Container Apps automatically creates new revisions when any revision-scope change is made. A revision-scope change can be a change on the revision suffix, on the container configuration or on the scaling rules. So, when a new version of the application is deployed (using a different image version, for example), a new revision is created. While the latest revision isn't ready, the container app keeps sending the traffic to the previous one. To be considered ready, a revision should have at least one of its replicas ready, that happens when all its containers start, and their configured probes are reporting as healthy. At this stage, the traffic starts to be forward to the new revision, avoiding any downtime during the deployment process [41]. Revisions also simplifies the task of rolling back to a previous version, since the ACA can retain inactive revisions, allowing to revert to an earlier version with a single click.

3.5.3. Service-to-service invocation

On section 2, the author explored the advantages of using Dapr to develop distributed applications. In this project, some of the Dapr building blocks were used. Service-to-service invocation was one of them, as it works as service discovery. The next figure shows how Dapr service invocation works. When the Service A wants to call the Service B, it does it by calling its own Dapr sidecar that is running locally on the same pod (when hosted on Kubernetes). Then the sidecar will find the Service B's location using the name resolution component, and forward the request to the Service B's sidecar, that will finally delivery it to the specified endpoint or method on Service B. Figure 29 shows the communication flow between two services when using service-to-service invocation.

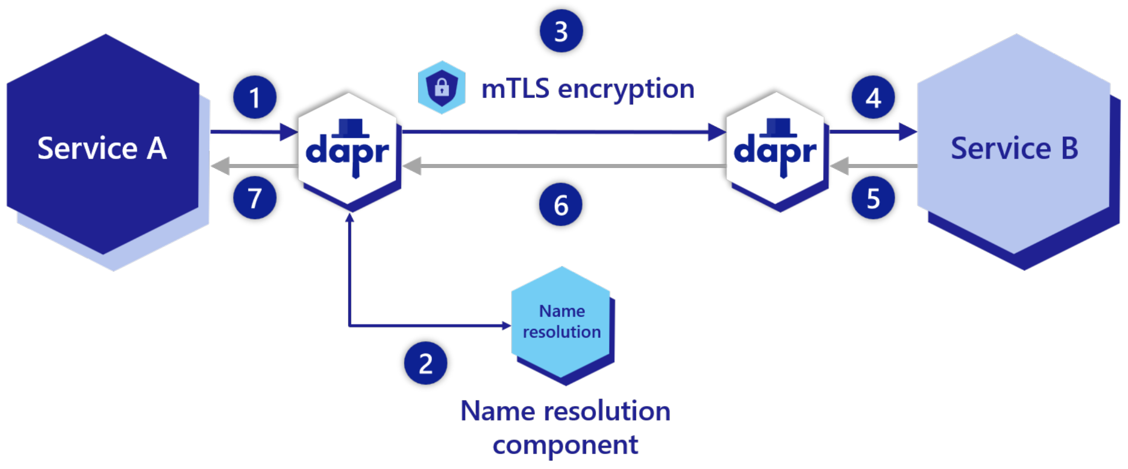


Figure 29 – Dapr service-to-service invocation [42]

Beside the service discovery feature, the Dapr service invocation offers other advantages, such as fault tolerance, including resiliency by retrying failed calls on transient errors. These automatic retries are performed with a backoff period between each one.

The public gateway is one of the services that takes advantage of this feature. It forwards the requests to the other services using the Dapr SDK and the Dapr ID of the service that should be invoked. This Dapr ID is defined on the service deployment and should be immutable to avoid breaking changes.

It's possible to see in the Figure 30 that all communications between services are done through gRPC and between the sidecars of each service. Notice that all Azure Container Apps are running on the same Azure Container App Environment. All container apps in the same environment share the same virtual network and log to the same destination. Different container apps must be in the same environment to communicate via Dapr and use the same Dapr configuration as well.

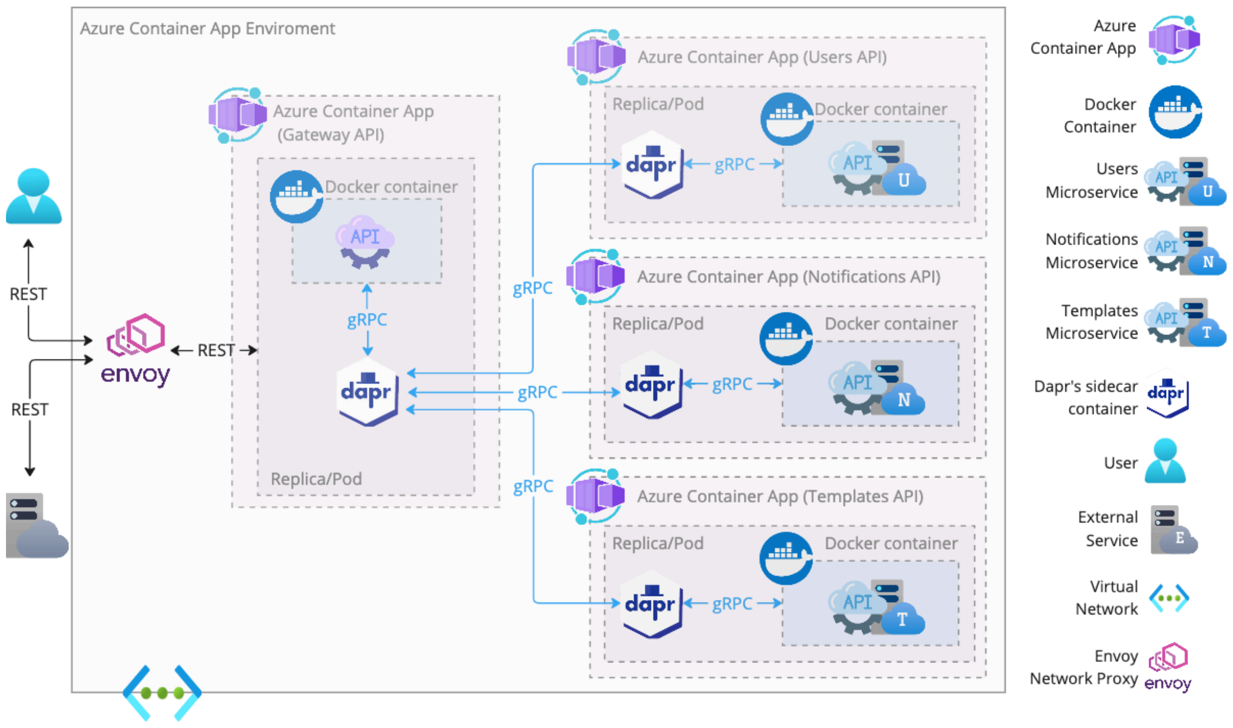
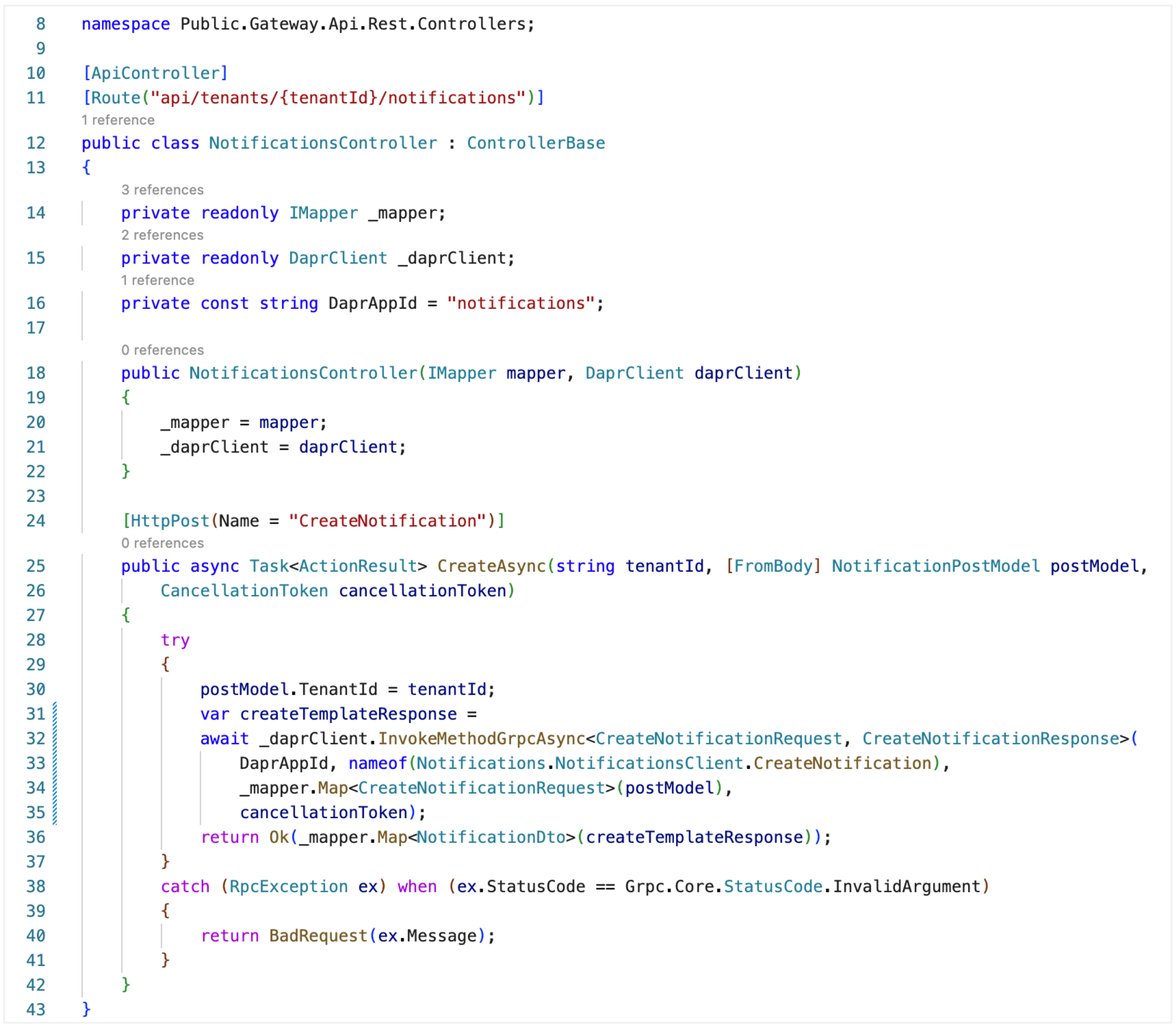


Figure 30 - Communication between the public gateway and the other microservices using Azure Container Apps and Dapr

The ingress is what enables the Container Apps to communicate with the public web and other resources on the same virtual network or in the same environment. When the ingress is enabled, it can be configured to accept traffic from anywhere (external) or just internal (virtual network and Container Apps Environment). When enabling ingress, there is no need to create or manage any other resource to enable incoming traffic (such as a public IP address or a load balancer).

By default, Azure Container Apps uses Envoy as an edge network proxy. A network proxy (or proxy server) works as an intermediary between two entities that want to communicate with each other. This type of service can provide many features such as an extra layer of security, load-balancing, resiliency, or routing. In the specific case of Azure Container Apps, it is mainly used to achieve load-balance capabilities (to distribute the requests across the multiple replicas of an application) and for allowing the applications to scale to zero [43]. Figure 30 is the evolution of the Figure 22 and shows in more detail how the communication flows between external network entities and the backed services running on ACA, using Dapr service-to-service invocation. The Envoy proxy receives REST requests from the public internet and forwards them to the most appropriate replica of the gateway (load balancing). Then, all the internal communications between services inside the cluster is done through each Dapr sidecar. While Envoy is responsible for load-balancing the external traffic, Dapr provides load balancing of service invocation requests.

The Figure 31 shows how the Dapr SDK allows to use the service invocation feature. In this specific case, the gateway API is forwarding a notification creation request to the notifications service. At the notifications service deployment, the container app was configured to use “notifications” as the Dapr ID. That ID is known by the gateway, and it’s saved as a constant on the respective controller. Along with the Dapr ID, the second argument that is necessary to define it's the method name that we want to invoke (or the route, in case of using HTTP). The “DaprClient” is a class provided by the “Dapr.AspNetCore” package (version 1.11.0) and its instance is provided via Dependency Injection (DI). DI is a design pattern that allows to manage and provide objects and services to other components that depend on it. Instead of creating themselves, the responsibility of creating these dependencies is normally centralized on a single place, and they are injected on a “container”, that is used by the dependents. This pattern can be useful to build modularized and loosely coupled applications. In this case, the class is injected via constructor and it’s available for all methods in the controller, but it can also be injected on each method instead.



```
namespace Public.Gateway.Api.Rest.Controllers;

[ApiController]
[Route("api/tenants/{tenantId}/notifications")]
1 reference
public class NotificationsController : ControllerBase
{
    3 references
    private readonly IMapper _mapper;
    2 references
    private readonly DaprClient _daprClient;
    1 reference
    private const string DaprAppId = "notifications";

    0 references
    public NotificationsController(IMapper mapper, DaprClient daprClient)
    {
        _mapper = mapper;
        _daprClient = daprClient;
    }

    [HttpPost(Name = "CreateNotification")]
    0 references
    public async Task<ActionResult> CreateAsync(string tenantId, [FromBody] NotificationPostModel postModel,
        CancellationToken cancellationToken)
    {
        try
        {
            postModel.TenantId = tenantId;
            var createTemplateResponse =
            await _daprClient.InvokeMethodGrpcAsync<CreateNotificationRequest, CreateNotificationResponse>(
                DaprAppId, nameof(Notifications.NotificationsClient.CreateNotification),
                _mapper.Map<CreateNotificationRequest>(postModel),
                cancellationToken);
            return Ok(_mapper.Map<NotificationDto>(createTemplateResponse));
        }
        catch (RpcException ex) when (ex.StatusCode == Grpc.Core.StatusCode.InvalidArgument)
        {
            return BadRequest(ex.Message);
        }
    }
}
```

Figure 31 – Public gateway invoking the notifications service through the Dapr SDK

The service-to-service invocation are also used in other services. Every time that a service needs to call another service, Dapr is used. The subscribers for example are using this building block to make requests to the users' service and the templates' service as well. The Figure 32 shows the communications between the microservice that sends email notifications and the other microservices. Notice that the Azure Function is also running inside an Azure Container App (and in the same environment), because this is the only way to reach the other microservices using Dapr.

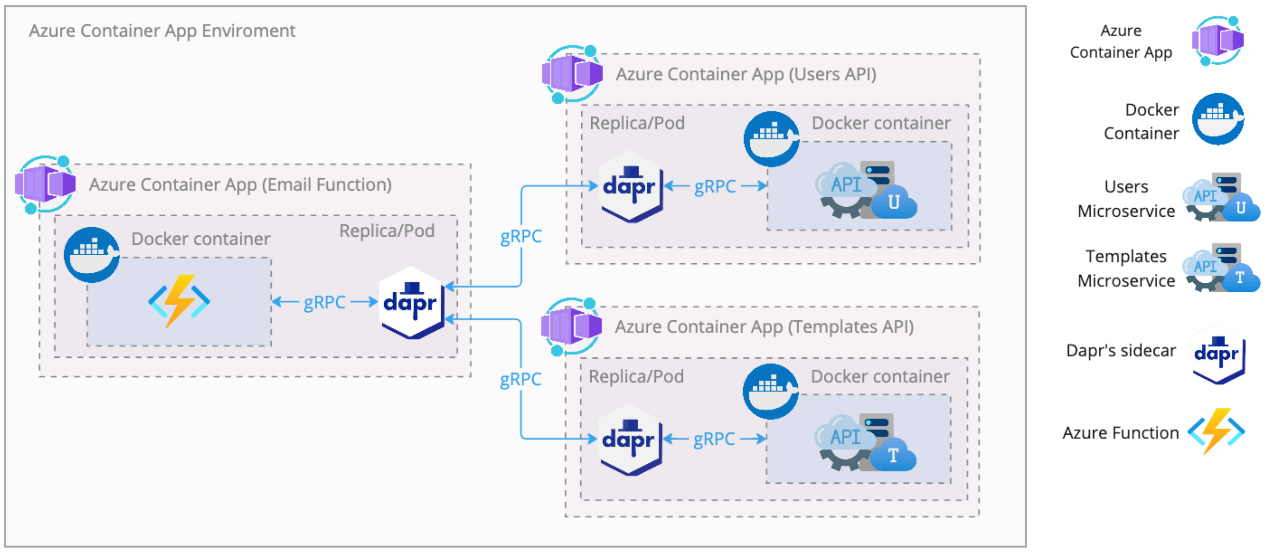


Figure 32 - Communication of the email notification service with the others microservices

3.5.4. Publish and subscribe

Dapr also provides an API to send and receive messages in a platform-agnostic way. It allows to use queues and topics from different message brokers, as seen on section 2. The Figure 33 shows how the notification service uses this Dapr building block to publish messages on an Azure Service Bus topic.

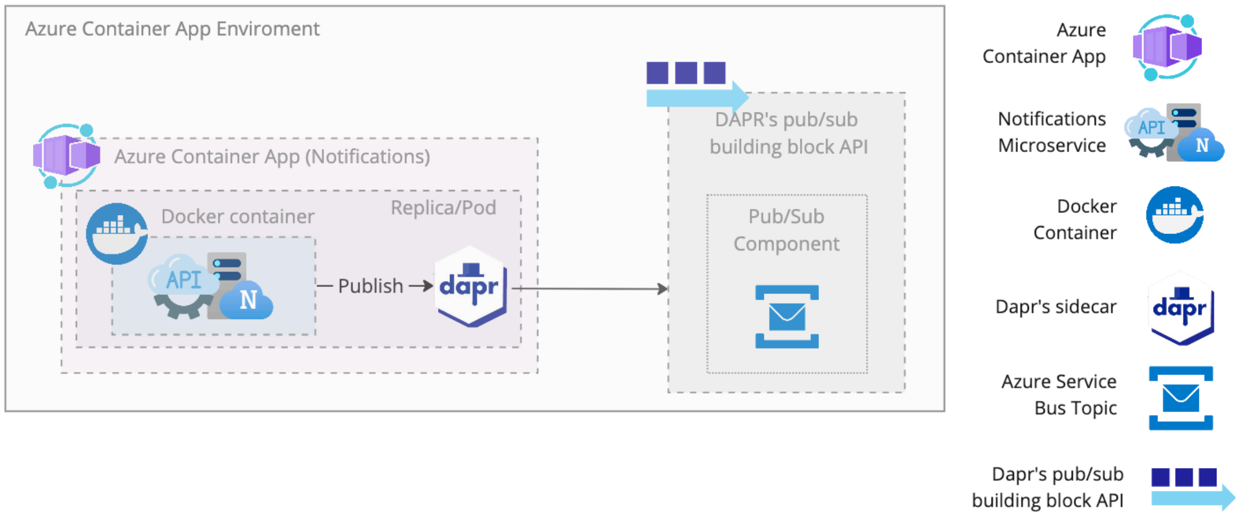
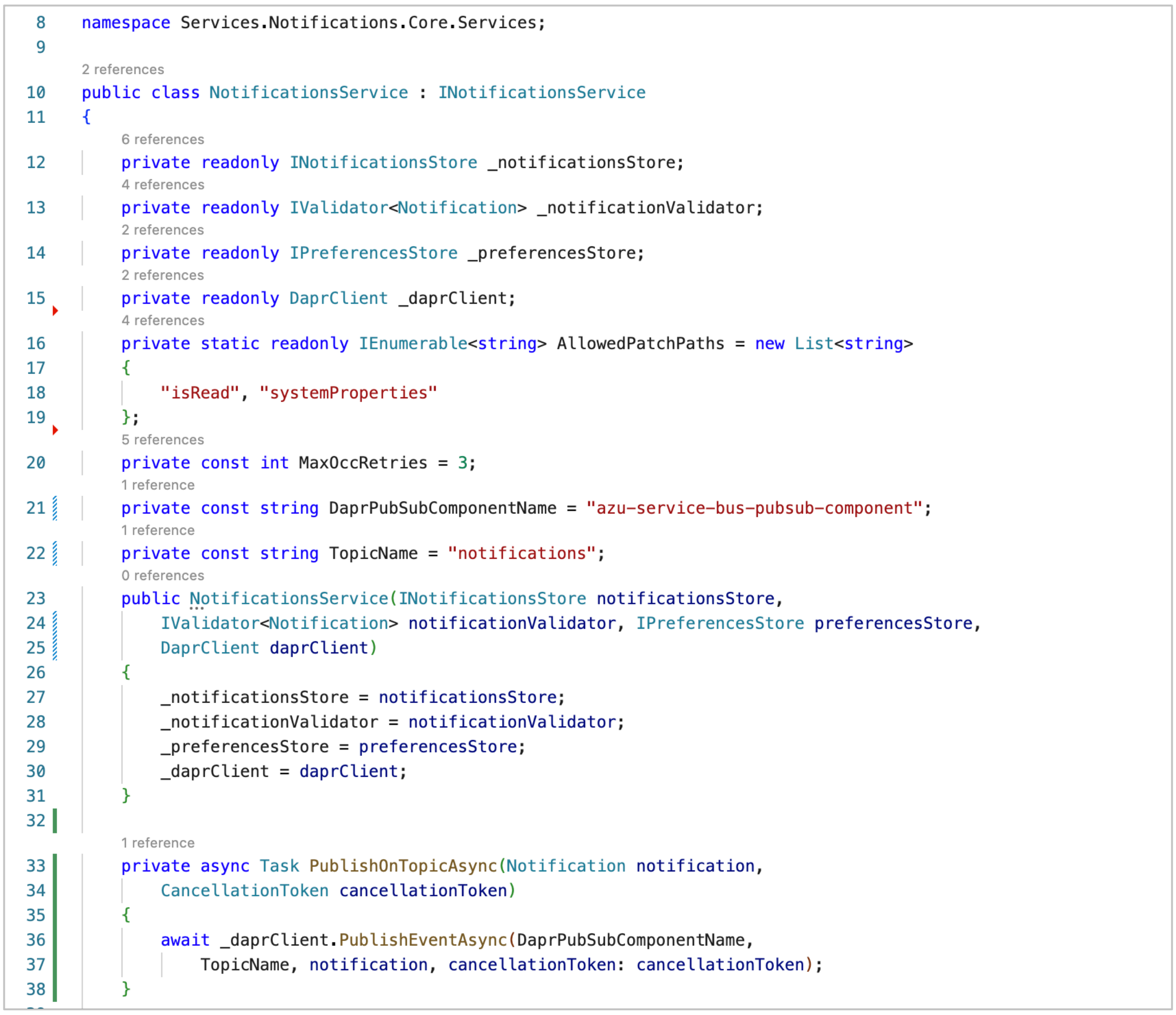

Azure Container App Enviroment
DAPR's pub/sub building block API
Azure Container App (Notifications)
Docker container
Replica/Pod
Publish
Pub/Sub Component
Azure Container App
Notifications Microservice
Docker Container
Dapr's sidecar
Azure Service Bus Topic
Dapr's pub/sub building block API

Figure 33 - Message publishing with Dapr

The notifications service makes the publish request to the Dapr sidecar, which will make a network call to the pub/sub building block API. This API will then use the publish/subscribe component that was chosen when calling the sidecar, that in this case encapsulates the Azure Service Bus as message broker. Figure 34 shows how the notifications service uses the DaprClient SDK to publish a new message (notification) on a topic.

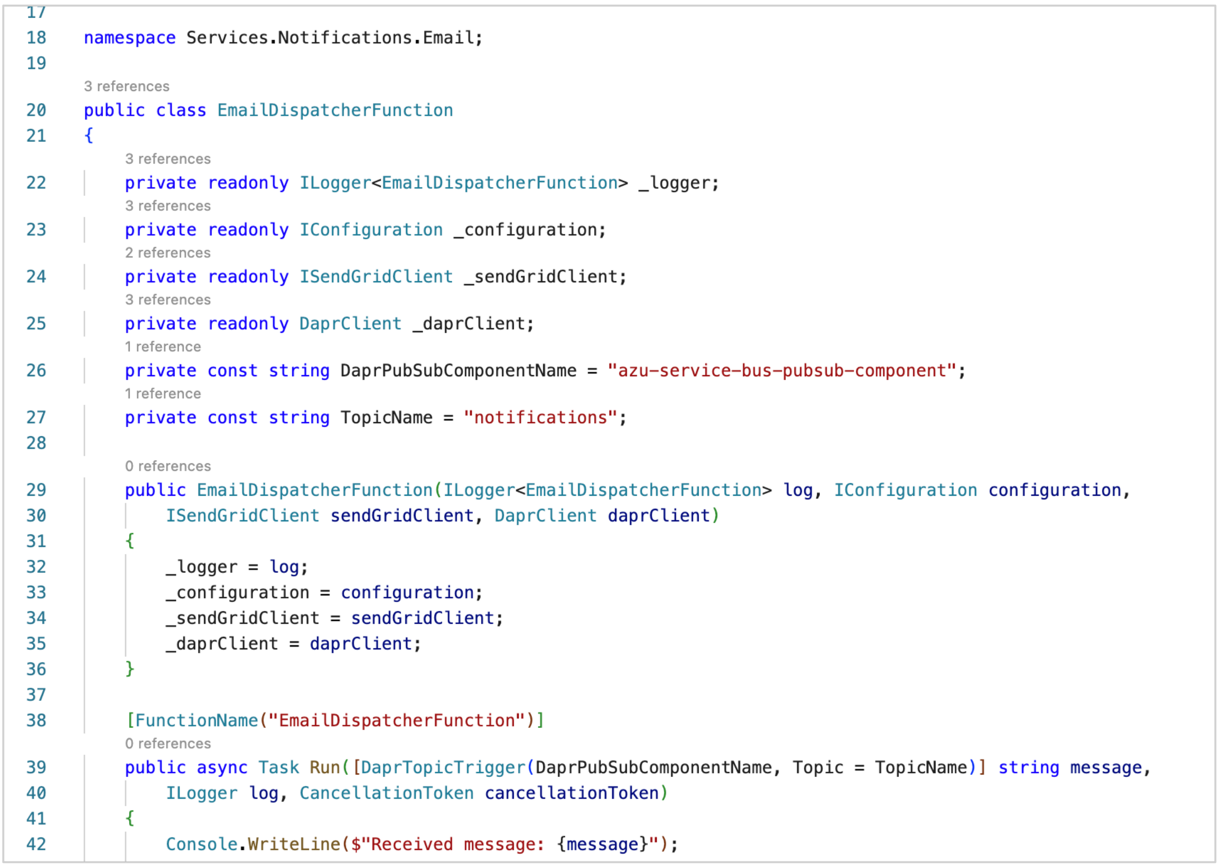


```
namespace Services.Notifications.Core.Services;

2 references
public class NotificationsService : INotificationsService
{
    6 references
    private readonly INotificationsStore _notificationsStore;
    4 references
    private readonly IValidator<Notification> _notificationValidator;
    2 references
    private readonly IPreferencesStore _preferencesStore;
    2 references
    private readonly DaprClient _daprClient;
    4 references
    private static readonly IEnumerable<string> AllowedPatchPaths = new List<string>
    {
        "isRead", "systemProperties"
    };
    5 references
    private const int MaxOccRetries = 3;
    1 reference
    private const string DaprPubSubComponentName = "azu-service-bus-pubsub-component";
    1 reference
    private const string TopicName = "notifications";
    0 references
    public NotificationsService(INotificationsStore notificationsStore,
        IValidator<Notification> notificationValidator, IPreferencesStore preferencesStore,
        DaprClient daprClient)
    {
        _notificationsStore = notificationsStore;
        _notificationValidator = notificationValidator;
        _preferencesStore = preferencesStore;
        _daprClient = daprClient;
    }

    1 reference
    private async Task PublishOnTopicAsync(Notification notification,
        CancellationToken cancellationToken)
    {
        await _daprClient.PublishEventAsync(DaprPubSubComponentName,
            TopicName, notification, cancellationToken: cancellationToken);
    }
```

Figure 34 – Dapr SDK to publish events

Using this building block, the application is completely independent from the Azure Service Bus. It does not even know what message broker is using. If the application's administrator decides to stop using the Azure Service Bus and start using Redis (for example), the code would remain the same. The only change would be on the Dapr component configuration, that instead of having the settings needed to connect to the Azure Service Bus would have the settings to connect to the Redis service.

As seen earlier in this article, Azure Functions allows to write event-driven code that runs for a limited time. In addition to all the built-in triggers that are available to use out of the box, Dapr added support for triggering functions on a Dapr service invocation, publish/subscribe events, or Dapr input/output bindings. The next figure shows how the email function is triggered. Instead of using the Azure Service Bus Topic trigger, that is obviously specific for this message broker, it uses the Dapr topic trigger to fire the function when a message is added to the defined topic. This way, the subscribers are kept agnostic from the infrastructure that is being used, just like the publishers. To use this type of trigger, the function app must be aware of the Dapr Publish/Subscribe component name that it should use, and the topic name as well. These names should match the ones that are being used on the publishing action, as shown in Figure 35.



```
namespace Services.Notifications.Email;

3 references
public class EmailDispatcherFunction
{
    3 references
    private readonly ILogger<EmailDispatcherFunction> _logger;
    3 references
    private readonly IConfiguration _configuration;
    2 references
    private readonly ISendGridClient _sendGridClient;
    3 references
    private readonly DaprClient _daprClient;
    1 reference
    private const string DaprPubSubComponentName = "azu-service-bus-pubsub-component";
    1 reference
    private const string TopicName = "notifications";

    0 references
    public EmailDispatcherFunction(ILogger<EmailDispatcherFunction> log, IConfiguration configuration,
        ISendGridClient sendGridClient, DaprClient daprClient)
    {
        _logger = log;
        _configuration = configuration;
        _sendGridClient = sendGridClient;
        _daprClient = daprClient;
    }

    [FunctionName("EmailDispatcherFunction")]
    0 references
    public async Task Run([DaprTopicTrigger(DaprPubSubComponentName, Topic = TopicName)] string message,
        ILogger log, CancellationToken cancellationToken)
    {
        Console.WriteLine($"Received message: {message}");
```

Figure 35 – Dapr topic trigger used to trigger the Azure Function

In the Figure 23 in section 3.1, the author has shown a high-level diagram of the proposed architecture. The Figure 36 is a more detailed version, after introducing new concepts that were used on the implementation, and that are consolidated in this following figure. In fact, the diagram shows multiple services. The Notifications API is only responsible for storing the notifications and sending them into a topic. Each topic subscriber is seen as separated service, since it makes sense to independently deliver, deploy and scale each one.

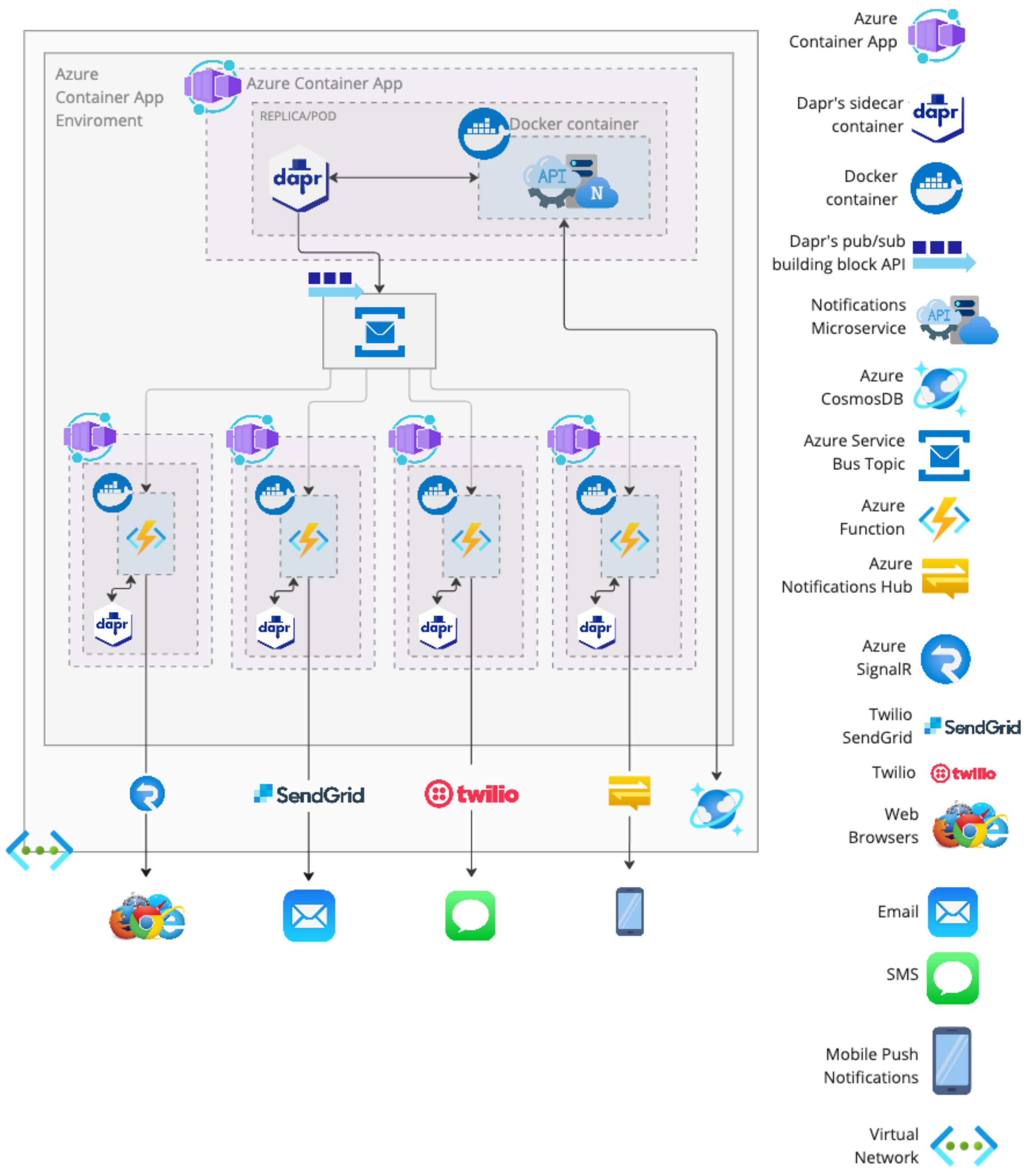


Figure 36 - Detailed architecture of the Notifications API and subscribers

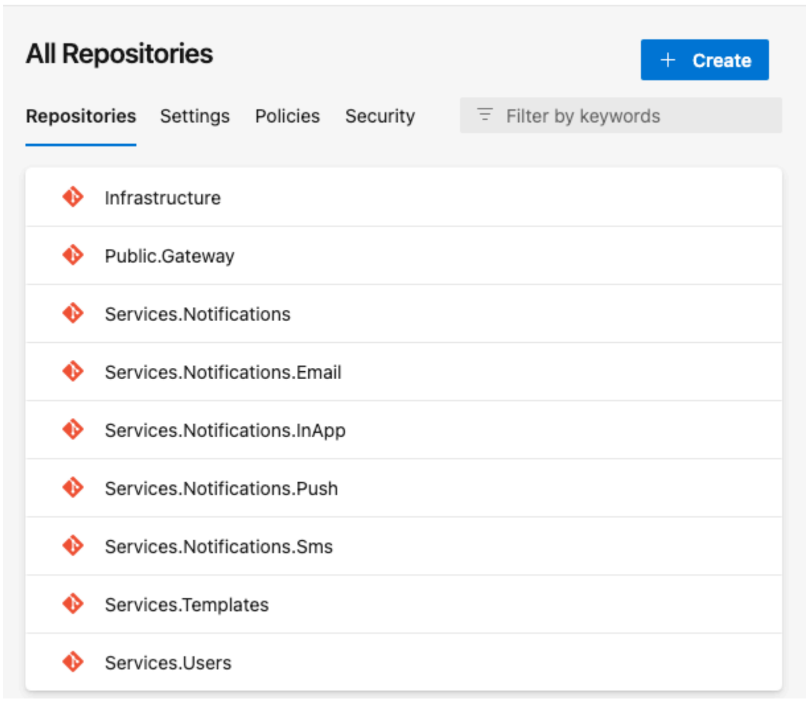
3.6. Build, deployment, and scaling

3.6.1. Overview

Since the Azure is the cloud provider of the author's company and it's where the application is hosted, he chose to use the Azure DevOps to manage this project due to the easy integration that it offers. Azure DevOps is a set of services to plan, develop, test, deploy and monitor software projects, such as Azure boards (to plan and track work), Azure Pipelines (to build, test and deploy software), Azure Repos (for Git hosting), Azure Test Plans (manual and exploratory DevOps testing tools) and Azure Artifacts (package repository) [44]. For this project, the only services used were the Azure Repos, Azure Pipelines and Azure Artifacts.

3.6.2. Repositories

Each service has its own GIT repository. The only repository that does not have an associated service is the "Infrastructure" repository. This repository holds the deployment files required to create the resources that are needed and common to more than one service, such as the azure cosmos account, the database, the virtual network, the service bus, and a few others. All these resources are shared among services. When a resource belongs to a single scope of a service, the deployment file is placed on the respective repository. The Figure 37 shows all the repositories on the Azure DevOps that the author used on this project.



Infrastructure
Public.Gateway
Services.Notifications
Services.Notifications.Email
Services.Notifications.InApp
Services.Notifications.Push
Services.Notifications.Sms
Services.Templates
Services.Users

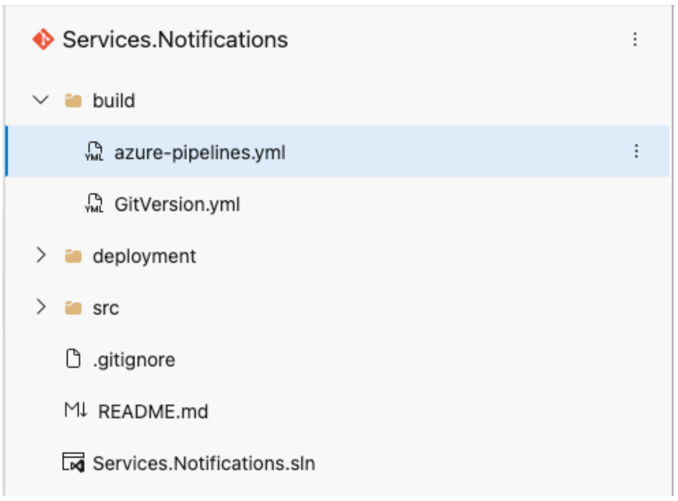
Figure 37 - Repositories on Azure DevOps

3.6.3. Build pipelines

Each service has its own build pipeline. The build pipeline is defined on a YAML file that is stored on the corresponding repository, inside the "build" folder. YAML is human-readable data serialization language that is often used to create configuration files. In this file, we can configure multiple tasks that we want to execute on the build. On this project, all these YAML files are very similar, and the main tasks are:

- Versioning
- Build the projects
- Execute the unit tests
- Pack and publish the API client packages
- Build the docker image and push it to the repository
- Publish the build artifact

Figure 38 is a print screen from the Notifications service repository, used to illustrate the files that compose a service build folder.



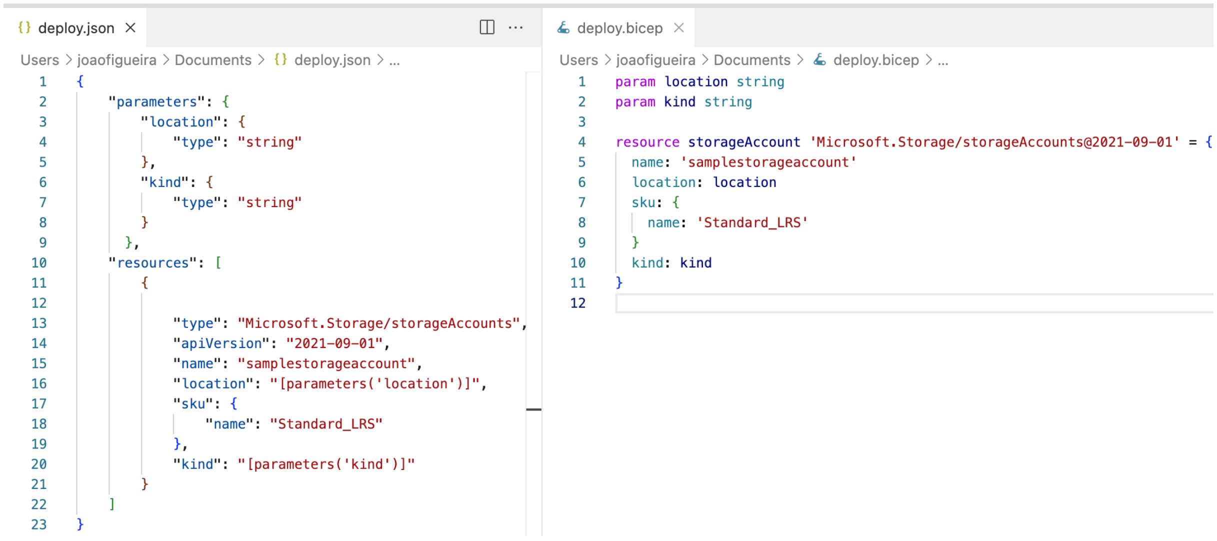
Services.Notifications
build
azure-pipelines.yml
GitVersion.yml
deployment
src
.gitignore
README.md
Services.Notifications.sln

Figure 38 – Notifications Service build folder

3.6.4. Release pipelines

The release pipelines are triggered by the build pipelines. When a build succeeds (from the main branch), a new release is automatically created and installed. Each service/build pipeline has its own release pipeline as well. These releases will deploy the actual resources on the cloud (or update them).

In this project, the author used an Infrastructure as code (IaC) approach. IaC is a way of automating the provisioning of the infrastructure (resources), that improves its consistency considering that it avoids the manual configurations that are error prone and hard to track over time. This way, all the environments are configured using well defined and versioned configuration files with a high-level descriptive coding language [45]. Azure provides support to IaC through the Azure Resource Manager (ARM), which has an API that allows to use ARM templates or bicep files to define the infrastructure that must be deployed. ARM templates are JavaScript Object Notation (JSON) files, while Bicep is a domain-specific language (DSL) that uses declarative syntax [46]. The author chose to use bicep because it provides a cleaner and more concise syntax. In addition, Microsoft has released an official extension for editing bicep files on Visual Studio Code, that includes syntax validation, intellisense, code navigation and some other features that are very helpful and improves the developing experience. The Figure 39 compares an ARM Template with a bicep file, where both are deploying the exact same sample resource.


{} deploy.json
Users > joaofigueira > Documents > {} deploy.json > ...
deploy.bicep
Users > joaofigueira > Documents > deploy.bicep > ...

```json
{
    "parameters": {
        "location": {
            "type": "string"
        },
        "kind": {
            "type": "string"
        }
    },
    "resources": [
        {

            "type": "Microsoft.Storage/storageAccounts",
            "apiVersion": "2021-09-01",
            "name": "samplestorageaccount",
            "location": "[parameters('location')]",
            "sku": {
                "name": "Standard_LRS"
            },
            "kind": "[parameters('kind')]"
        }
    ]
}
```

```bicep
param location string
param kind string

resource storageAccount 'Microsoft.Storage/storageAccounts@2021-09-01' = {
  name: 'samplestorageaccount'
  location: location
  sku: {
    name: 'Standard_LRS'
  }
  kind: kind
}

```

Figure 39 - ARM Template vs. Bicep template

When the Azure Resource Manager receives a request to deploy a bicep, it will convert each resource into a REST API operation and forward it to the correct resource provider [47]. This resource provider is defined by the “type” attribute, and the “apiVersion” is used as the API version for the REST operation to avoid being exposed to possible breaking changes introduced in later versions.

The bicep file for each service is placed inside of a folder named “deployment” alongside a shell script and multiple parameter files (one for each environment). The parameters are JSON files allows to define different parameters for each environment to have different configurations considering the different requirements. For example, a production environment will normally have more resource power and more backup instances that a development environment. These types of differences are configurable through these files. The shell script contains a command from the Azure Command-Line (Azure CLI), that will deploy the bicep file using a parameters file. Figure 40 is a print screen from the Notifications service repository, used to illustrate the files that compose a service build folder.

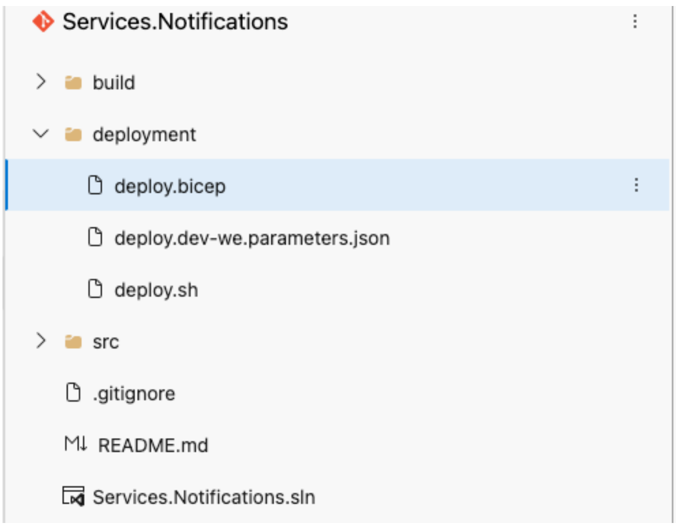
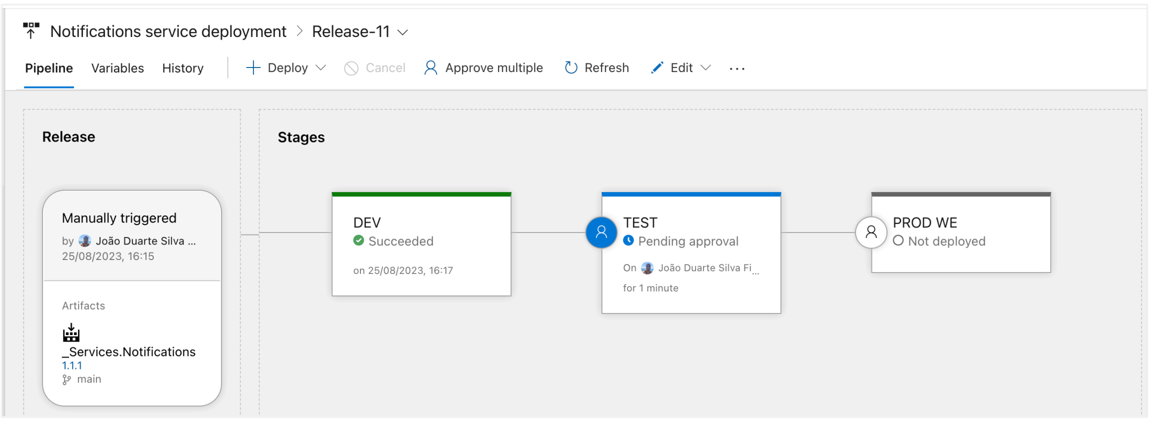

Services.Notifications
build
deployment
deploy.bicep
deploy.dev-we.parameters.json
deploy.sh
src
.gitignore
README.md
Services.Notifications.sln

Figure 40 – Notifications Service deployment folder

The job of these release pipelines is simply to execute the shell script for the desired environment and with the right parameters. In the Figure 41, we can see the release was deployed in the development environment and is now waiting for approval so it can be deployed in test, and then rolled out into production.


Notifications service deployment > Release-11

Pipeline Variables History | Deploy Cancel Approve multiple Refresh Edit ···

Release

Manually triggered
by João Duarte Silva ...
25/08/2023, 16:15

Artifacts
_Services.Notifications
1.1.1
main

Stages

DEV
Succeeded
on 25/08/2023, 16:17

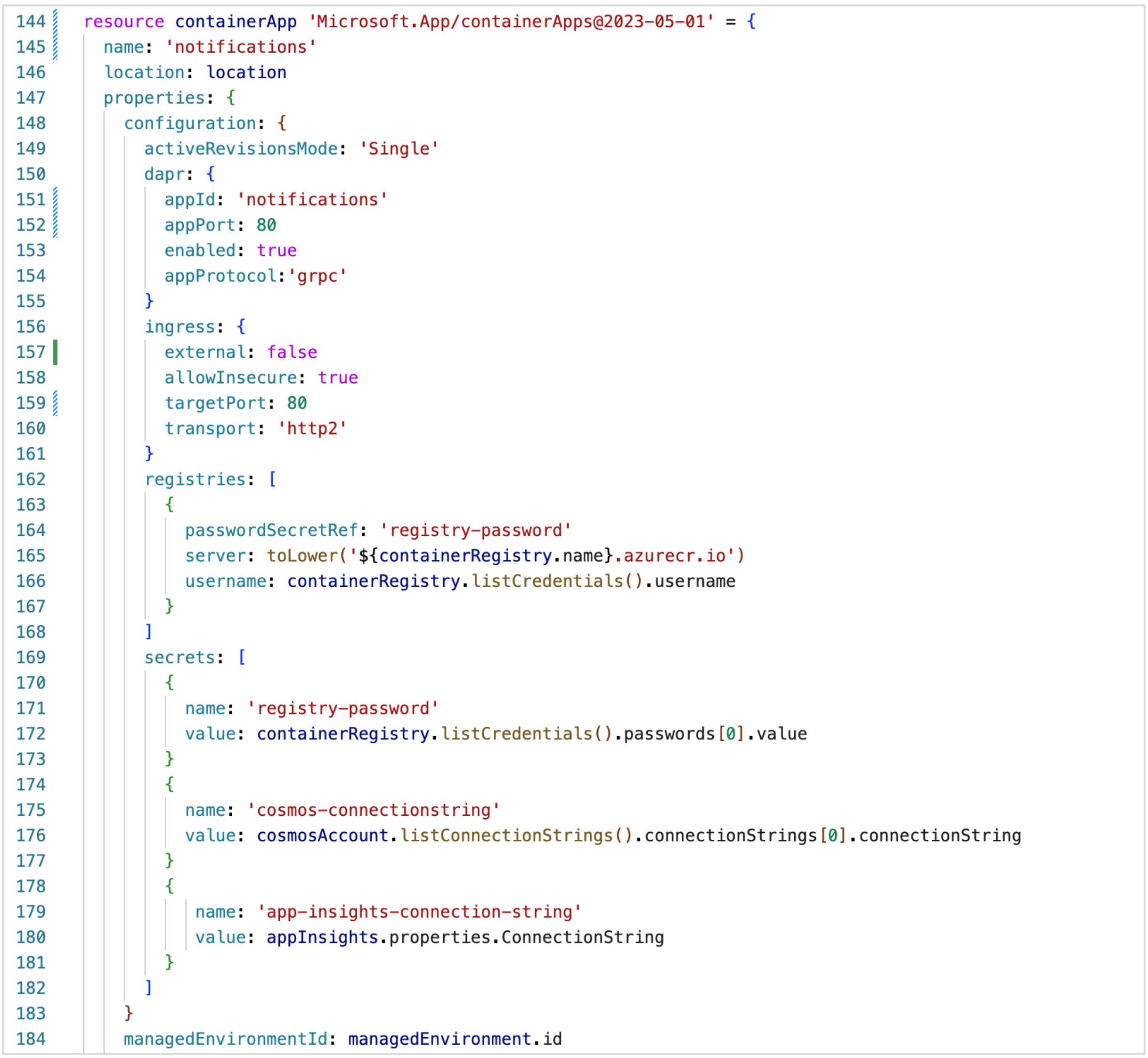
TEST
Pending approval
On João Duarte Silva Fi...
for 1 minute

PROD WE
Not deployed

Figure 41 - Notifications API release example

3.6.5. Deployment, Autohealing and Autoscaling

For the scope of this project, there are two resources that are worth analysing to understand how the application will scale: the Azure Container App that is hosting the notification API and the Azure Container App that is hosting one of the subscribers (in this case the author has chosen the email subscriber). The Figure 42 is the first part of the notification API resource, that will be presented in three different parts.



```
resource containerApp 'Microsoft.App/containerApps@2023-05-01' = {
  name: 'notifications'
  location: location
  properties: {
    configuration: {
      activeRevisionsMode: 'Single'
      dapr: {
        appId: 'notifications'
        appPort: 80
        enabled: true
        appProtocol:'grpc'
      }
      ingress: {
        external: false
        allowInsecure: true
        targetPort: 80
        transport: 'http2'
      }
      registries: [
        {
          passwordSecretRef: 'registry-password'
          server: toLower('${containerRegistry.name}.azurecr.io')
          username: containerRegistry.listCredentials().username
        }
      ]
      secrets: [
        {
          name: 'registry-password'
          value: containerRegistry.listCredentials().passwords[0].value
        }
        {
          name: 'cosmos-connectionstring'
          value: cosmosAccount.listConnectionStrings().connectionStrings[0].connectionString
        }
        {
           name: 'app-insights-connection-string'
           value: appInsights.properties.ConnectionString
        }
      ]
    }
    managedEnvironmentId: managedEnvironment.id
```

Figure 42 – Bicep template to deploy the Notifications API (PART I)

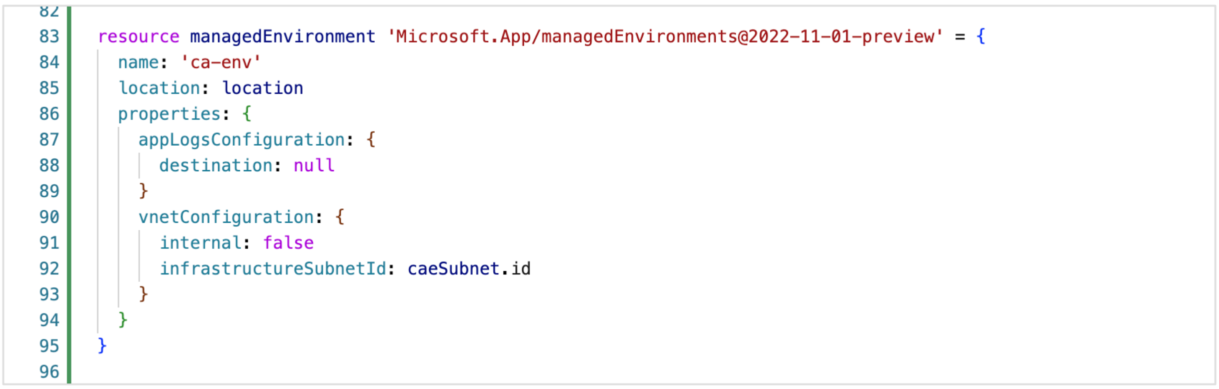
As previously said, the Azure Container App offers built-in integration with Dapr and for that reason, setting the “properties.configuration.Dapr.enabled” flag to “true” will automatically deploy the sidecar inside the same pod. Furthermore, the “properties.configuration.Dapr.appPort” and the “properties.configuration.Dapr.appProtocol” tells Dapr which port the application is listening on and which protocol (http or gRPC) it is using, respectively. The “properties.configuration.Dapr.appId” is the application identifier that will be used on service-to-service invocation calls. There are other configurable settings that weren’t used such as the log level, maximum size of the request body (default is 4MB) and the max size of the http headers (default is 65KB) [48].

The “properties.configuration.Dapr.appId” is the application identifier that will be used on service-to-service invocation calls. There are other configurable settings that weren’t used such as the log level, maximum size of the request body (default is 4MB) and the max size of the http headers (default is 65KB) [48].

The ingress allows to expose the application to the public internet or not. In this case, the notifications API is an internal service that should only be invoked by the other services on the same environment. For this reason, the “properties.configuration.ingress.external” is set to “false”. On the other hand, the public gateway API would have this setting set to “true”, so it can have a public IP address to receive requests from the outside network. The “properties.configuration.ingress.allowInsecure” indicates whenever the HTTP connections are allowed or should be automatically redirected to HTTPS. The container port used for ingress traffic is defined on the “properties.configuration.ingress.targetPort” [48].

The “properties.configuration.registries” allows to define a collection of container registry credentials that will be used by the Azure Container App to get the container image, and the “properties.configuration.secrets” is a collection of sensitive configuration values, such as passwords, connections string or API keys, that can be referenced in the environment variables. Storing this sensitive information on the Container App secrets helps to protect the application from unauthorized access and data breaches.

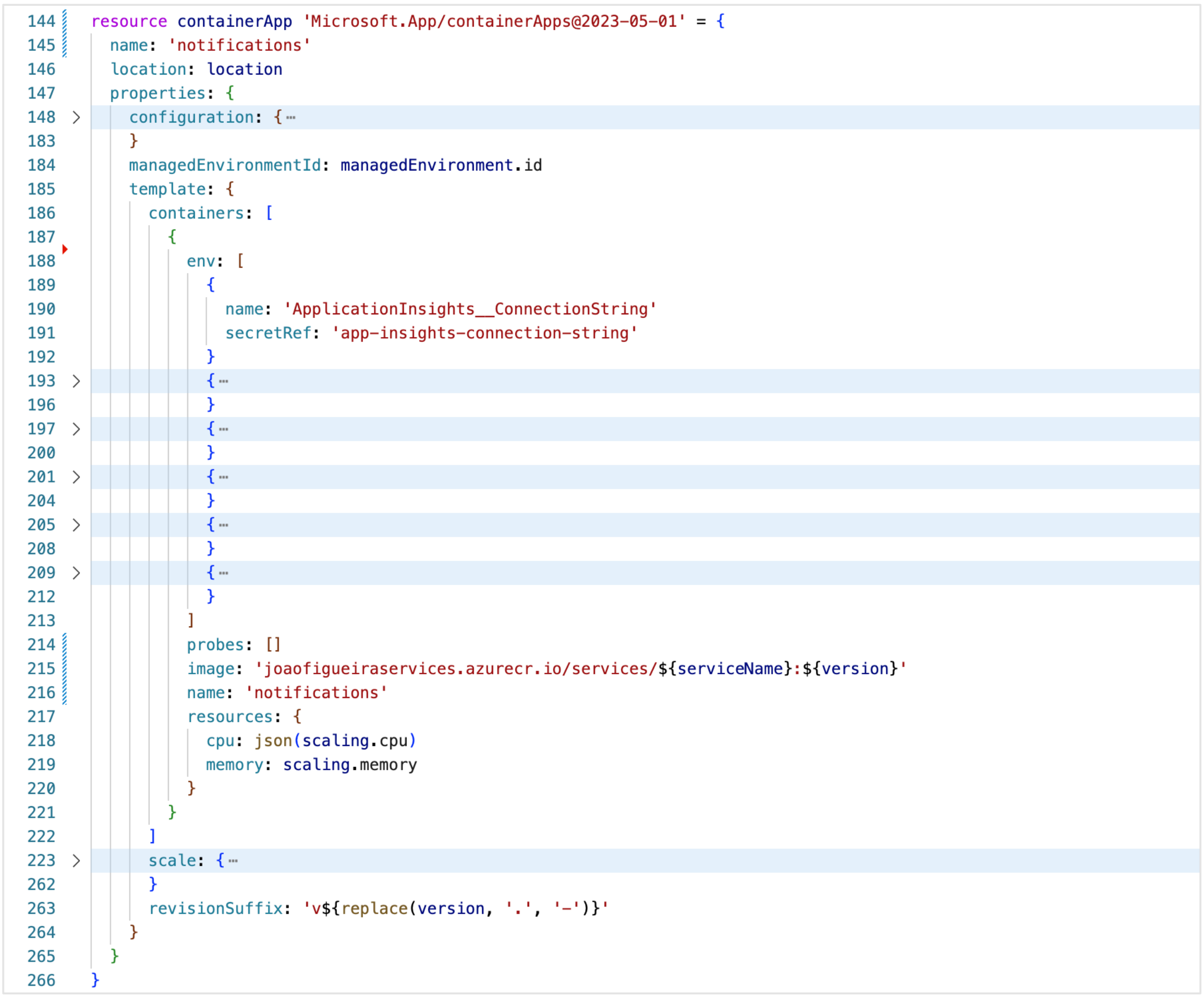
The Azure Container App Environment resource is defined on the “Infrastructure” deployment since it is used by several services. Because at least one of the container apps will need to receive external traffic, the “properties.vnetConfiguration.internal” must be set to “false”, as shown in the Figure 43.



```
resource managedEnvironment 'Microsoft.App/managedEnvironments@2022-11-01-preview' = {
  name: 'ca-env'
  location: location
  properties: {
    appLogsConfiguration: {
      destination: null
    }
    vnetConfiguration: {
      internal: false
      infrastructureSubnetId: caeSubnet.id
    }
  }
}
```

Figure 43 - Bicep template to deploy the Azure Container App Environment

The second part of the resource's bicep template (Figure 44) includes configurations on the container level, such as its name and the image (that was pushed to the repository on the build pipeline). The "properties.template.containers.resources" are the resource power required, and it's possible to set the CPU (in cores) and the required memory. These values are configured on the parameters file and may be different for each environment. In this specific case, on the development environment the container was configured to have 0.25 CPU and 0.5Gi of memory.

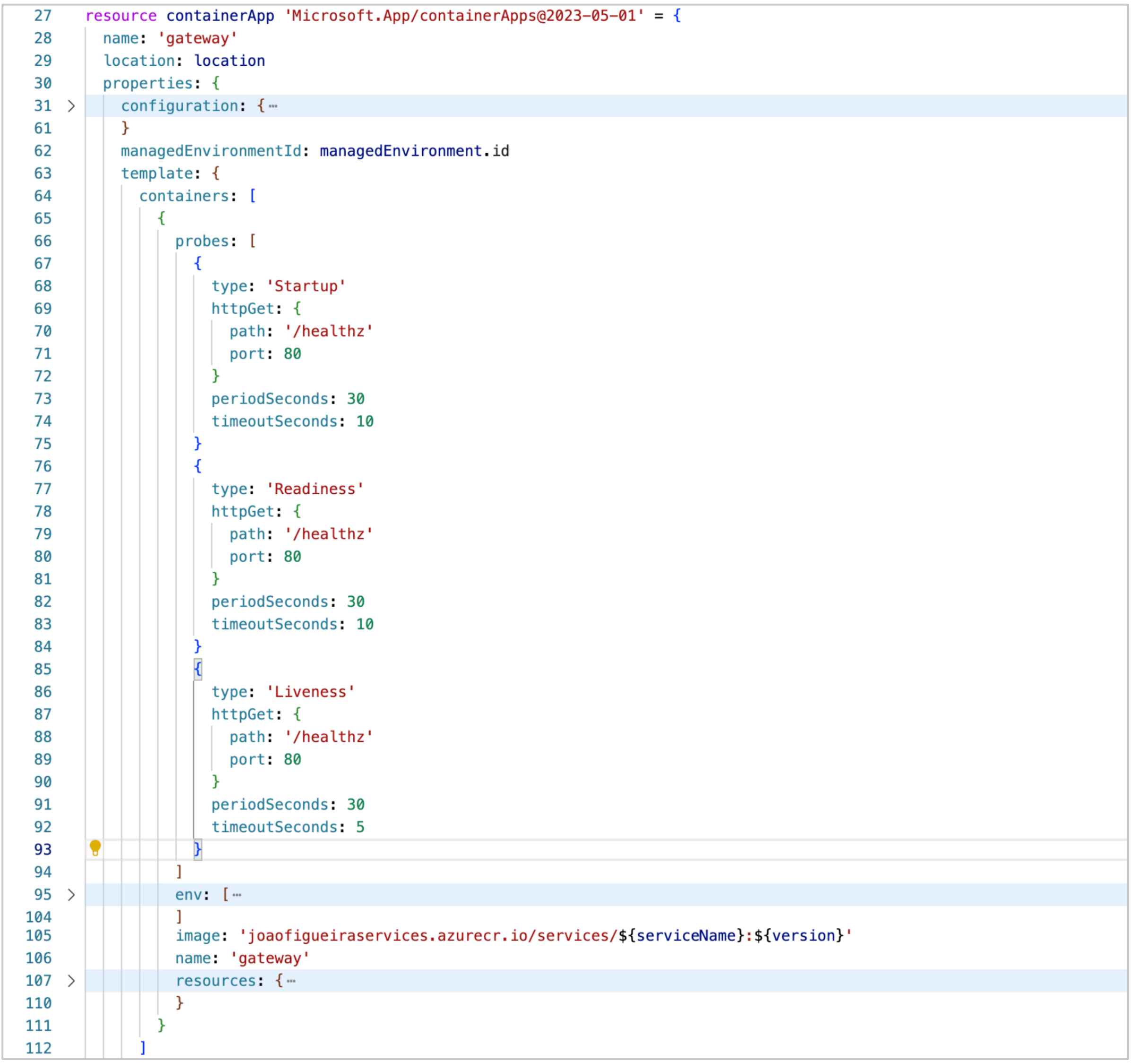


```
resource containerApp 'Microsoft.App/containerApps@2023-05-01' = {
  name: 'notifications'
  location: location
  properties: {
    configuration: {⋯
    }
    managedEnvironmentId: managedEnvironment.id
    template: {
      containers: [
        {
          env: [
            {
              name: 'ApplicationInsights__ConnectionString'
              secretRef: 'app-insights-connection-string'
            }
            {⋯
            }
            {⋯
            }
            {⋯
            }
            {⋯
            }
            {⋯
            }
          ]
          probes: []
          image: 'joaofigueiraservices.azurecr.io/services/${serviceName}:${version}'
          name: 'notifications'
          resources: {
            cpu: json(scaling.cpu)
            memory: scaling.memory
          }
        }
      ]
      scale: {⋯
      }
      revisionSuffix: 'v${replace(version, '.', '-')}'
    }
  }
}
```

Figure 44 - Bicep template to deploy the Notifications API (PART II)

Azure Container Apps also provide support for probes, that allows to regularly inspect the status of the container. These probes are based on Kubernetes health probes, which are divided into 3 different types: startup, readiness, and liveness. The startup probes checks if the application has successfully started and disables other probes until it succeeds so that those probes don't interfere with the application startup. After startup, the readiness probes are used to know when the application is ready to accept incoming requests. When the application is up and running, the liveness probes are used to periodically check if the container is still responsive, or if it needs to be restarted [49].

Unfortunately, as the reader can see in previous figure, there is no probes defined on the notifications API. Although Kubernetes already supports gRPC health probes [50], the Azure Container App only supports setting probes using HTTP(S) and TCP, which is a significant drawback. The Figure 45 shows the public gateway API probes, that in this case (being a REST API) uses HTTP.

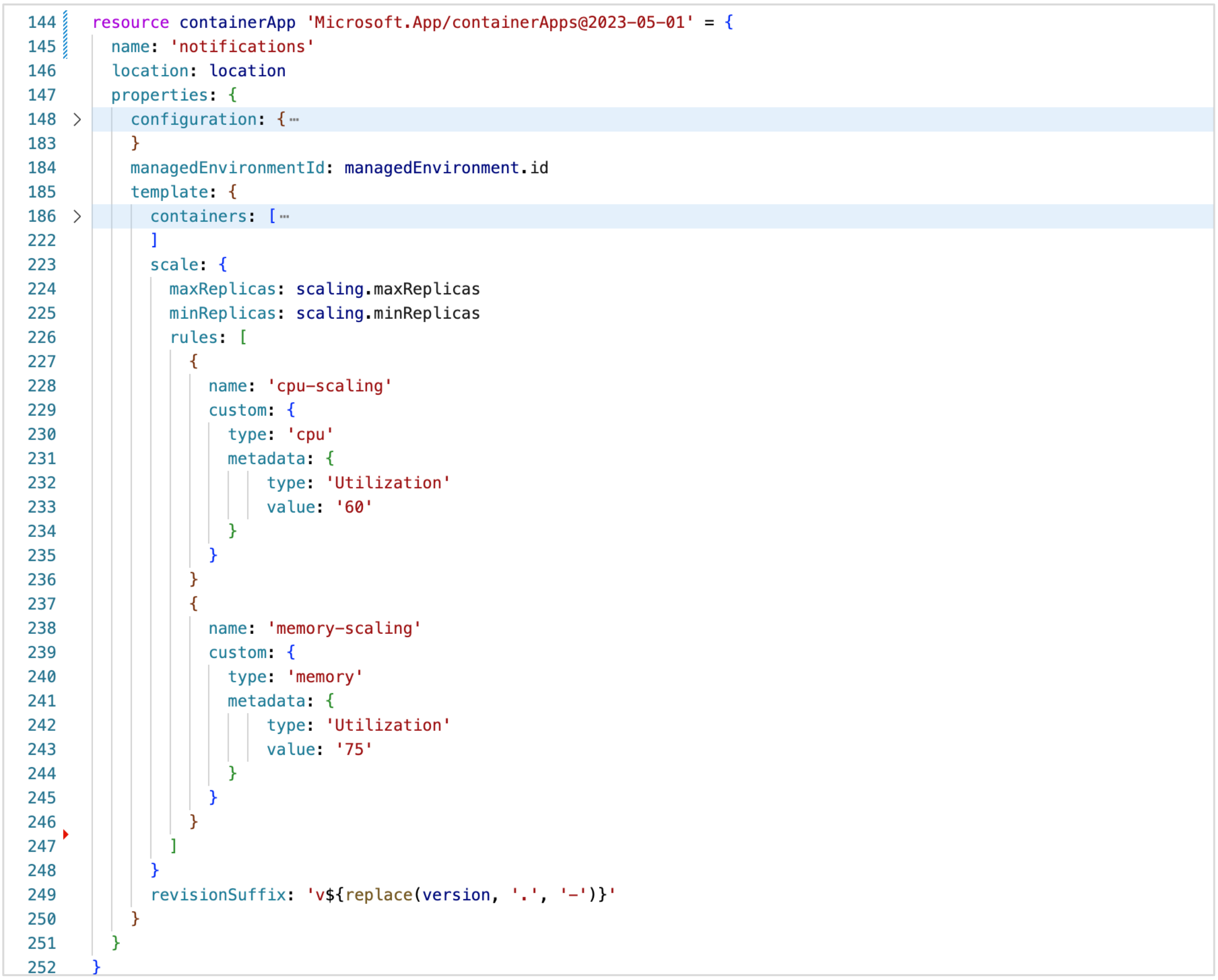


```
resource containerApp 'Microsoft.App/containerApps@2023-05-01' = {
  name: 'gateway'
  location: location
  properties: {
    configuration: {⋯
    }
    managedEnvironmentId: managedEnvironment.id
    template: {
      containers: [
        {
          probes: [
            {
              type: 'Startup'
              httpGet: {
                path: '/healthz'
                port: 80
              }
              periodSeconds: 30
              timeoutSeconds: 10
            }
            {
              type: 'Readiness'
              httpGet: {
                path: '/healthz'
                port: 80
              }
              periodSeconds: 30
              timeoutSeconds: 10
            }
            {
              type: 'Liveness'
              httpGet: {
                path: '/healthz'
                port: 80
              }
              periodSeconds: 30
              timeoutSeconds: 5
            }
          ]
          env: [⋯
          ]
          image: 'joaofigueiraservices.azurecr.io/services/${serviceName}:${version}'
          name: 'gateway'
          resources: {⋯
          }
        }
      ]
```

Figure 45 - Gateway API probes

The “properties.template.containers.probes.periodSeconds” defines how often the probe will be checked and the “properties.template.containers.probes.timeoutSeconds” defines the time that Kubernetes should wait for the probe reply before it times out. In addition to these options, it’s also possible to configure a success threshold (number of successful probe executions before marking the container as healthy again), a failure threshold (number of failed probe executions before marking the container as unhealthy) and an initial delay (before performing the first probe).

Figure 46 is the third and last part of the bicep template used to deploy the Notifications API. In terms of scaling, the Azure Container Apps provides multiple options. Kubernetes offers out-of-the box horizontal scaling based on CPU and memory metrics. Scale horizontally means that the Azure Container App will add (or remove) replicas (or pods, in the Kubernetes language), to perform at a certain level and match the demand. In the next figure we can see that the template allows to define the range (minimum and maximum) of replicas that the container app is allowed to use. This is useful to control the billing, since the more replicas are running, the more expensive it is. It’s also important to notice that, just like the “resource power”, the minimum and maximum replicas are also configurable parameters for each environment.



```
resource containerApp 'Microsoft.App/containerApps@2023-05-01' = {
  name: 'notifications'
  location: location
  properties: {
    configuration: {⋯
    }
    managedEnvironmentId: managedEnvironment.id
    template: {
      containers: [⋯
      ]
      scale: {
        maxReplicas: scaling.maxReplicas
        minReplicas: scaling.minReplicas
        rules: [
          {
            name: 'cpu-scaling'
            custom: {
              type: 'cpu'
              metadata: {
                type: 'Utilization'
                value: '60'
              }
            }
          }
          {
            name: 'memory-scaling'
            custom: {
              type: 'memory'
              metadata: {
                type: 'Utilization'
                value: '75'
              }
            }
          }
        ]
      }
      revisionSuffix: 'v${replace(version, '.', '-')}'
    }
  }
}
```

Figure 46 - Bicep template to deploy the Notifications API (PART III)

An Azure Container App will automatically scale on concurrent http requests. By default, it will add one replica to the cluster when a single revision is handling more than 10 concurrent requests, until it reaches the maximum number of replicas defined. From the figure above, it's possible to see that in this case the author has defined two more rules for scaling, based on CPU and memory metrics. The first is the CPU utilization and is set to 60. This means that, when the average usage of CPU across all replicas reaches 60% or more of the provided capacity for a defined period, it will scale out the application. The Equation 1 calculates the average utilization performance of a resource, with n being the total number of replicas running. The summation goes through each replica represented by the i.

$$Utilization = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{Resource\ Power\ Utilization_i}{Resource\ Power\ Requested_i}\right) \times\ 100$$

Equation 1 - Resource utilization percentage for all replicas

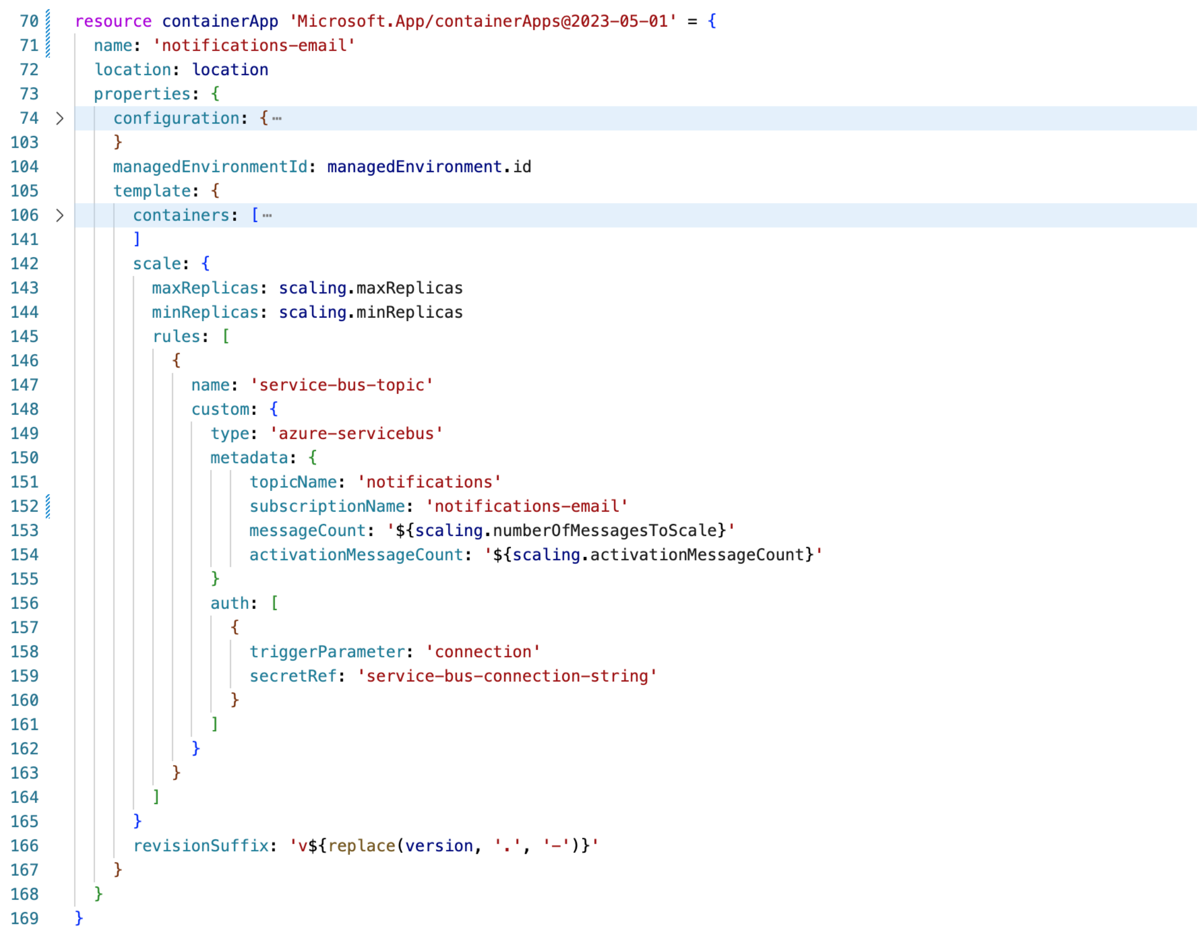
For example, if an Azure Container App is running with 2 replicas, assuming each replica requested 0.5 CPU cores and the first replica is using 0.3 and the second is using 0.2, the current utilization would be calculated by $\frac{1}{2}\times\ (\frac{0.3}{0.5} * 100 + \frac{0.2}{0.5} * 100\ ) = 50\%$. The second metric is the memory utilization, and it uses the same formula.

Instead of using the "Utilization" metric type, it's also possible to use the "AverageValue". The average value is calculated by summing the average current usage across all replicas, as shown on the Equation 2.

$$AverageValue = \frac{1}{n}\sum_{i=1}^{n} Resource\ Power\ Utilization_i$$

Equation 2 - Average utilization on all replicas

In some cases, the CPU and memory metrics may not be the best scalers to use when configuring autoscaling for an application. In fact, using these metrics as the only scaling rules won’t allow the application to scale in to zero [29]. As the author exposed on section 2, KEDA offers a way to scale based on events from various sources. One of these scalers is the Azure Service Bus trigger, that scales based on the number of messages on a queue or topic. On this project, this trigger was used to scale the subscribers that are listening on the notifications topic, such as the email dispatcher. This scaler helps to avoid accumulating messages in the topic if the producer sends messages faster than the subscribers can consume. Such effect would represent an increased delay on the notification delivery, which can be critical in some scenarios. In the Figure 47 it’s possible to see that in this case, the CPU and memory weren’t used as rule, because they are no reliable indicators of the load or demand, and scale them based on these metrics would make no sense.



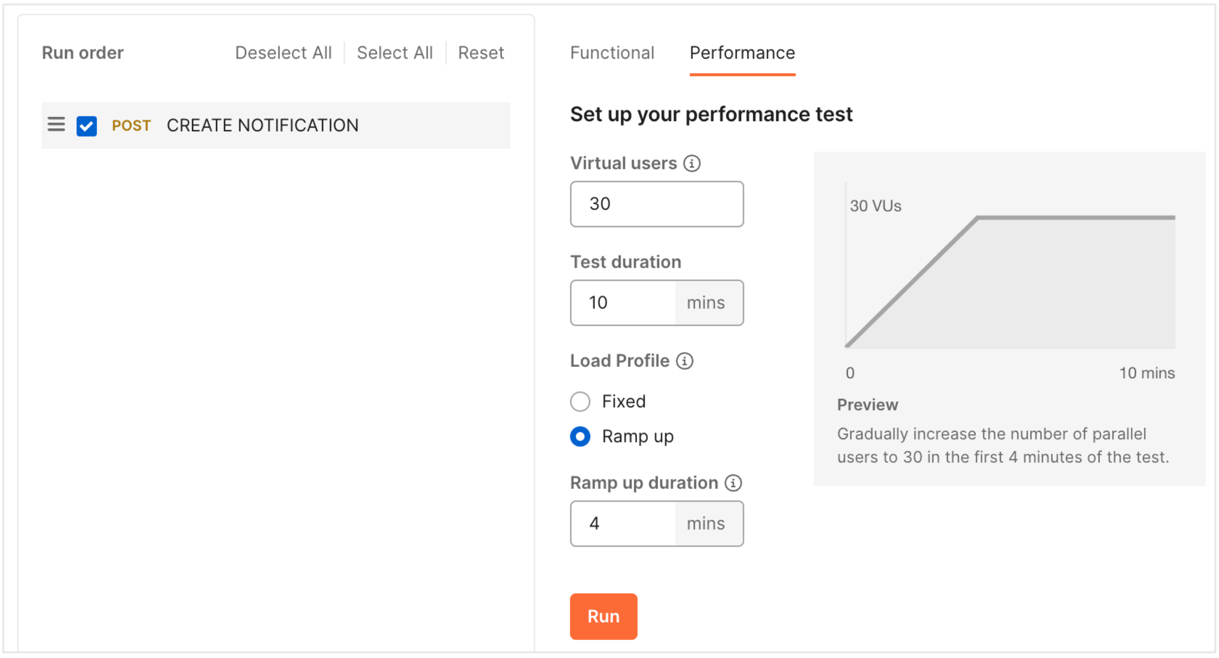
```
resource containerApp 'Microsoft.App/containerApps@2023-05-01' = {
  name: 'notifications-email'
  location: location
  properties: {
    configuration: {⋯
    }
    managedEnvironmentId: managedEnvironment.id
    template: {
      containers: [⋯
      ]
      scale: {
        maxReplicas: scaling.maxReplicas
        minReplicas: scaling.minReplicas
        rules: [
          {
            name: 'service-bus-topic'
            custom: {
              type: 'azure-servicebus'
              metadata: {
                topicName: 'notifications'
                subscriptionName: 'notifications-email'
                messageCount: '${scaling.numberOfMessagesToScale}'
                activationMessageCount: '${scaling.activationMessageCount}'
              }
              auth: [
                {
                  triggerParameter: 'connection'
                  secretRef: 'service-bus-connection-string'
                }
              ]
            }
          }
        ]
      }
      revisionSuffix: 'v${replace(version, '.', '-')}'
    }
  }
}
```

Figure 47 - Bicep template to deploy the Email subscriber

3.7. Load tests and metrics analysis

3.7.1. Overview

In this section, the author uses Azure Application Insights to capture important metrics under a load test to be analysed. A load test is a common exercise used to simulate real-world traffic to measure speed, capacity, scalability, and reliability of software systems. Such tests can help to identify issues such as bottlenecks before they reach production environments and affect possible clients. The Figure 48 shows the load test that was executed using Postman. It simulates multiple users making concurrent requests to create notifications for 10 minutes. The number of virtual users will gradually be increased along the first 4 minutes of test until it reaches 30. Along the 10 minutes, the load test produced 12,420 requests to the gateway API that forwarded them to the notifications API. Considering that there is a direct relation between the number of virtual users and the number of requests per second, this second metric will obviously also increase gradually along the first minutes, causing more pressure on the services between the 4th minute and the 10th.


Run order | Deselect All | Select All | Reset

POST CREATE NOTIFICATION

Functional | Performance

Set up your performance test

Virtual users: 30

Test duration: 10 mins

Load Profile: ( ) Fixed [✓ Ramp up]

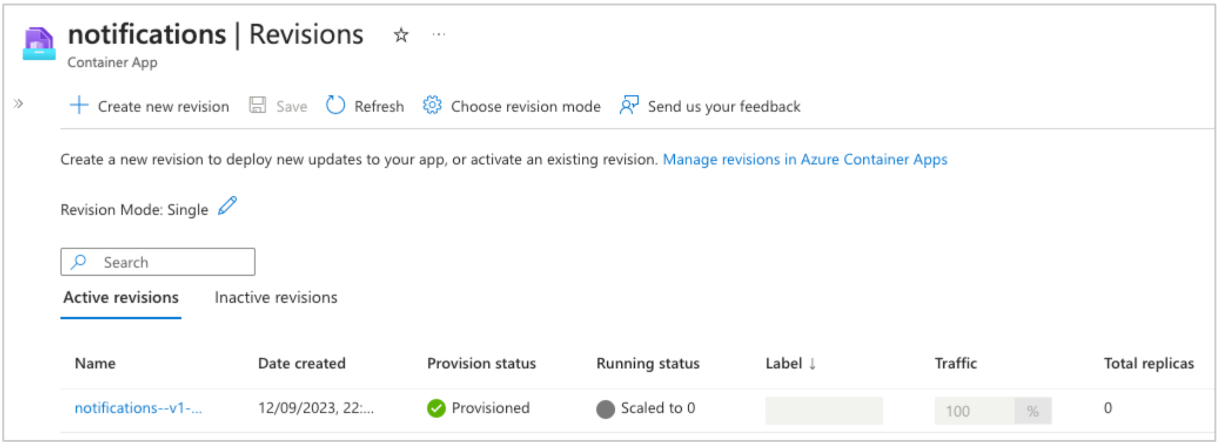
Ramp up duration: 4 mins

Preview

Gradually increase the number of parallel users to 30 in the first 4 minutes of the test.

Figure 48 - Performance test using Postman (v10.16.9)

All the microservices in this system were configured to scale to 0 in the development environment, since most of the time there's no traffic, making no sense to keep a replica running all day long for each service and paying for that. On other scenarios, where the warmup time can be a problem (like production), the trade-off might be worth it. Using the Azure Portal, the author confirmed that the only active revision is correctly scaling to 0, as shown in Figure 49.



Create a new revision to deploy new updates to your app, or activate an existing revision. Manage revisions in Azure Container Apps

Revision Mode: Single

Name	Date created	Provision status	Running status	Label ↓	Traffic	Total replicas
notifications--v1-...	12/09/2023, 22:...	✓ Provisioned	● Scaled to 0		100 %	0

Figure 49 - Notifications API Container App scaled to 0

3.7.2. Scaling analyze

The Figure 50 shows a dashboard with two graphics using the same timeline, that allows the reader to compare and relate the number of incoming requests with the number of replicas deployed on the cluster for each microservice. Before the load test, both services were scaled to 0 since no requests were received for a long time. When the requests started coming in, the gateway and notifications APIs were gradually scaled out. Both were configured to use a maximum of 5 replicas each, and the gateway reached that limit when the request rate reached its peak, while the notifications API handled the incoming traffic with only 3. When the load test ended and the services stopped receiving requests, both applications scaled in back to 0.

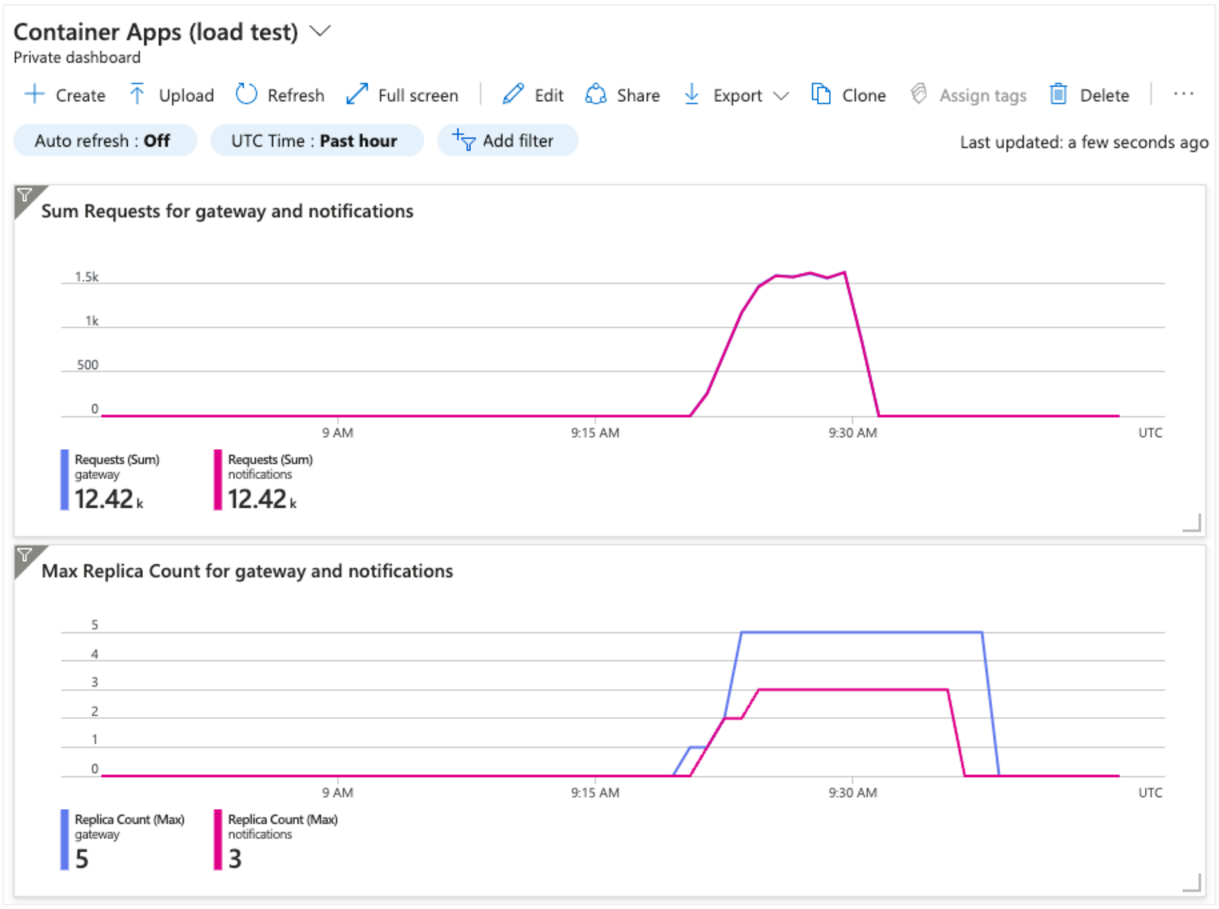
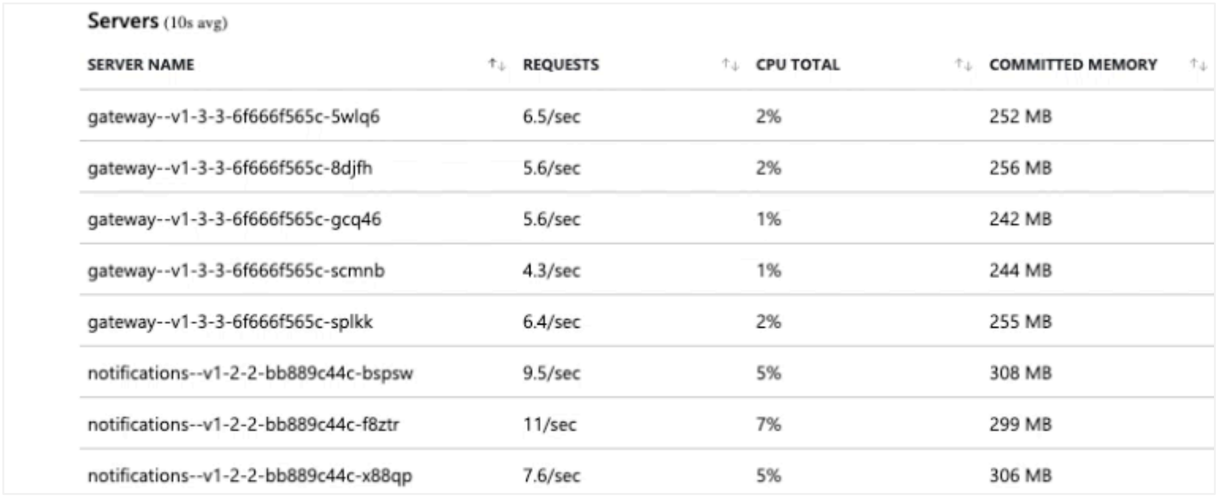


Figure 50 - Relation between incoming requests and number of replicas

3.7.3. Load-balancing analysis

In terms of load balancing, the Azure Application Insights allows to monitor the servers (or replicas, in this case) that are running with the "Live Metrics". The Figure 51 is a screenshot of the replicas running when the incoming requests rate was on its peak. The number of requests per second on each replica of the same service is similar, which means that the load balancing is fairly distributing the workload.



Servers (10s avg)

SERVER NAME	REQUESTS	CPU TOTAL	COMMITTED MEMORY
gateway--v1-3-3-6f666f565c-5wlq6	6.5/sec	2%	252 MB
gateway--v1-3-3-6f666f565c-8djfh	5.6/sec	2%	256 MB
gateway--v1-3-3-6f666f565c-gcq46	5.6/sec	1%	242 MB
gateway--v1-3-3-6f666f565c-scmnb	4.3/sec	1%	244 MB
gateway--v1-3-3-6f666f565c-splkk	6.4/sec	2%	255 MB
notifications--v1-2-2-bb889c44c-bspsw	9.5/sec	5%	308 MB
notifications--v1-2-2-bb889c44c-f8ztr	11/sec	7%	299 MB
notifications--v1-2-2-bb889c44c-x88qp	7.6/sec	5%	306 MB

Figure 51 - Replicas running on a specific instant of the load test

CHAPTER 4

Results Discussion

In this chapter, the author will discuss the results and evaluate his solution, to better understand how the problems that were mentioned in the first chapter were addressed, and how effective his solution is. The author will answer the research questions raised.

How can a software system benefit from the inclusion of a microservices architecture combined with cloud and virtualization techniques such as Virtual Machines and Containers, in its development? In this research the author used a microservices architecture to build his artifact. This decision was based on the fact that microservices are easier to manage when compared to less modularized solutions, such as monoliths. By having individual pipelines for builds and releases of each service, it's very easy to deploy a new version, or rollback when problems that escaped the quality assurance stage are found on production. Running these microservices on containers reduces the startup time, that is extremely important for rapidly scale them horizontally when the workload increases, and the performance starts to degrade. Using Azure Container Apps to run these containers in the cloud the teams can take advantage of many features, such as zero downtime when deploying new versions. It also abstracts the complexity of exposing a service to the public internet, removing the weight of being responsible to manage all the resources needed to do so, such as public IPs and load balancers. Being powered by Kubernetes, Azure Container Apps is a fantastic solution to easily run a microservice without dealing to the complexity of Kubernetes deployments and multiple configurations.

How can a system use self-adaptive techniques to adapt and optimize itself on different workload conditions to be cost effective and keep the desired performance? As previously said, microservices are designed to be scaled horizontally. Again, scale horizontally means adding or removing instances of the same service in order to meet the desired performance and goals on different workload conditions. Manually scale them is not feasible in most of the times, since it would require permanent attention and control on the metrics of each service, in each environment. For this reason, the service should be able to self-adapt itself, considering predefined rules. Using Azure Container Apps (powered by Kubernetes) it's possible to define the minimum and maximum number of instances that the service can provide. It will scale in and out considering the scaling rules defined at the time of the version creation. These rules can be based on multiple inputs, such as CPU and memory metrics, or external triggers using KEDA. When talking about the scaling rules, the built-in integration with KEDA is actually the main advantage of using Azure Container Apps compared with the other ways of running container with Kubernetes orchestration. KEDA adds support to scale to and from zero, that will be an important factor to reduce costs, since when the application is scaled to zero, the Azure Container App it is billed at zero cost. As seen on the load tests of the previous chapter, the application is scaled to a total of zero replicas before the test begins. When it starts, an instance is quickly raised and starts to respond to the requests. As the traffic intensifies, more instances are added to the cluster to keep the performance on an acceptable level, as shown on Figure 50. In the same figure, it's also clear that when the load test ended, the application scaled back to zero, to improve costs. Having in mind that all this scaling was managed by the system itself, the author considers that the objective of self-adaption was successfully accomplished.

How can we abstract the microservices from the underlying infrastructure and deal with the challenges of a distributed system running on the cloud? Azure Container Apps also offers a built-in integration with Dapr. Dapr allows to program in multiple languages and frameworks and use its APIs to abstract the application code from the underlying infrastructure. In this artifact, the author used Dapr to invoke different services in the same cluster. This way, a service doesn't have to know the IP address of another service that it wants to call. In other words, Dapr worked as service discovery and load balancer. Author also used Dapr to publish and subscribe messages from a message broker topic, which in this case was Azure Service Bus. But using Dapr, there's no specific implementations coupled to this message broker, which give the teams the option to easily change the underlying type of resource, without having to change the code. In terms of abstraction from the underlying infrastructure, Dapr is a major ally. However, it stills in evolution and doesn't support all the needs yet. For example, in the artifact produced, there is still a dependency between the code and Azure Cosmos DB, the database used to store the notifications. Since Dapr doesn't not fully support querying data stores and that's a deal breaker.

CHAPTER 5

Conclusions

Cloud computing and distributed microservices introduces many complex concepts and sensible problems. However, the benefits that it offers outcomes the effort needed to build up a solution and making worth the trade-off in some case. It's also important to understand that there's no one-fit-all solution. Each case has its own details that define the best solution.

Azure Container Apps is a very straightforward way of running a container application on the cloud and take advantage of the Kubernetes orchestration platform without dealing with its complexity. The built-in integrations with Dapr, KEDA and Envoy are extremely useful to deal with multiple complex problems such as external ingress, service discovery and load-balancing. However, it has its own limitations. Some situations might require more control and flexibility over the infrastructure that may not yet be supported. For example, the author failed to implement the correct probes on the Notification API, since gRPC probes are not configurable through the Azure Container Apps, despite the fact that is supported by Kubernetes. With other advantages such as the automatic version creation (the revisions) on new changes that offers quick rollbacks and zero downtime on deployments, this type of platform can offer powerful tools to develop distributed systems.

From this research, the author concludes that designing and developing microservices using this framework can produce self-adaptive systems, capable of self-optimization and self-monitoring. Including a microservices architecture, in conjunction with independent pipelines for each service and with different stages for each environment, can significantly increase teams' governance, as well as the confidence to release new versions into production, knowing that the rollback can be achieved with a low effort move. The proposed implementation framework applied in real production case resulted in more frequent releases and a faster and easier development process. The overall system availability, performance and cost were also positively affected.

As said, the proposed technology still presents limitations. As future work, these limitations should be addressed in order to improve the overall solution quality. In particular, the health probes on gRPC communications are extremely important and a workaround should be implemented.

References

[1] J. Venable and R. Baskerville, “Eating our own Cooking: Toward a More Rigorous Design Science of Research Methods,” *The Electronic Journal of Business Research Methods*, 2012. [Online]. Available: https://www.researchgate.net/publication/285741099

[2] J. vom Brocke, A. Hevner, and A. Maedche, “Introduction to Design Science Research,” 2020, pp. 1–13. doi: 10.1007/978-3-030-46781-4_1.

[3] O. Debagy and P. Martinek, “A Comparative Review of Microservices and Monolithic Architectures,” in *IEEE International Symposium on Computational Intelligence and Informatics*, 2018.

[4] L. De Lauretis, “From Monolithic Architecture to MicroservicesArchitecture,” in *IEEE 30th International Symposium on Software Reliability Engineering Workshops*, Institute of Electrical and Electronics Engineers Inc., Oct. 2019, pp. 93–96. doi: 10.1109/ISSREW.2019.00050.

[5] M. Viggiato, R. Terra, H. Rocha, M. T. Valente, and E. Figueiredo, “Microservices in Practice: A Survey Study,” in *VI Workshop on Software Visualization, Evolution and Maintenance*, 2018. [Online]. Available: https://www.researchgate.net/publication/326610698

[6] S. Hassan and R. Bahsoon, “Microservices and Their Design Trade-offs: A Self-Adaptive Roadmap,” in *Proceedings - 2016 IEEE International Conference on Services Computing*, Institute of Electrical and Electronics Engineers Inc., Aug. 2016, pp. 813–818. doi: 10.1109/SCC.2016.113.

[7] P. Siriwardena and K. Indrasiri, *Microservices for the Enterprise: Designing, Developing, and Deploying*. Apress, 2018.

[8] G. Blinowski, A. Ojdowska, and A. Przybylek, “Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation,” *IEEE Access*, vol. 10, pp. 20357–20374, 2022, doi: 10.1109/ACCESS.2022.3152803.

[9] N. Bjorndal, M. Mazzara, N. Dragoni, S. Dustdar, and A. Bucchiarone, “Migration from Monolith to Microservices : Benchmarking a Case Study,” *The Journal of Object Technology*, 2021, doi: 10.13140/RG.2.2.27715.14883.

[10] C. Richardson, *Microservices Patterns*. Manning Publications, 2019.

[11] K. Baylov and A. Dimov, “An Overview of Self-Adaptive Techniques for Microservice Architectures,” in *Serdica Journal of Computing*, 2017, pp. 115–136. [Online]. Available: https://scholar.google.com

[12] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 2nd ed. O’Reilly, 2021.

[13] A. Rashid and A. Chaturvedi, “Virtualization and its Role in Cloud Computing Environment,” *International Journal of Computer Sciences and Engineering*, vol. 7, no. 4, pp. 1131–1136, Apr. 2019, doi: 10.26438/ijcse/v7i4.11311136.

[14] P. Sareen, “Cloud Computing: Types, Architecture, Applications, Concerns, Virtualization and Role of IT Governance in Cloud,” in *International Journal of Advanced Research in Computer Science and Software Engineering*, 2013, p. 2277. [Online]. Available: www.ijarcsse.com

[15] S. Tamane, “A Review on Virtualization: A Cloud Technology,” in *International Journal on Recent and Innovation Trends in Computing and Communication*, International Journal on Recent and Innovation Trends in Computing and Communication, 2015. [Online]. Available: http://www.ijritcc.org

[16] N. Roman Herbst, S. Kounev, and R. Reussner, “Elasticity in Cloud Computing: What It Is, and What It Is Not,” in *10th International Conference on Autonomic Computing*, pp. 23–27.

[17] F. Douglis and O. Krieger, “Virtualization,” *IEEE Internet Comput*, vol. 17, no. 2, 2013.

[18] “Docker Overview.” Accessed: Sep. 23, 2023. [Online]. Available: https://docs.docker.com/get-started/overview/

[19] A. M. Potdar, D. G. Narayan, S. Kengond, and M. M. Mulla, “Performance Evaluation of Docker Container and Virtual Machine,” in *Third International Conference on Computing and Network Communications*, Elsevier B.V., 2020, pp. 1419–1428. doi: 10.1016/j.procs.2020.04.152.

[20] J. O. Kephart and D. M. Chess, “The Vision of Autonomic Computing,” *Computer (Long Beach Calif)*, vol. 36, no. 1, pp. 41–50, 2003, doi: 10.1109/MC.2003.1160055.

[21] N. C. Mendonca, P. Jamshidi, D. Garlan, and C. Pahl, “Developing Self-Adaptive Microservice Systems: Challenges and Directions,” *IEEE Softw*, vol. 38, no. 2, pp. 70–79, Mar. 2021, doi: 10.1109/MS.2019.2955937.

[22] D. Weyns, “Engineering self-adaptive software systems - An organized tour,” in *2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems*, Institute of Electrical and Electronics Engineers Inc., Jan. 2018, pp. 1–2. doi: 10.1109/FAS-W.2018.00012.

[23] M. G. Hinchey and R. Sterritt, “Self-Managing Software,” *Computer (Long Beach Calif)*, vol. 39, no. 2, pp. 107–109, Feb. 2006, doi: 10.1109/MC.2006.69.

[24] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, “Rainbow: Architecture-Based Self-Adaptationwith ReusableInfrastructure,” *Computer (Long Beach Calif)*, pp. 46–54, 2004.

[25] S. R. Nadaf and H. K. Krishnappa, “Kubernetes in Microservices,” *International Journal of Advanced Science and Computer Applications*, vol. 2, no. 1, pp. 7–18, Nov. 2022, doi: 10.47679/ijasca.v2i1.19.

[26] “Kubernetes Components.” Accessed: Sep. 23, 2023. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components

[27] N. Poulton, *The Kubernetes Book*. JJNP Consulting Limited, 2023.

[28] “Dapr Overview.” Accessed: Sep. 23, 2023. [Online]. Available: https://docs.dapr.io/concepts/overview/

[29] "KEDA Concepts." Accessed: Sep. 23, 2023. [Online]. Available: https://keda.sh/docs/2.10/concepts/

[30] "Amazon Maintains Lead in the Cloud Market." Accessed: Dec. 27, 2023. [Online]. Available: https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/

[31] A. Giretti, *Beginning gRPC with ASP.NET Core 6: Build Applications using ASP.NET Core Razor Pages, Angular, and Best Practices in .NET 6*, 1st ed. Apress, 2022.

[32] "Azure Cosmos DB." Accessed: Sep. 23, 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/cosmos-db/introduction

[33] "Azure Service Bus." Accessed: Sep. 23, 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview

[34] "Azure Service Bus Queues and Topics." Accessed: Sep. 23, 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-queues-topics-subscriptions

[35] "Azure Functions", Accessed: Sep. 23, 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview

[36] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The Rise of Serverless Computing," *Commun ACM*, vol. 62, no. 12, pp. 44–54, Dec. 2019, doi: 10.1145/3368454.

[37] "Azure SignalR Overview." Accessed: Sep. 23, 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-signalr/signalr-overview

[38] "Azure Notifications Hub." Accessed: Sep. 23, 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/notification-hubs/notification-hubs-push-notification-overview

[39] "Azure Container Apps overview." Accessed: Sep. 23, 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/container-apps/overview

[40] "Containers on Azure Container Apps." Accessed: Sep. 23, 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/container-apps/containers

[41] "Revisions in Azure Container Apps." Accessed: Sep. 23, 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/container-apps/revisions

[42] "Dapr Service-to-Service invocation overview." Accessed: Sep. 23, 2023. [Online]. Available: https://docs.dapr.io/developing-applications/building-blocks/service-invocation/service-invocation-overview

[43] "Network proxying in Azure Container Apps." Accessed: Sep. 23, 2023. [Online]. Available: . https://learn.microsoft.com/en-us/azure/container-apps/network-proxy

[44] "Azure DevOps." Accessed: Sep. 23, 2023. [Online]. Available: https://azure.microsoft.com/en-us/products/devops/#features

[45] "Infrastructure as Code (IBM)." Accessed: Sep. 23, 2023. [Online]. Available: https://www.ibm.com/topics/infrastructure-as-code

[46] "Infrastructure as Code (Microsoft)." Accessed: Sep. 23, 2023. [Online]. Available: https://learn.microsoft.com/en-us/devops/deliver/what-is-infrastructure-as-code

[47] "Azure Resource Manager." Accessed: Sep. 23, 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-resource-manager/templates/overview

[48] "Bicep." Accessed: Sep. 23, 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/templates/microsoft.app/containerapps?pivots=deployment-language-bicep

[49] "Health Probes on Azure Container Apps." Accessed: Sep. 23, 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/container-apps/health-probes?tabs=arm-template

[50] "Liveness probes with gRPC." Accessed: Sep. 23, 2023. [Online]. Available: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/#define-a-grpc-liveness-probe