Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 232 (2024) 264–273

Procedia Computer Science

www.elsevier.com/locate/procedia

5th International Conference on Industry 4.0 and Smart Manufacturing

Developing self-adaptive microservices

João Figueira[a,*], Carlos Coutinho[b]

[a]Instituto Universitário de Lisboa (ISCTE-IUL), Lisboa, Portugal
[b]Instituto Universitário de Lisboa (ISCTE-IUL), ISTAR, Lisboa, Portugal

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; Self-adaptive systems; Kubernetes; Azure Container Apps;

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. Multiple people working in the same code base is the first big difference that a recent graduate finds on his first touch with a real project, as the number of persons involved is higher, and the tasks should be done in a parallel way, to deliver faster. This leads to problems like merge conflicts and dependencies between tasks from different developers. Of course, there are ways to solve these, but it's complicated and time-consuming. These problems can potentially be amplified by the architecture of the software. If it is composed all in one piece and all the functionalities and components are tightly coupled, splitting work across different teams and individuals becomes a really difficult task. In addition, software companies that make use of a continuous delivery approach normally have multiple stages before dropping new code into production, to try to

10.1016/j.procs.2024.01.026

protect their customers from errors and unexpected behaviors. Considering that it is usual to have a dedicated environment to develop and test new features before rolling them out into production, if the code must be deployed as a whole, it will also represent a high-risk operation, because it's more probably to contain some unexpected effects that weren't spotted by the quality assurance team.

Another important point is that the workload in a production environment is completely different from the workload in a test or development environment. Even in production there are substantial differences between regions and the workload is very dynamic throughout the day. When the software users are more active, the workload grows, and when they're sleeping, the workload is relatively insignificant. Meaning that if the system keeps the same power of resources all day long, it won't take full advantage of it and the company will be overpaying.

All these problems are some of the reasons that led to a change in the old architectural style and techniques used by many systems. Cost-effectiveness, scalability, resilience, and ease of deployment are now (more than ever) key factors to consider when designing a software piece. Having this in mind, this article aims to answer the following research questions:

- 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?

2. Literature Review

2.1. Software Architectures

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 software evolves, as well as its 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 analyzed and sometimes tested before committing to one.

Monolithic architecture (MA) has been for many years the main choice of developers for modeling 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, using a single code base for all [2, 32].

At the time of this writing, the Microservices architecture (MSA) has become the most popular way of developing applications. Large companies like Netflix, Amazon, eBay, and many others have migrated their software systems from the old patterns (MA) to this architecture type [2]. In an MSA, the key idea involves isolating business functionalities into microservices that interact through standardized interfaces [1] based on lightweight mechanisms such as HTTP or gRPC. Each microservice should be designed, developed, deployed, and administrated independently [30].

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 [28]. Multiple studies have shown that monolithic systems normally outperform the microservices architecture on small and medium systems [2, 28, 29]. 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 [33]. On the other hand, microservices are emerging due to multiple reasons. In [2], 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 [6].

- 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 [33].
- 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 once the rollback can be achieved 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 [3], 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 removing 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 [33]. Beside of that, microservices also increase the complexity of the service governance, including monitoring, testing, discovery, and others [30].

2.2. Cloud computing

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 local machines and hardware [10]. In fact, cloud computing clients can take advantage of the pay-as-use model, which is a payment model where you pay for what you use which fits your needs and allows you to scale gradually, quickly, and as needed. A cloud client pays to use a small set of resources from a shared pool, and it's billed considering its utilization. A cloud can be defined as a pool of virtualized and configurable computing resources, including hardware, storage, networks, interfaces, and services [10, 11, 12]. 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" [13], avoiding under and over-provisioning with real-time monitoring. All of these are translated in a cost-effective system, more reliable and highly scalable.

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 centers).
- 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.

2.3. Virtualization, virtual machines, and containers

It's the backbone technique of cloud computing [12]. Virtualization allows the system to run as an isolated system, but in a shared environment [9]. It creates an abstract layer of system resources, decoupling the software from the hardware. The three main characteristics of virtualization are [10, 11]:

- Partitioning - the available physical system resources are shared by portioning it into multiple parts so they 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.

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. Fig. 1a represents a diagram of a traditional host architecture, with 3 different layers:

- 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…)

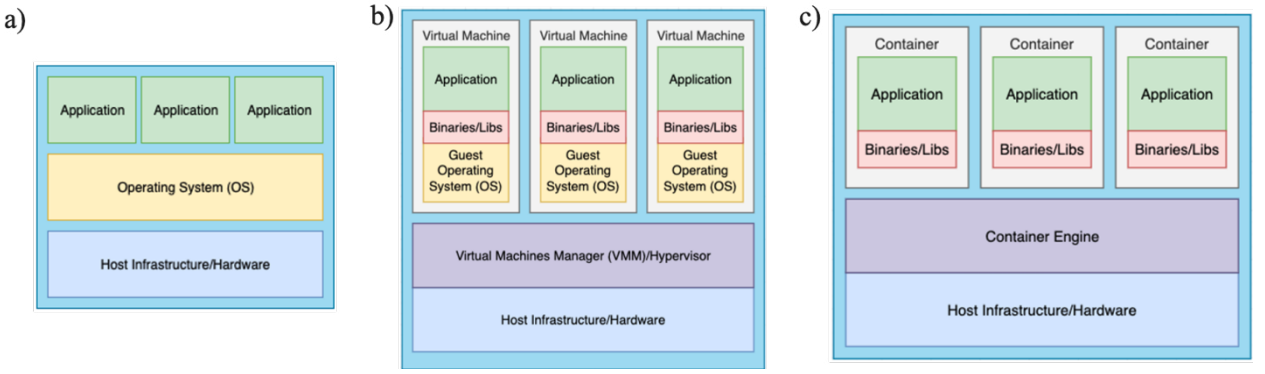

Figure 1 – (a) Traditional host architecture; (b) Virtual Machine architecture running on a host machine (Type I); (c) Container architecture.

However, most of the time the programs and applications running on a host machine don't take full 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" [12] – Fig. 1b. 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, and 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 it from potential treats as viruses and attacks.

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 – Fig. 1c. Docker is one of the most well-known and widely used platforms for developing, shipping, and running container applications with fastest and consistent deliveries [17].

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, VMs are considered 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 [14], 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. As expected, the docker container performed better in every test that was performed, confirming that the resource usage is more effective using this technique.

2.4. Self-adaptive microservices

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 [5]. 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 [4] the authors described self-adaptive systems as systems that are constantly monitoring their behavior to modify themselves at runtime to preserve or enhance their quality attributes. There are four types of self-management objectives [5, 6, 7]:

- 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 - the ability to autonomously recover from failures and even predict them.
- Self-protecting - against malicious attacks or cascading failures.

In [5], Danny Weyns suggests a conceptual model of a self-adaptive system based on four main actors: "Managed System", "Managing System" "Environment" and "Adaptation Goals" as shown on the Fig. 2a. 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.

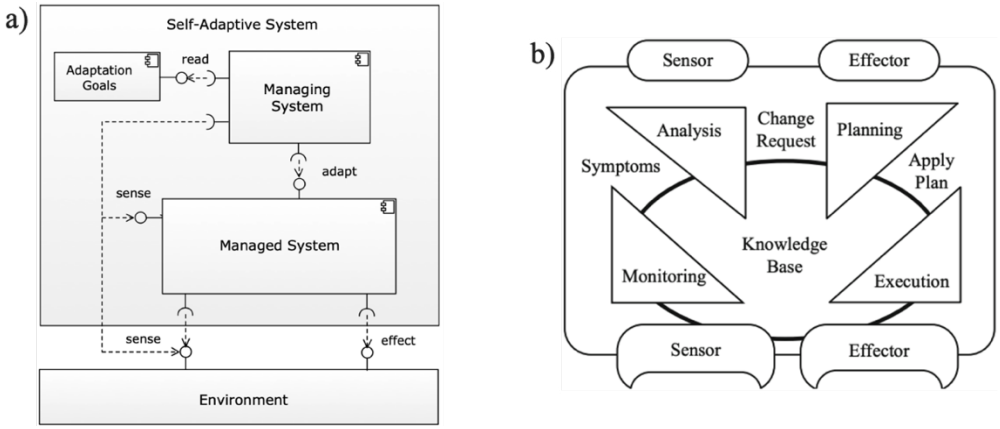

Figure 2 - (a) Conceptual model of a self-adaptive system [5]; (b) The MAPE-K loop [8]

To make runtime changes on the system, the managing element usually has a control loop that consists in four different stages: Monitor, Analyze, Plan, and Execute. These elements share common Knowledge, and this is called MAPE-K [4, 5], represented on the Fig. 2b. 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 [5, 6].

2.5. Kubernetes, Azure Container Apps, KEDA and DAPR

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 can detect unhealthy containers, restarting or replacing them with new ones and detect host failures [15]. 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" [16] 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. These background processes are constantly running and trying to adjust the cluster to match as far as possible the desired state (e.g., the minimum number of pods). For instance, if an application needs to be scaled out to respect some configuration rule, more Pods would be added to the nodes. This is called "horizontal scaling" and consists in adding more instances instead of switching to more powerful resources ("vertical scaling").

On the other hand, the worker nodes are where user applications run. The "kubelet", an agent running on each node, is responsible for reporting back the overall status of the node to the control plane, receives instructions on what actions to take and keeps monitoring the containers within the Pods, ensuring they are running and healthy [16, 27].

However, in [15] 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.

Azure Container Apps (ACA) provides an abstraction over Kubernetes. It's one of the many ways that Azure offers to run microservices and containerized applications on a serverless platform. The main difference when compared to Azure Kubernetes Service (AKS) it's that ACA doesn't provide direct access to the Kubernetes API, meaning that it's less flexible. However, ACA it's easier to use, 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, and each revision may have many replicas (or instances). Each replica may 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, it can also scale to 0 replicas, which is an important feature since it won't be billed usage charges. ACA provides built-in integration with DAPR and KEDA, which allows to easily communicate with other services and to scale in and out (horizontally) based on external triggers. When DAPR is enabled, the ACA launches a secondary process alongside the application, often called as "DAPR's sidecar" that will locally communicate with the main container.

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) [23]. 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 [23].

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 [18].

At this moment, Dapr offers 9 different types of building blocks. The architecture suggested in this paper will use 2 of these components in its implementation: the "service-to-service invocation" that allows 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; and the "Publish and subscribe" that allows 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.

3. Proposed architecture for development of self-adaptive microservices

In this chapter the author will propose an implementation for a real use case scenario. From the above research, the author proposes to use Kubernetes through the Azure Container Apps for developing self-adaptive microservices. Figure 3 presents a first diagram of the author's solution, and it shows the high-level architecture of a system composed by several microservices. Each microservice 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 services. 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 [34]. However, it still has low compatibility with 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. Fig. 3 shows four microservices, but the notifications service is the focus of this project and is the one that the author will propose an architecture for, assuming that the others are already implemented. Using ACA and Dapr, the system can take advantage of the service-to-service invocation building block and call the other services by their unique names, working as service discovery as long as they're in the same Azure Container App Environment.

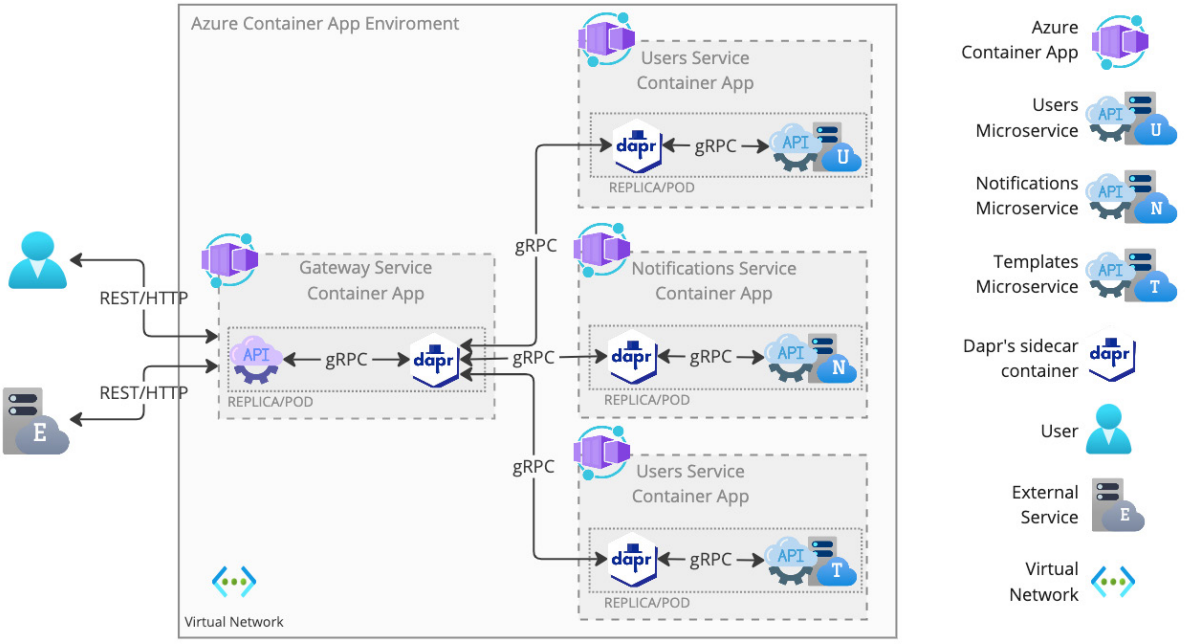

Figure 3 – High-level diagram of the system

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. The service runs on a container mounted with docker and it's orchestrated by Kubernetes abstracted by an ACA. The notifications are stored on an Azure Cosmos DB collection, which 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 [19]. After saving a notification on the database, that notification is added to an Azure Service Bus topic. Azure Service Bus is a message broker with message queues and publish-subscribe topics [20]. A message broker allows applications to exchange messages in an asynchronous and decoupled way, keeping the sender and the receiver anonymous [31]. 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 [24]. When a message is added to a service bus topic, all the subscribers are notified. To keep the service as much agnostic from the platform as possible, the service doesn't publish the message directly on the Azure Service Bus but uses the Dapr sidecar by calling the publish API. The sidecar will forward the request to the Dapr publish/subscribe building block API, that will then use the desired (previously configured) publish/subscribe component that encapsulates a specific message broker (that in this case is the Azure Service Bus). In addition, 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. Beside of that, 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, which have a built-in trigger to respond to messages from a pre-defined Service Bus topic. Azure Functions are a serverless Functions-as-a-Service (FaaS) solution that allows to implement event-driven logic, called "functions" that will run for a limited amount of time. These functions can be triggered by specific events such as HTTP requests, timers, service bus events, and others [21]. 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 [22]. In the Fig. 4, four different types of channels are represented, and for each one it's proposed a different technology to be used by the subscriber to send it to a specific platform. Azure SignalR 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 [25]. 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 [26]. 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.

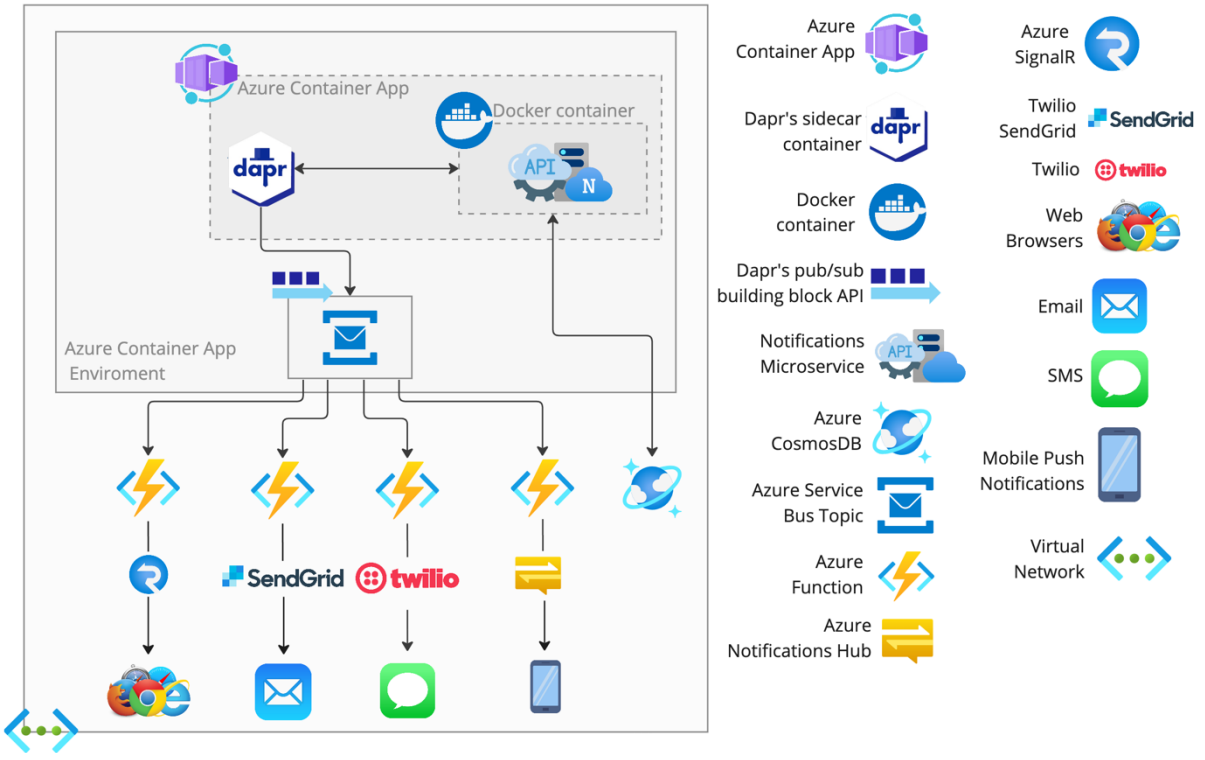

Figure 4 - High-level diagram of the notifications microservice

4. Conclusions

Microservices are easier to manage when compared to less modularized solutions, such as monoliths. Running these microservices on containers reduces the startup time, that is extremely important for quickly 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. The built-in integration with KEDA allows the microservices to self-adapt based on multiple different external triggers, including scaling to zero, that is an important feature to reduce costs. Using Dapr, developers can abstract the application code from the underlying infrastructure by programming against its building blocks. Dapr also provides functionalities such as service discovery and load balancing when using the service-to-service invocation.

Acknowledgements

This work was supported by Fundação para a Ciência e a Tecnologia, I.P. (FCT) [ISTAR Projects: UIDB/04466/2020 and UIDP/04466/2020].

References

[1] S. Hassan and R. Bahsoon (2016). "Microservices and Their Design Trade-Offs: A Self-Adaptive Roadmap," 2016 IEEE International Conference on Services Computing (SCC), San Francisco, CA, 2016, pp. 813-818, doi: 10.1109/SCC.2016.113.

[2] O. Al-Debagy and P. Martinek (2018). "A Comparative Review of Microservices and Monolithic Architectures," 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI), Budapest, Hungary, 2018, pp. 000149-000154, doi: 10.1109/CINTI.2018.8928192.

[3] Sam Newman (2021). “Building Microservices: Designing fine-grained systems.”. O'Reilly Media, Inc.

[4] N. C. Mendonça, P. Jamshidi, D. Garlan, and C. Pahl (2021). "Developing Self-Adaptive Microservice Systems: Challenges and Directions," in IEEE Software, doi: 10.1109/MS.2019.2955937

[5] Danny Weyns (2018). “Engineering Self-Adaptive Software Systems – An Organized Tour”. 1-2. 10.1109/FAS-W.2018.00012.

[6] Krasimir Baylov and Aleksandar Dimov (2017). “An overview of self-adaptive techniques for microservices architectures” in Serdica Journal Computing, No 2, 115-136

[7] M. G. Hinchey and R. Sterritt (2006). "Self-managing software," in Computer, vol. 39, no. 2, pp. 107-109, Feb. 2006, doi: 10.1109/MC.2006.69.

[8] J. O. Kephart and D. M. Chess (2003). "The vision of autonomic computing," in Computer, vol. 36, no. 1, pp. 41-50, Jan. 2003, doi: 10.1109/MC.2003.1160055

[9] Fred Douglis and Orran Krieger (2013). “Virtualization” in IEEE Internet Computing, vol 17, Issue 2, March-April 2013, doi 10.1109/MIC.2013.42

[10] Aaqib Rashid and Amit Chaturvedi (2019). “Virtualization and its Role in Cloud Computing Environment” in International Journal of Computer Sciences and Engineering, vol.-7, Issue 4

[11] Pankaj Sareen (2013). “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, Volume 3, Issue 3, March 2013

[12] Sharvari Tamane (2015). “A Review on Virtualization: A Cloud Technology” in International Journal on Recent and Innovation Trends in Computing and Communication, Volume: 3 Issue: 7

[13] Nikolas Roman Herbst, Samuel Kounev, Ralf Reussner (2013) “Elasticity in Cloud Computing: What It Is, and What It Is Not” in International Conference on Autonomic Computing

[14] Amit M Potdara, Narayan D Gb, Shivaraj Kengondc, and Mohammed Moin Mullad (2020). “Performance Evaluation of Docker Container and Virtual Machine” in Third International Conference on Computing and Network Communications (CoCoNet’19)

[15] Sarah R Nadaf and H. K. Krishnappa (2022). “Kubernetes in Microservices” in International Journal of Advanced Science and Computer Applications (2023), 2(1): 7-18

[16] Kubernetes components. https://kubernetes.io/docs/concepts/overview/components [Online; Accessed 12-04-2023]

[17] Docker overview. https://docs.docker.com/get-started/overview [Online; Accessed 12-04-2023]

[18] Dapr overview. https://docs.dapr.io/concepts/overview [Online; Accessed 13-04-2023]

[19] Azure CosmosDB. https://learn.microsoft.com/en-us/azure/cosmos-db/introduction [Online; Accessed 16-04-2023]

[20] Azure Service Bus. https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview [Online; Accessed 12-04-2023]

[21] Azure Functions. https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview [Online; Accessed 20-04-2023]

[22] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski (2019). “The Rise of Serverless Computing” in Communications of the ACM, Vol. 62, No. 12

[23] Keda. https://keda.sh/docs/2.10/concepts/ [Online; Accessed 01-05-2023]

[24] Azure service bus topics and queues. https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-queues-topics-subscriptions [Online; Accessed 01-05-2023]

[25] Azure SignalR. https://learn.microsoft.com/en-us/azure/azure-signalr/signalr-overview [Online; Accessed 03-05-2023]

[26] Azure Notifications Hub. https://learn.microsoft.com/en-us/azure/notification-hubs/notification-hubs-push-notification-overview [Online; Accessed 12-05-2023]

[27] Nigel Poulton (2023). “The Kubernetes Book”. JJNP Consulting Limited.

[28] Grzegorz Blinowski, Anna Ojdowska and Adam Pryzbylek (2022) “Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation”

[29] Nathan Cruz Coulson, Stelios Sotiriadis, and Nik Bessis (2020). “Migration from Monolith to Microservices : Benchmarking a Case Study” in IEEE Internet of things journal, vol. 7, no. 5

[30] Prabath Siriwardena, Kasun Indrasiri (2018). “Microservices for the Enterprise: Designing, Developing, and Deploying”. Apress.

[31] Abhishek Mishra and Ashirwad Satapathi. (2022) “Developing Cloud-Native Solutions with Microsoft Azure and .NET: Build Highly Scalable Solutions for the Enterprise“. Apress.

[32] Lorenzo De Lauretis (2019) “From Monolithic Architecture to Microservices Architecture” in IEEE International Symposium on Software Reliability Engineering Workshops

[33] Chris Richardson (2019) “Microservices Patterns”. Manning Publications.

[34] Anthony Giretti (2022) “Beginning gRPC with ASP.NET Core 6: Build Applications using ASP.NET Core Razor Pages, Angular, and Best Practices in .NET 6, 1st Edition”. Apress.