sustainability

MDPI

Article

The "Socialized Architecture": A Software Engineering Approach for a *New Cloud*

Pedro Malo-Perisé and José Merseguer *

Departamento de Informática e Ingeniería de Sistemas, Universidad de Zaragoza, 50009 Zaragoza, Spain; 736201@unizar.es

* Correspondence: jmerse@unizar.es

Abstract: Today, the *cloud* means a revolution within the Internet revolution. However, an oligopoly sustaining the *cloud* may not be the best solution, since ethical problems such as privacy or even transferring data sovereignty could eventually happen. Our research, coined as the "socialized architecture," presents a novel disruptive approach to completely transform the *cloud* as we know it today. The approach follows ideas already working in the field of volunteer computing, since it tries to socialize spare computing power in the infraused hardware that institutions and normal people own. However, our solution is completely different to current ones, since it does not create hyper-specialized muscles in client machines. The solution is new since it proposes a software engineering approach for developing "socialized services", which, leveraging an asynchronous interaction model, creates a network of lightweight microservices that can be dynamically allocated and replicated through the network. The use of state-of-the-art patterns, such as Command Query Responsibility Segregation, helps to isolate domain events and persistence needs, while an API *Gateway* addresses communication. All previous ideas were tested through a complete and functional *proof of concept*, which is a prototype called *Circle* implementing a social network. *Circle* has been useful to expose problems that need to be addressed. The results of the assessment confirm, in our view, that it is worth to start this new field of work.

Keywords: software architecture; microservice; *cloud*; distributed systems

Citation: Malo-Perisé, P.; Merseguer, J. The "Socialized Architecture": A Software Engineering Approach for a *New Cloud*. *Sustainability* **2022**, *14*, 2020. https://doi.org/10.3390/su14042020

Academic Editor: Tin-Chih Toly Chen

Received: 2 January 2022
Accepted: 6 February 2022
Published: 10 February 2022

1. Introduction

It is well-recognized that we live in a highly digitized world, where computer systems are a nuclear part of our lives, being indispensable to develop a dignified life. It is not difficult to see the benefits they have brought, e.g., improvement in countless processes, economic growth, and wealth. Its flaws are not hidden either: job destruction as a result of automation, the digital divide between gender and generations, or the privacy scandals that flood the media [1,2]. However, in addition to all these important questions, there is another problem: the dependence of these systems of hosting in the cloud. In 2018, 37% of deployments were *on premise*, that is, hosted on the infrastructure of the organization that developed the system. Just two years later, in 2020, 83% of deployments are made on cloud solutions [3]. In two years, the use of the cloud has increased its market share by 10%. At this rate, in 2025 the cloud will be the only option on the market. This progression is caused by good reason: cloud hosting has been a revolution within the internet revolution.

The cloud revolution has brought unprecedented cost reductions, which is the main reason why companies decide to migrate to the cloud [4]. The cloud offers unattainable scalability, replication, and fault tolerance solutions compared to *on-premise* approaches. Such simplicity of use makes it possible to deploy an infrastructure in just a few clicks, which, until a few years ago, would have involved months of work for several systems engineers. However, reality says that around 74% of the cloud hosting market is own by three companies: Amazon, Microsoft, and Google [5]. They rule a market used by 93%

of worldwide companies. The math is clear: of every 100 companies that use the cloud, 68 will depend on data hosted by them. If, as we have said, those 68 systems are nuclear pieces in the lives of millions of people, the power that we are granting to these companies is similar to that of states. Without going any further, 50% of governments already use cloud solutions in their deployments [6].

The fact that such a small group of companies account for such a high proportion of deployments may imply ethical problems [7]. Let us remember that the business model of cloud providers is not based only on offering computing capacity but also on hosting the data from which the applications are fed. Although there are regulatory frameworks that protect users from third-party access to these data, there is no barrier, beyond the legal one, that prevents either an operator or the very same company access without leaving a trace to the content of an infrastructure over which, ultimately, they have full control. Privacy is perhaps the most immediate problem, but we must not forget others: censorship of critical systems or sites, manipulation of stored data, or disruption of functioning for spurious purposes. Ultimately, the cloud may imply to effectively transfer data sovereignty, and the control of the correct and continuous operation of our systems, to large corporations.

Despite this scenario, the fact that these issues are possible does not mean they will necessarily happen. They are just a possibility, and as such should be considered and, ideally, prevented. Obviously, prevention is not reason enough to give up all benefits that the cloud offers. Computer systems are developed, for the most part, by companies that seek to maximize profitability. Social and ethical motivations in the fight against oligopolies will never be a strong argument to give up to them. This is why, if it is intended to reverse this situation and regain control over systems requiring hosting, an option is to struggle for a sustainable profitability—that is, to offer alternatives that can compete, with the "status quo" of the cloud, and endure in terms of cost and convenience. This was precisely the objective of this research. The main contributions of this work are:

- Explores the focal ideas needed to propose a *new cloud* based on hosting "socialized services". Microservices and an asynchronous interaction model are at the core of the solution.
- Offers a proposal for developing "socialized services" using state-of-the-art software engineering patterns, such as CQRS (Command and Query Responsibility Segregation) [8] and an API *Gateway*.
- Develops a prototype, called *Circle*, that proves the feasibility of the ideas presented and exposes the problems to address.

Although it is evident that an alternative to the cloud goes beyond the work of a single team, this article starts by unveiling some of the many problems to address and the many disciplines involved, such as, software engineering, distributed systems, or security. As a modest contribution in this direction, the study offers preliminary ideas, mostly in the software engineering field. Hence, other researchers and companies can later continue by refining and envisioning more proper solutions, which can drive to a real sustainable *new cloud*. On the other hand, the *open innovation* field [9,10] proposes a new innovation model, in which companies commercialize external, and also internal, ideas by deploying outside, and also in-house, pathways to the market. Hence, ideas that originated outside the company can follow different paths for commercialization, such as licensing agreements or startup projects. In this regard, ideas proposed in this work can be the foundation to a new product, i.e., this *new sustainable cloud*, which overcomes the problems of the current one.

In the end, the work presented in this article must be seen as a vision statement and its "proof of concept." In this regard, we propose an architecture that, at an acceptable cost, allows reliable and safe deployments outside the current cloud but without incurring the unsustainable *on-premise* approach, with disadvantages such as high maintenance costs or equipment obsolescence. We coined our solution as the "socialized architecture".

The balance of the article is as follows. Section 2 positions the research of this article and reviews the literature. Section 3 describes the method followed in this research.

Section 4 presents the findings of our research and evaluates them. Section 5 discusses important aspects of this research. Section 6 concludes the article.

2. Problem Statement and Literature Review

The alternative we propose to the massive use of the cloud is to "socialize computing"—that is, the distribution of the computing needs, and eventually the storage, of a system among its users wishing to collaborate with companies that seek actions with a transformative impact or any entity wishing to bet on the reduction in cloud hosting. The process of "socialization" consists of hosting, altruistically or not, one or more parts of the system that you want to support in your own infrastructure. This gives rise to highly distributed environments, executed on a heterogeneous infrastructure.

Far from previous approaches, which are revised in Section 2.1, what we propose is the fragmentation of a multipurpose system (the backend of any modern system) into a series of independent services. Hence, each user willing to participate in the "socialization" process can locally host one or more replicas, of any of the services that make up the system, the "socialized services". These socialized replicas would be peers to others hosted, either on the premises or in the cloud, by the organization that owns the system. Depending on the number of socialized replicas at each moment, the organization could dynamically scale them up. Ideally, with a sufficient mass of users, it could be reduced to zero regarding the dependence from the cloud. The idea of altruistically distributing the computation needs not being new, there are a number of mature technologies that allow to carry out the idea presented. The main techniques, technologies, and methodologies that give support to our proposal are:

- **Virtualization**. Specially lightweight virtualization based on containers, such as Docker [11].
- **Asynchronous communication**. Advances in distributed systems offer interaction patterns that allow for reliable low-latency communications.
- **Microservices**. A design pattern that promotes cohesive and focused services with low coupling among them, as well as desirable scalability properties.
- Other patterns, such as CQRS, that enable separation of concerns in asynchronous environments.

2.1. Literature Review

Volunteer Computing (VC), a.k.a. cycle stealing system or public-resource computing, appeared late in the 1980s. The survey in [12] revises dozens of works in the field of VC, proposed in the last years; hence, the idea of collaborating with a cause, by offering local computing capability, is by no means new. VC is defined as a kind of distributed computing, where anybody with a computer can donate idle computing resources to run computational and storage-intensive tasks [13].

As summarized in [12], a VC system is made of volunteer nodes that donate spare resources (Resource Nodes) and an entity that manages the donated resources and gives a point of entry to volunteers (Resource Controller) and users of the VC system. The behavior of VC systems is summarized in [12] as follows. The user submits the task(s) to the Resource Controller, which after preprocessing selects a suitable or a collection of suitable Resource Node(s) to deploy the task(s); then, the Resource Node(s) processes the task(s) and returns the result(s) back to the Controller. Based on the deployment, the architecture of VC systems can be centralized (client/server, C/S), decentralized (P2P), or hybrid. C/S implies the existence of dedicated machine(s) acting as server(s) and providing resource-controlled services. P2P relies on volunteers acting as resources and controllers; then, communication can be coordinated without a central authority. A hybrid architecture blends the flexibility and scalability of P2P with the security and trust leverage of C/S. However, a centralized server still offers functionalities such as a global resource directory [12].

It is important to note that the "socialized architecture" can perfectly fit, and be implemented, with the three models: C/S, P2P, and hybrid. This is because no restrictions

are made regarding where the infrastructure layer should be placed; it just provides an API, with a minimum number of functionalities. In fact, this layer is a lightweight Resource Controller in charge of solving services addresses, load balancing, information aggregation, and translating the communication protocol. Then, the infrastructure layer acts as an isolated and loosely coupled component, which communicates asynchronously with clients and resources. As a consequence, we can choose an appropriate implementation depending on what features need to be favored, for example, scalability versus security. In *Circle*, the Resource Controller is implemented as an API *Gateway* by leveraging GraphQL [14].

We have not found works in the literature strictly having the same goal of the "socialized architecture", i.e., to propose a complete alternative to the current "status quo" of the cloud. However, in the VC field, and sharing similarities with our work, Kirby et al. [15] and Che and Hou [16] discussed initial models for desktop clouds, then proposing alternatives of possible architectures and discussing some of the challenging aspects to address. From these initial models, several projects have borrowed ideas to implement solutions, most of them in the form of a prototype or proof of concept. **cuCloud** [17] is a project that follows the model in [16]. It proposes a C/S architecture where the clients run guest Virtual Machines (VMs) and install, as it happened in BOINC [18]-related projects, a middleware that controls the node, in this case for monitoring its utilization and QoS. This project was built on **CloudStack** (https://cloudstack.apache.org (accessed on 1 January 2022)). **AdHoc Cloud** [19] is based on BOINC with VirtualBox VMs installed in the volunteer machines. The architecture is C/S, where the server schedules, monitors, and manages the jobs and the overall system, including the VMs. The client installs a middleware that allows to communicate with the server and executes the jobs. **Nebula** [20] uses both dedicated and volunteer nodes. The computation is made within the native client sandbox provided by the Chrome browser; in this way, there is no need of VMs, as it happened in previous approaches, while the security characteristics of Chrome can be used. The system monitors the volunteers, updates them, and assigns tasks while controlling the load balancing. **P2PCS** [21] follows a P2P architecture to get a fully distributed cloud system maintained by an overly network. As in previous approaches, each volunteer node needs to install a middleware, in this case as a daemon that provides an interface to send requests to the system and to communicate with peers. A prototype based on Java has been developed for P2PCS. Finally, the table in Figure 1 summarizes the main characteristics of these desktop clouds and compares them with the "socialized architecture". The column "Sw Eng. Approach" refers whether the approach defines a software engineering approach for developing client applications.

Approach	Architecture	Only volunteer	Virtualization	Middleware in client host	Sw Eng. Approach	Related projects
Socialized architecture	C/S, P2P, hybrid	No	Containers	No	Yes	
cuCloud	C/S	Yes	VMs	Yes	No	BOINC, cloudStack
AdHoc Cloud	C/S	Yes	VMs	Yes	No	BOINC, VirtualBox
Nebula	C/S	No	No	No	No	Chrome
P2PCS	P2P	Yes	No	Yes	No	

Figure 1. Comparison with desktop clouds.

Among projects dedicated to particular cloud features, such as cloud storage, we can mention: **Storage@home** [22], **Fatman** [23], **STACEE** [24], and **SASCloud** [25]. They are volunteer storage cloud projects that propose alternatives to commercial products like Amazon's Simple Storage Service (Amazon S3). **Storage@home** aggregates storage donated by volunteers and provides back-up functionalities in a C/S architecture. **Fatman** uses tens of thousands of underutilized servers to create an archival system, also under a C/S architecture. **STACEE** proposes a four-layers architecture (Backend, Services, Adaptation, and Economic indicators) based on economic metrics, such as energy, that help to minimize energy consumption and maximize user engagement. It leverages a model in which the provision of the resources is accomplished dynamically. **SASCloud** offers a secure storage

service in a mobile ad hoc cloud system. In this case, the architecture is hybrid, which means to have a central authority and clients, which register as nodes. Another important group of projects develop cloud features regarding services, such as social networking in the cloud. Among them, we can mention **SOCIALCLOUD** [26] and **SoCVC** [27]. The first one is a proposal in a paradigm that leverages social networks, like Facebook or Twitter, to build cloud computing services by harnessing trust relationships common in social networks. **SoCVC** (Social Cloud for Volunteer Computing) uses the APIs of different social networks, like Facebook, to identify the users. Moreover, it proposes algorithms to indicate the social reputations of the user's of the system. Finally, other systems of interest in this category are discussed in [28–30].

Although different from our work in objectives and functionalities, there are fully functional projects in the field of VC worthy of being compared with the "socialized architecture". They offer technical solutions that may eventually be of interest, and perhaps reused, in our context. **BOINC** [18] is used for computing intensive tasks for scientists. It is a general-purpose middleware and offers a *client* to be installed in the volunteer machine. BOINC architecture and the scheduling problem for assigning tasks to volunteers are explained in [31], while new task assignment algorithms, which claim to minimize completion time, are given in [32]. BOINC uses traditional technologies, e.g., relational databases for storage, web services for offering functionality, and daemon processes for computing. As we can see the target of the project, architecture and technologies are far from our proposal. **SETI@home** [33], hosted by BOINC, is a project for signal processing in the extraterrestrial environment. Based on a C/S architecture, the clients download and just process the work units and return results to the server. The most salient feature is the redundant computation to detect malicious users. As in the previous case, SETI@home largely differs from our proposal in goal and architecture. **DreamLab** [34] is a mobile app to collectively help in computing for COVID-19 projects. Regarding BOINC and SETI@home, it installs a piece of software, on general-purpose equipment, for it to execute computational-intensive processes. In all three cases, the users who participate do it selflessly. However, in these projects, the local software pieces always act as simple slave executors of a specific, highly specialized task, which is computationally intensive. Different to our proposal, the software pieces are remotely orchestrated, so they are not nuclear pieces of a larger computer system. They are only a hyper-specialized muscle in a task.

Among the most recent proposals in the VC field we have found the following. The work in [35] proposes the integration of VC and vehicular ad-hoc networks (VANET). The idea is to utilize the surplus vehicular computing resources. The work defends the existence of a high amount of available resources in different scenarios: parked vehicles, vehicles at a traffic signal, vehicles in congestion, or smoothly moving vehicles. The goal is not to propose a concrete architecture but the ideas needed for carrying out a master–slave computation leveraging the VANET for VC. In this regard, the work discusses a taxonomy for this new field, the scenarios where to apply it, and the challenges. The applications for this surplus of computing are the usual ones: high-performance computing, autonomous vehicles, intrusion detection, content distribution, connectivity, and efficient communication. In the scope of 5G cellular technologies, Cao et al. [36] proposed a novel user cooperation approach in both computation and communication for mobile edge computing (MEC) systems. The idea is to improve the energy efficiency for latency-constrained computation. The work is based on the premise that wireless devices can offload computation-intensive and latency-critical tasks to access points and cellular base stations in close proximity.

3. Method

The methodology we followed for this research was to initially develop a proof of concept. We thought that a functional prototype would help us to critically assess the concepts, given in the previous section, underpinning the "socialized architecture". For such a disruptive concept, a mere theoretical proposal and discussion might not be

appropriate, due to the scarcity of sources on the matter. In addition, the fact of facing a real development allows,aspects that would otherwise be hidden to emerge.

A complete development of the "socialized architecture" implies to devise many challenging aspects. Among others, we would need to develop new technologies, to address intricate issues regarding security or to develop micro-services for different application domains. Then, we propose to implement a small-scale system prototype powerful enough to consider as many casuistic and features as possible, while keeping a substantial basis for reasoning and obtaining meaningful conclusions regarding our proposals.

We developed *Circle* (https://github.com/Pitazzo/circle (accessed on 1 January 2022)), a social network, similar to Twitter [37], offering a set of cohesive functionalities, that conforms to a complete system for evaluation. Registered users publish short texts receiving likes from other users, and it can be retrieved by more recent or best scored publications, as well as popular users. *Circle* also offers a system of subscriptions and notifications. Table 1 summarizes the main functionalities implemented. Although the domain is seemingly easy, it is rich enough to support a wide range of use cases. In addition, it can be fragmented into sub-domains that can be managed by separate microservices, without demanding cross-communication needs.

Table 1. Requirements for *Circle*.

Users	
FR1	The system allows registration using a nickname and an email.
FR2	The system allows to edit the user profile.
FR3	The system allows to consult other user profiles, by nickname or by querying the n-top with more publications or the n-top with more subscribers.
Contents	
FR4	The system allows to add publications, with title and body.
FR5	The system allows to give likes to publications by others.
FR6	The system allows to consult the n-top more recent publications or the n-top publications with more likes.
Subscriptions	
FR7	The system allows users to subscribe to other users.
Notifications	
FR8	The system notifies subscribed users, by email, when publications are added.
FR9	The system notifies users, by email, when another user is subscribed to his/her publications.
FR10	The system notifies users, by email, when another user gives a like to one of his/her publications.

Circle is used through an API, queried by GraphQL [14], offering services for supporting user requirements, persistence, and all technical infrastructure for the system to be fully operative. Although no user interface was developed, the richness of the system forces to deal with the following aspects to address the needs of experimentation and analysis consistently:

- The need of implementing mechanisms for ensuring integrity while reacting to domain events. For example, when we query the users publishing the most, then the users and contents microservices need to be consistent while each one keeps managing its own entities.
- The need to interact with external services based on internal events. For example, when sending emails to report content published to subscribed services.
- The need of aggregating information when querying different services. For example, when getting the user profile with the list of his/her publications.
- The need of scaling services. It should be easy to add replicas for a given service.

4. Results

This section presents and discusses the proposals currently considered to carry out the concepts behind the "socialized architecture". As previously remarked, our aim was to open a field of work. Hence, these are preliminary ideas, a starting point, for other researchers to fertilize the field. We think that offering an initial body of solutions, although modest, is the best way for researchers to grab and understand the underlying concepts. Consequently, the reader should not take for granted that all the solutions explored here are the best; what we defend is that they work to carry out the "socialized architecture", just to demonstrate the feasibility of the idea.

4.1. An Asynchronous Interaction Model

In today's cloud, control over the infrastructure is complete. Therefore, the operation of the cloud is usually guaranteed, through Service Level Agreements, with figures higher than 99%, obviously due to the high degree of replication. Additionally, these companies invest large amounts in the deployment of submarine cables and optic fiber, then ensuring low latencies in their data-centers. In summary, this first-class infrastructure achieves stability and security in deployments, which is precisely another of the successes of the cloud. However, by relying the "socialized architecture" on a heterogeneous leased infrastructure, made of different hardware solutions, it cannot enjoy the advantages of the cloud. On the contrary, new problems arise:

- **Non-homogeneous latency times**. The same service could be replicated both on a *Raspberry Pi* plugged to a home network or in a high-performance data-center of a university.
- **Unstable network bandwith**. Home networks have different quality of service (QoS) policies than ad hoc networks in data-centers. Furthermore, they are affected by other users' use of bandwidth. For example, a user who hosts a replica of a "socialized service" may decide to consume a streaming video service at the same time.
- **NATs and private IPs**. Hardly any current home network has a static public IP address. Most of them are behind a NAT4 and their associated shared addresses can change at the discretion of the internet service provider. This means that any incoming communication to an instance of a "socialized service" may not be possible, as the other services do not know how to reach such replica.

As a result, we need to recognize that the reliability of the overall "socialized architecture" is necessarily low. These problems led us to rule out synchronous communication protocols, such as REST [38] or RPC, since they require limited latency times, a degree of network stability, and a method to enroute messages. Therefore, we envision, as a principal use case, that interactions, among "socialized service", will be demanded without all the interlocutors being alive synchronously. Undoubtedly, interaction patterns based on asynchronous communications are the natural choice for these scenarios. *Message passing* [39] is a natural choice in this context, and if we combine it with the concept of *message broker* [40] the problem can be controlled:

- Latency times loose importance, since senders continue their workflow until, if necessary, a response arrives.
- The instability of the network becomes a minor problem, as long as the broker remains up. When a message is sent, if one of the possible receiving replicas is disconnected from the network, another will take the message from the queue to complete the communication. If, for example, a network partition left all the replicas without communication with the broker, the reliability mechanisms of the latter would make the message persist locally, until some replica is ready to process it.
- Routing messages to services is no longer a problem, since the broker acts as a centralized agent, being the services who initiate the communication. The broker, hosted at a known static address, is always traceable by the services, regardless if they are behind a NAT. In addition, the public IP-change process becomes completely transparent.

Certainly, the use of asynchronous pattern interactions convert the system into a reactive one. Then, each service can independently act, and it can react to domain events produced in the scope of the system. This model is known as event-driven architectures [41], a category to which the "socialized architecture" belongs. As a consequence, services are completely decoupled from each other. Another advantages of using a *broker* that help in the design of the architecture are:

- Offers delivery policies, timeouts, or centralized security, which are all configured in a common place.
- Reduces an attack's exposition to the broker, so we do not need to secure each service endpoint.
- Provides a centralized method to obtain usage logs and metrics.
- There are many open-source implementations and protocols to run the broker. Among them, RabbitMQ [42] stands out, as it will be discussed in Appendix A.

However, we can also point out some drawbacks:

- The broker becomes a single point of failure: if it stops working the entire system would too. However, this is a well-studied problem, and most implementations offer replication and clustering mechanisms that minimize this risk.
- Running the broker implies an additional infrastructure cost, which is even higher if the degree of replication necessary to mitigate the above risk is applied.

For designing and implementing an asynchronous model, different approaches can be taken. Considering current technologies, we strongly believe that queuing systems are today a plausible approach. In queuing systems, a queue, managed by a *broker*, allows to publish and consume messages, which are simple data structures. Queues allow many different management policies, for example, in *Circle* we used FIFO. For the "socialized architecture", we regard a message as a serialized object that represents either a command, a query, or an event (as required by the CQRS pattern, explained in Section 4.3). Each service will be represented by a queue.

For an asynchronous model to be implemented using queues, several design decisions need to be taken:

- Each service will publish, as messages, its commands, queries, and events. They will be published in a dedicated *exchange*.
- Each message will be replicated as many times as services are subscribed to it. Obviously, each replica will populate the queue representing such a service, and the queue will be attached to the exchange of the publisher service. By doing so, we ensure that different consumers (other services) do not steal messages from each other.
- All replicas representing the same service will be using the same queue for a particular type of command, query, or event. In this way, the different replicas are competing for the same message, but eventually only one achieves it. As a consequence, scaling a service is as simple as to add more replicas to the queue representing such service; then there is no need of load balancing. Hence, we achieve "scalability by design."
- Each service replica owns a queue of where to get the answers for its requests. This is because together with the publication of the query, each replica appends the name of the queue, where the executor service must respond.

Figure 2 represents the queue design for carrying out the proposed asynchronous model.

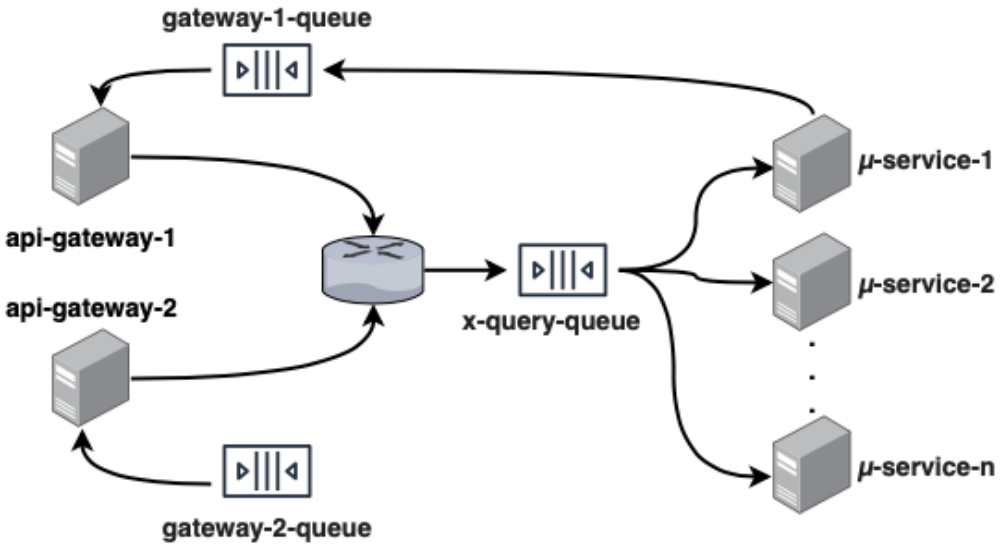

Figure 2. The system of queues.

FR4 of *Circle* is a good example to understand the approach. In a traditional REST [38] (Representational State Transfer) approach, the *contents service* will communicate with the *users service* directly, to increment the number of publications of the author. However, in our reactive proposal, upon sending the publication, the contents service only publishes an event announcing a new *post*. Services interested in such an event are already subscribed and will react as indicated by its business logic. In this way, we can maximize service decoupling, while the latency of direct communication stops being an issue.

4.2. Microservice Architectures

A first step to achieve "socialized services" goes through the "partition of the system" in small software pieces, which eventually will be "containerized" and hosted by third parties. Undoubtedly, the smaller the pieces the best, otherwise the socialization process lose a main incentive: to get a minimum impact hosting a replica in the altruistically leased hardware. Certainly, no one will host a container if it means to sacrifice a good part of his/her resources. Hence, it is vital to "minimize" the replicas, both in size and in its resource demands.

Being mandatory requirements, both the "partition of the system" and the "minimization" of such partitions, the architectural solution is clear: microservices. They are described by Newman [43] as "an approach to distributed systems that promotes the use of small independent services with their own life-cycles, which collaborate jointly." Microservices promote a fragmented development, where the different services communicate among them, while trying to maximize cohesion and minimize coupling. This can be achieved, among others, by reducing the scope of the services to concrete domains and specialized use cases—hence, obtaining small services, as desired in the "socialized architecture".

We understand that reducing the size is not a main goal of the microservices architecture but a consequence of its principles. However, for us, such a reduction is determinant for choosing the pattern. Moreover, microservices come with other advantages for the "socialized architecture":

- **Resilience.** Microservices systems are more fault tolerant; when a service falls, the rest of the system is not affected, only those services relying on it. This is important for the "socialized architecture" since the underlying infrastructure is more prone to failure as previously discussed.
- **Scalability.** Microservices scale better than monolith approaches. When a use case is overloaded it is enough to horizontally scale only the services related to it. In our context, this is desirable since the organization can react more easily before a low number of replicas of a given service.

- **Easier deployment.** The individual deployment of a service is easier than that of a larger piece of software. There are less dependencies, and lighter configurations are also easier. This is important to encourage to users the local installation of replicas.

Microservices also have downsides. First, they are much more complex to develop than monolithic approaches, basically due to the inherent communication processes and the variety of failures they may cause. Second, the testing process is necessarily more complex. Third, if synchronous communication patterns among services are proposed, which is the common way for them, then more latencies are obtained. The latter should not be a problem in our context.

4.3. Approach for Development

Different to approaches discussed in Section 2.1, the "socialized architecture" should not impose a specific programming language, development framework, or whatever restriction is needed for the environment where "socialized services" will be deployed. For example, in SETI@home the tasks are completely dependent on the environment provided for the client application. For the "socialized architecture" to be an alternative, their services should be developed using state-of-the-art software engineering. A first implication of such requirement is the use of virtualization techniques, as it will be discussed in Section 4.4. However, a second implication arises, which is the need to offer design guidelines for the development of the "socialized services".

The essence of the "socialized architecture" relies on distributing the computation needs among home computers, which basically means to distribute the replicas of the services among these computers. Such services, designed as microservices, should offer the system functionalities, i.e., the business logic and the persistence model. However, such design must also take into account the asynchronous model, and the reactive nature of the architecture, as proposed in Section 4.1. A solution we envision for the design of the microservices is the use of the Command Query Responsibility Segregation (CQRS) design pattern [8,44]. In addition, it greatly helps to address an appropriate "separation of concerns," which is an advantage to design well-focused microservices. In the following, we explain the main characteristics of the pattern and analyze its suitability for the "socialized architecture".

CQRS enables a system for two clearly differentiated interaction patterns: commands and queries. *Commands* are actions implying a change on the system state, while no information is returned. Whereas, *queries* are operations that are not changing the system state but returning the user some data. Queries and commands will follow well-differentiated paths in the system, using segregated processing models. On the other hand, CQRS introduces the concept of *bus*. Commands and queries are dispatched using different buses, to which *handlers* are subscribed. Usually, the bus only manages calls to local functions representing the *handlers*, which results in a monolithic approach. Being our architecture based on microservices, this approach is not feasible. Instead, we opted for creating an API *Gateway* to publish commands and queries. The API *Gateway* will also help in implementing the asynchronous model, in the case of *Circle* packaging commands and queries into an AMQP [45] message. Then, services will subscribe to these messages. The function triggered when a message is received fulfills the mission of classical *handlers*. Figure 3 summarizes our idea of using the CQRS pattern.

The API *Gateway* is a common pattern in microservice architectures. In particular, it will be in charge of solving services addresses, load balancing, information aggregation, and translating the communication protocol. Moreover, while it would be reducing the exposition of the system in terms of security, it could also act as an authentication entity. As drawbacks, it represents a single point of failure and requires additional infrastructure for its deployment.

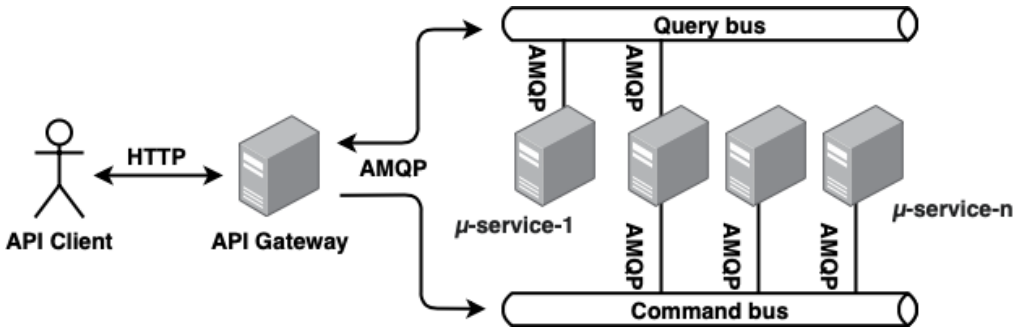

Figure 3. High-level view of the CQRS pattern applied to microservices.

Following CQRS, each service plays as an executor of the logic in its corresponding use case. Moreover, it is of interest for each service not to own the state and be replicable. So, the system could execute different replicas of the service without an additional configuration. For the architecture to address all the restrictions above, each service will also need to accomplish the following well-defined functionalities:

- Implement an API that offers the system functionalities assigned to it. The API is made of commands and queries, according to CQRS.
- Attend the API orders (commands and queries). It will accept or reject them once the parameters have been validated. The order will eventually be executed if applicable.
- Emit domain events as a result of the commands that trigger them.
- Listen to domain events, produced by other services, and react to them.
- Interact with a persistence service whenever a command or query requires it.

For *Circle*, we devised a users service, a contents service, and a subscription service, so to resemble the main system functionalities. Figure 4 depicts an UML components diagram for one of the services in *Circle*. It faithfully reflects our canonical proposal for each service of the architecture. Hence, each service is made of a couple of sub-components:

- An **amqp-controller**, in charge of communicating with the rest of the system components. It reacts to the commands, the queries, and the events it has been subscribed. It also publishes domain events needed by the business logic it implements.
- The **core** of the service, which encapsulates the business logic and also manages the components persistence. Previously, it registers *callbacks* in the **amqp-controller** for being executed when needed.

The advantages of using CQRS are clearly explained in [8]. Fowler advocates CQRS before CRUD systems because they overcome the "anemic" nature of the latter. Additionally, Fowler highlights the performance of CQRS. However, CQRS increases system complexity by forcing the developer to think in a different way. For example, in an *init session* use case, when the system changes state for establishing the session a *command* should carry out the action. Consequently, nothing should be returned by the system longer than the acceptance of the order. However, for the system to inform the user about the result of the command, a subscription mechanism, as the one proposed, is mandatory. Such a mechanism completely fits with our asynchronous model.

Summarizing, the cost of applying CQRS is high. However, it was decided to propose it due to its performance and asynchronous nature. As explained, we need to take into account the uncertainty of the underlying hardware infrastructure. Therefore, we opted for considering the worst-case scenario and then assumed that the performance should be highly optimized. Finally, regarding our case study, it is true that it is not calculation intensive nor critical about obtaining immediate responses, but we implemented it following our architectural proposal, obviously to obtain accurate results and to obtain the corresponding lessons.

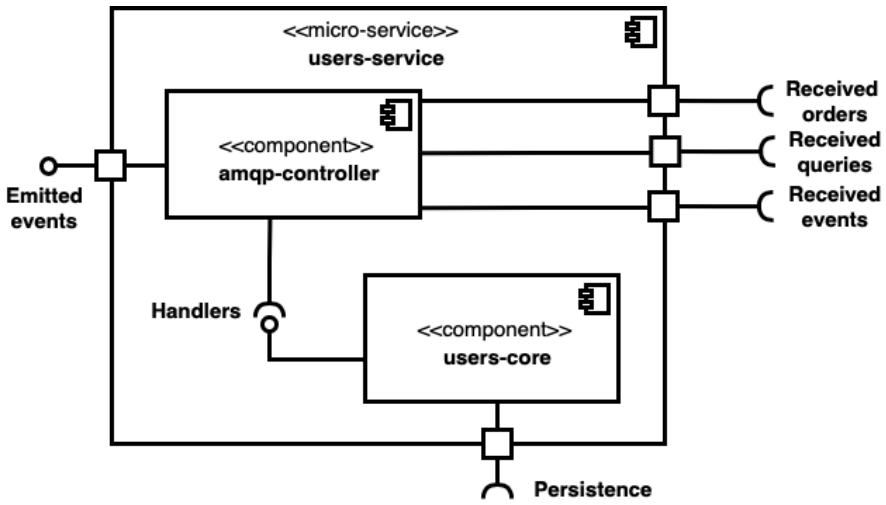

Figure 4. UML components diagram for the user service of *Circle*.

4.4. Virtualization and Deployment

The premise, given in Section 4.3, of not imposing specific environments leads to opt for solutions based on virtualization. Moreover, when we deal with dependencies, configurations, heterogeneous environments, or incompatible versions or systems, the installation processes used are complex and error prone.

A solution to these problems lies in virtualization, especially lightweight containers. Operating-system-level virtualization provides watertight runtime environments. This allows a high degree of isolation, at the same time that it is hardly an overhead to the system, since it is not necessary to emulate the entire virtualized system. For the purpose of the "socialized architecture", we consider this as an appropriate solution:

1. It provides an easy method to package software together with its dependencies and configurations.
2. It offers isolated execution environments.
3. It hardly implies an overhead in the system greater than the one that would derive from executing the service natively.

On the other hand, for the "socialized architecture" model to succeed, it is important to obtain a significant number of users. So, the existence of processes that automatically deploy the "socialized services"can greatly help to this end, and we can argue that this is a mandatory requirement. The deployment of *Circle* lets us to learn about the problems needed to be addressed for achieving such automation. Section 5.3 discusses the solution we took for deploying *Circle*and the issues we found.

4.5. Evaluation

The assessment of *Circle* is of importance, as it will help to appraise our proposal from at least two points of view: First, the correctness of the design decisions made to shape the architecture and, second, the feasibility of the technologies we selected. For such an assessment, we selected three standard metrics: the response time, the resource demand, and the latency.

Table 2 summarizes the results we obtained for response times when executing the different requirements. As it can be observed, the mean time was below 600 ms, even for concurrent request, which is a reasonable result. At the light of these results and with a deep investigation, not here reported, we can assess the following. Regarding technologies, we could inform the suitability of all of them. However, for GraphQL we report the following. We need to consider that the results may vary considerably depending on the selected fields. In the tests, we asked for information for each entity once at a time. For example, when querying users, we did not query information about his/her related posts. Nested queries would imply the need of launching a second service, which will make it

impossible to compare results homogeneously. Subsequently, we performed additional tests for evaluating nested queries. Then, we identified exponential response times. FR3 specially highlights this situation. Here, response times could reach even 6000 ms. This behavior for nested queries is a consequence of the sequential sending of the messages to the microservices. Following with FR4, the query related to posts is not sent until all the information regarding the users is obtained. This is the standard GraphQL behavior, but no trivial solutions exist.

Table 2. Response times.

Code	Success	T_{mean} Sequential (ms)	T_{mean} Concurrent (ms)
FR1	Yes	412.3	494.6
FR2	Yes	408.1	548.4
FR3	Yes	468.4	600.9
FR4	Yes	458.3	546.2
FR5	Yes	438.6	569.9
FR6	Yes	432.9	547.6
FR7	Yes	407.3	507.9
FR8	Yes	493.4	641.8
FR9	Yes	521.8	633.3
FR10	Yes	416.2	518.1

Table 3 presents the resource demands of the containerized services. The *notifications* service was identified as the most demanding. This is due to its interaction with the email external service. We can conclude that the results are satisfactory since the impact of the containers, in the overall context of the system, is low when they execute. The results were obtained using the very same tricks offered by Docker when executing.

Table 3. Resource demands (information obtained from Docker).

Service	RAM Mean Demand (MB)	CPU Peak (%)
users-service	85.0	4.5
content-service	95.0	3.0
notifications-service	110.0	6.0

Regarding latency times, an average access time between the replicas and the message broker of 35 ms was calculated. Given the high volume of messages exchanged by replicas and broker, it is especially important to get this figure as low as possible. This being one of the most critical factors for the performance of the system, we believe that we can be satisfied with the result.

5. Discussion

Circle was developed using an iterative approach. Once use cases were clearly defined, then a few of them were developed, so we could understand the technologies involved and refine the asynchronous interaction model proposed, see Section 4.1. After that, we developed an initial complete prototype, where agile concepts started to guide the project. Later, the prototype was refactored using Domain Driven Design [46], and a common library was defined, see Section 5.1. Finally, part of the project was redesigned following good practices on software quality. This last refactoring ended up producing the current version of our proof of concept.

In the following, we discuss three important aspects that came up in the development of *Circle*. They are key to understand the development of "socialized services" since they tackle decisions regarding implementation and design. On the other hand, the development of *Circle*, as a reference application for the "socialized architecture", implied to select many technologies. The choices necessarily were based on the design advantages that some of

them can offer over others. Appendix A reports the most interesting technologies selected for *Circle*.

5.1. Library

While implementing the first prototype, we understood the need of sharing code among different system components. We identified some common domain objects but specially the classes implementing the communication among services. Hence, we created the **circle-core** library, which is imported by all services, including the API *Gateway* (explained in Section 4.3). Their contents have been designed in three layers:

- **Domain layer.** Objects related to the application domain. For instance, a class to encapsulate unique identifiers.
- **Application layer.** Classes encapsulating the logic of the services. For instance, objects representing commands, queries, or events.
- **Infrastructure layer.** Classes managing communication among services and encapsulating AMQP [45] implementation.

Undoubtedly, as a design decision, the use of a common library reduces code duplication along the project. The idea here is that "socialized services" can reuse the **infrastructure layer**, while the implementation of the other layers can be seen as guidelines for designing the application specific needs (the implementation of *Circle* is available in https://github.com/Pitazzo/circle (accessed on 1 January 2022)).

5.2. GraphQL

GraphQL [14], developed by Facebook, is an API's design pattern trying to replace REST by overcoming some of its limitations. GraphQL, as a query language for a data model given by the server, executes on HTTP, while it interacts with web services in a richer way than REST. GraphQL offers three interaction methods:

- Using *queries*. The server supports nested queries, which can also be concatenated.
- Using *mutations*. These operations change the server internal state. Like *queries*, they can be concatenated and parameterized.
- Using *subscriptions*. They are a special data type but are similar to *queries*. However, they can update data dynamically. For example, if you subscribe to a web service for an asset that is "publicly traded," then you obtain its current value, and as soon as the server updates the info, the *subscription* updates the client.

GraphQL is used in *Circle* as a communication protocol between clients and the API *Gateway*. We chose it for several reasons—initially, because of its semantic richness but also because GraphQL is well-aligned with the CQRS pattern. On the one hand, it segregates *commands* from *queries*, as dictated by the pattern. On the other hand, the use of *subscriptions* enhances its symbiosis with CQRS.

Regarding the latter, CQRS makes it difficult to manage API clients, since there is not a choice for managing asynchronous responses to orders. A common solution is to introduce "polling" mechanisms, which can improve the user experience. By "polling" we mean to periodically execute the service for simulating synchronism. This solution, although functional, is not efficient nor elegant, from a design point of view. However, GraphQL *subscriptions* offer a solution to this problem, as follows. GraphQL allows one to subscribe to a query, which offers the result of a command, then synchronously receiving the result once it has been completed; obviously this does not break CQRS principles. However, for reaching this behavior, we need to use "domain events." Once the execution of an order has been completed, then, it triggers the publication of a "domain event" notifying this fact. The API *Gateway*, subscribed to this event, will be notified and will extract the *payload* result of the received event. As a result of the notification, the API *Gateway* will publish an update in the subscription of the client, which was registered after dispatching the command. Hence, the client can receive the result of the command without the need of "polling" mechanisms. Figure 5 shows the mechanism described.

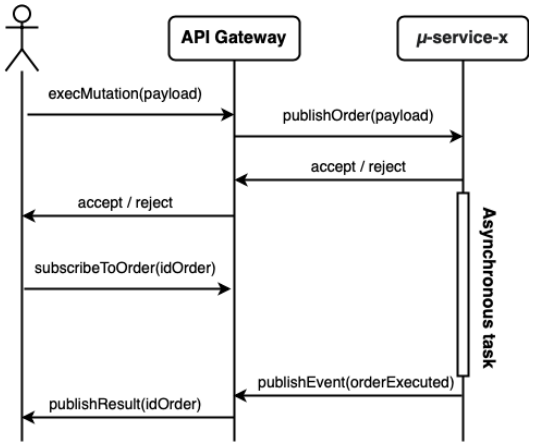

Figure 5. API *Gateway* and client interaction, using GraphQL subscriptions.

5.3. Containers and Security

Dependencies, configurations, and code in a "socialized application" should be packed guaranteeing fair execution in any kind of environment, without user intervention. Containers are a solution since they allow you to easily run replicas of the system, simply by executing a shell command. *Circle* uses Docker, a popular and lightweight virtualization solution, where each service has been deployed. Container images has been uploaded to a public repository in Docker **Hub** (https://hub.docker.com (accessed on 1 January 2022)). These services can be executed using the commands in Figure 6. Once the images have been downloaded, then their execution starts automatically. They are synchronized with the broker and the database, without user intervention. The user can execute only one service, one replica of each service or even several replicas of the selected services. Figure 7 depicts a deployment view of the system, which can be found in a **GitHub** (https://github.com/Pitazzo/circle (accessed on 1 January 2022)) repository.

The proposed deployment model comes with a security issue since we are using an infrastructure outside of the organization. In fact, we are submitting the code, with the business logic, and the credentials. Initially, we proposed obfuscation, i.e., to transform the code using cryptography techniques. In this way, it is not possible to apply reverse engineering to obtain or alter the code, while it can be successfully executed. Regarding the credentials, the literals in the code representing them are altered using bit-wise operations. The praxis of including credentials in the code is a non-recommended one. However, we proposed it for the sake of using obfuscation. The good praxis is to include them in a separate configuration file together with the code. In our case, the literals were automatically substituted, in compilation time, from those of an external file. Certainly, obfuscation can be an initial solution but is insufficient in industrial terms of deployment.

```
docker run -d pitazzo/circle:users-service
docker run -d pitazzo/circle:content-service
docker run -d pitazzo/circle:notifications-service
```

Figure 6. Commands for executing system services.

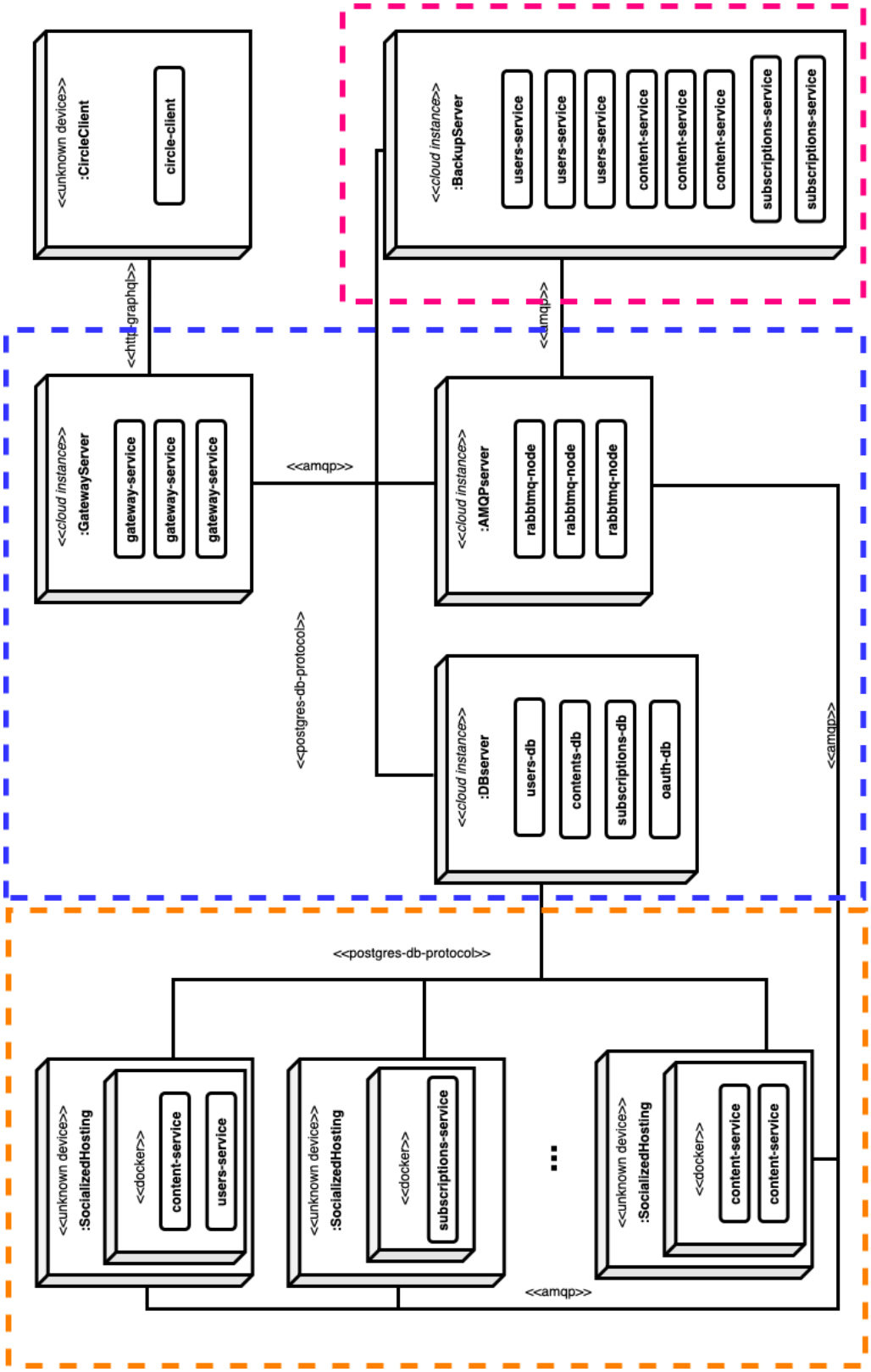

Figure 7. *Circle* deployment view.

6. Conclusions

The main contributions of this work are:

- An open proposal to overcome the cloud oligopoly, which explores solutions in the software engineering field, such as patterns and asynchronous interactions.

- The implementation of a social network that implements most of the underlying concepts and exposes the problems.

Although a great effort, in both aspects, was invested, we recognize that a significant amount of work is still necessary for the "socialized architecture" to become a reality. Therefore, the *open innovation* field, mentioned in the introduction, could be a proper mean for accelerating the proposed ideas as inputs for companies that want to boost their internal innovation. In the following, we discuss some pending, important aspects.

- How to distribute the persistence is a major open issue for the "socialized architecture". Although our research is focused on regaining "data sovereignty," our proof of concept still depends on cloud solutions for storing the system state.
- How to authenticate users in the "socialized architecture" is an important issue, whose design must be addressed. Answers to the following questions are needed. Should authentication happen in the API *Gateway* or in the services?. Should the authentication service be socialized? and what are the implications?
- Currently, the parameters of the orders are validated by the services that receive them, as indicated by CQRS. However, it would be interesting to involve the API *Gateway* in the process. This could bring two advantages. First, response times would improve, since no waiting times between services would exist. Second, there would be no need of having ready services for processing a message upon its arrival, since the broker could persist them. Hence, the system could accept all requests, from clients, irrespective of the number of available replicas.
- As it was identified in the prototype evaluation, Section 4.5, there is a bottleneck in nested GraphQL queries. It would be interesting to automatically parallelize such queries or to offer design guidelines to minimize the cases when such paralleling is not possible.

Future lines of work must necessarily address the previous pending issues. Concretely:

- Regarding persistence, future studies could investigate:
  - "Socialize" the persistence together with the replicas, managing distributed transactions and consensus algorithms in a classic way, for example, as in [47].
  - Leveraging decentralized hosting solutions, such as the Interplanetary File System [48], or new ones based on blockchain as a distributed ledger.
- Security needs to solve authentication, but also other issues, as surveyed in [49].
- Fault tolerance is another subject where advances are needed for ensuring the reliability and sustainability of the "socialized services".

Finally, it is needless to say that our proposal cannot compete with the current cloud solutions, neither in performance, nor security, nor reliability. However, the important aspect of this research is to understand that there exists an alternative to the current cloud, which can bring more freedom to the Internet.

Author Contributions: Conceptualization, P.M.-P. and J.M.; methodology, P.M.-P.; software, P.M.-P.; investigation, P.M.-P. and J.M; writing—original draft preparation, P.M.-P.; writing—review and editing, J.M.; supervision, J.M.; funding acquisition, J.M. All authors have read and agreed to the published version of the manuscript.

Funding: J.M. has been funded by project PID2020-113969RB-I00 of the Spanish Ministry of Science and Innovation.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

Sample Availability: The implementation of *Circle* is available in GitHub repository https://github.com/Pitazzo/circle (accessed on 1 January 2022).

Abbreviations

The following abbreviations are used in this manuscript:

API	Application Program Interface
CQRS	Command Query Responsibility Segregation
DDD	Domain-Driven Design
FR	Functional Requirement
NAT	Network Address Translation
REST	Representational State transfer
RPC	Remote Procedure Call

Appendix A. Technologies

In the following some of the technologies used for developing *Circle* are briefly reviewed. We consider important their discussion since it can greatly help to take informed decisions for future developments.

Appendix A.1. NestJS

NestJS [50] is defined as a NodeJS [51] framework for building efficient, reliable and scalable server-side applications. It is known for promoting good development practices and for using TypeScript [52], the typed version of JavaScript, which confers applications a plus in security and robustness. It offers native support for injecting dependencies, which offers lot of value to the "socialized architecture" since: (1) the communication between layers is very rich, and (2) the code aligns with the hexagonal architecture [53] promoting maintainability.

Appendix A.2. RabbitMQ

Being nuclear the communication among services, the choice of a *broker* is a decisive design decision. RabbitMQ [42] offers native support to AMQP [45], the protocol used in *Circle* for services interaction. Moreover, NodeJS owns libraries implementing mechanisms for a high-level interaction with RabbitMQ, such as amqp-lib. Applications for the "socialized architecture" need an infrastructure supporting the management of the queues, see Section 4.1, and message passing, both mechanisms are offered by RabbitMQ. If the infrastructure goes down, RabbitMQ is able to wake it up from the last valid state, by transparently managing the persistence of the messages in the queues. Also, it can store messages until some replica is available for consuming. Finally, it is important, for the "socialized architecture", its mechanisms for ensuring quality of service, which go beyond simple *round-robin*.

Appendix A.3. TypeORM

Regarding persistence, there are at least two important aspects for the "socialized architecture". First, to manage communication among services and their corresponding database representations. Second, the "re-hydration", i.e., how to transform stored entities into objects. We choose TypeORM [54], as it is the framework of reference for TypeScript. It implements *repository* [55] as a persistence pattern. Hence, it offers a class for each entity to interact with the persistence mechanism, with a transparent solution for managing SQL statements. The persistence model for the "socialized architecture"advocates for an independent data model for each service, which owns the entities need for implementing the corresponding use case. These entities share the same storage. For *Circle* we choose PostgreSQL [56], but certainly we consider it as an arbitrary decision since it does not offer significant advantages regarding other solutions.

References

1. Florea, D.; Florea, S. Big Data and the Ethical Implications of Data Privacy in Higher Education Research. *Sustainability* **2020**, *12*, 8744. https://doi.org/10.3390/su12208744.
2. Antonio, A.; David, T. The Gender Digital Divide in Developing Countries. *Future Internet* **2014**, *6*, 673–687. https://doi.org/10.3390/fi6040673.
3. Columbus, L. 83% of Enterprise Workloads Will Be in the Cloud by 2020. 2018. Available online: https://www.forbes.com/sites/louiscolumbus/2018/01/07/83-of-enterprise-workloads-will-be-in-the-cloud-by-2020/ (accessed on 1 January 2022).
4. Antova, L.; Bryant, D.; Cao, T.; Duller, M.; Soliman, M.A.; Waas, F.M. Rapid Adoption of Cloud Data Warehouse Technology Using Datometry Hyper-Q. In Proceedings of the SIGMOD Conference, Houston, TX, USA, 10–15 June 2018; Das, G., Jermaine, C.M., Bernstein, P.A., Eds.; ACM: New York, NY, USA, 2018; pp. 825–839.
5. Vargas, C. Cloud Market Share Report: AWS vs Azure vs Google Cloud 2019: McAfee. 2020. Available online: https://www.skyhighnetworks.com/cloud-security-blog/microsoft-azure-closes-iaas-adoption-gap-with-amazon-aws/ (accessed on 1 January 2022).
6. Rob van der Meulen. Understanding Cloud Adoption in Government. Available online: https://www.gartner.com/smarterwithgartner/understanding-cloud-adoption-in-government/ (accessed on 1 January 2022).
7. Faragardi, H.R. Ethical Considerations in Cloud Computing Systems. *Proceedings* **2017**, *1*, 166. https://doi.org/10.3390/IS4SI-2017-04016.
8. Fowler, M. CQRS. July 2011. Available online: https://martinfowler.com/bliki/CQRS.html (accessed on 1 January 2022).
9. Chesbrough, H.W. The Era of Open Innovation. *MIT Sloan Manag. Rev.* **2003**, *44*, 35–41.
10. Baierle, I.C.; Benitez, G.B.; Nara, E.O.B.; Schaefer, J.L.; Sellitto, M.A. Influence of Open Innovation Variables on the Competitive Edge of Small and Medium Enterprises. *J. Open Innov. Technol. Mark. Complex.* **2020**, *6*, 179. https://doi.org/10.3390/joitmc6040179.
11. Docker Inc. Docker—Empowering App Development for Developers. 2020. Available online: https://www.docker.com (accessed on 1 January 2022).
12. Mengistu, T.M.; Che, D. Survey and Taxonomy of Volunteer Computing. *ACM Comput. Surv.* **2019**, *52*, 59:1–59:35. https://doi.org/10.1145/3320073.
13. Durrani, M.N.; Shamsi, J.A. Volunteer computing: Requirements, challenges, and solutions. *J. Netw. Comput. Appl.* **2014**, *39*, 369–380. https://doi.org/10.1016/j.jnca.2013.07.006.
14. Facebook. GraphQL—A Query Language for Your API. 2012–2021. Available online: https://graphql.org (accessed on 1 January 2022).
15. Kirby, G.; Dearle, A.; Macdonald, A.; Fernandes, A. An Approach to Ad hoc Cloud Computing. *arXiv* **2010**, arXiv:1002.4738.
16. Che, D.; Hou, W.-C. A Novel "Credit Union" Model of Cloud Computing. In Proceedings of the International Conference on Digital Information and Communication Technology and Its Applications, Dijon, France, 21–23 June 2011; Cherifi, H., Zain, J.M., El-Qawasmeh, E., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; pp. 714–727.
17. Mengistu, T.; Alahmadi, A.; Albuali, A.; Alsenani, Y.; Che, D. A "No Data Center" Solution to Cloud Computing. In Proceedings of the 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), Honolulu, HI, USA, 25–30 June 2017; pp. 714–717. https://doi.org/10.1109/CLOUD.2017.99.
18. University of California at Berkeley. BOINC. 2021. Available online: https://boinc.berkeley.edu (accessed on 1 January 2022).
19. McGilvary, G.A.; Barker, A.; Atkinson, M. Ad Hoc Cloud Computing. In Proceedings of the 2015 IEEE 8th International Conference on Cloud Computing, New York, NY, USA, 27 June–2 July 2015; pp. 1063–1068. https://doi.org/10.1109/CLOUD.2015.153.
20. Ryden, M.; Oh, K.; Chandra, A.; Weissman, J. Nebula: Distributed Edge Cloud for Data Intensive Computing. In Proceedings of the 2014 IEEE International Conference on Cloud Engineering, Boston, MA, USA, 11–14 March 2014; pp. 57–66.
21. Babaoglu, O.; Marzolla, M.; Tamburini, M. Design and Implementation of a P2P Cloud System. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC '12*; Association for Computing Machinery: New York, NY, USA, 2012; pp. 412–417. https://doi.org/10.1145/2245276.2245357.

22. Beberg, A.L.; Pande, V.S. Storage@home: Petascale Distributed Storage. In Proceedings of the 2007 IEEE International Parallel and Distributed Processing Symposium, Long Beach, CA, USA, 26–30 March 2007; pp. 1–6. https://doi.org/10.1109/IPDPS.2007.370672.
23. Qin, A.; Hu, D.M.; Liu, J.; Yang, W.J.; Tan, D. Fatman: Building Reliable Archival Storage Based on Low-Cost Volunteer Resources. *J. Comput. Sci. Technol.* **2015**, *30*, 273. https://doi.org/10.1007/s11390-015-1521-6.
24. Neumann, D.; Bodenstein, C.; Rana, O.F. STACEE: Enhancing storage clouds using edge devices. In Proceedings of the 1st ACM/IEEE Workshop on Autonomic Computing in Economics, Karlsruhe, Germany, 14 June 2011; pp. 19–26. https://doi.org/10.1145/1998561.1998567.
25. Al Noor, S.; Hossain, M.M.; Hasan, R. SASCloud: Ad hoc Cloud as Secure Storage. In Proceedings of the 2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom), Atlanta, GA, USA, 8–10 October 2016. https://doi.org/10.1109/BDCloud-SocialCom-SustainCom.2016.17.
26. Mohaisen, A.; Tran, H.; Chandra, A.; Kim, Y. SocialCloud: Using Social Networks for Building Distributed Computing Services. *arXiv* **2011**, arXiv:1112.2254.
27. Chard, R.; Bubendorfer, K.; Chard, K. Experiences in the design and implementation of a Social Cloud for Volunteer Computing. In Proceedings of the 2012 IEEE 8th International Conference on E-Science, Chicago, IL, USA, 8–12 October 2012; pp. 1–8. https://doi.org/10.1109/eScience.2012.6404452.
28. Caton, S.; Haas, C.; Chard, K.; Bubendorfer, K.; Rana, O.F. A Social Compute Cloud: Allocating and Sharing Infrastructure Resources via Social Networks. *IEEE Trans. Serv. Comput.* **2014**, *7*, 359–372. https://doi.org/10.1109/TSC.2014.2303091.
29. Kuada, E.; Olesen, H. A Social Network Approach to Provisioning and Management of Cloud Computing Services for Enterprises. In Proceedings of the CLOUD COMPUTING 2011, the Second International Conference on Cloud Computing, GRIDs, and Virtualization, Rome, Italy, 25–30 September 2011.
30. Mcmahon, A.; Milenkovic, V. Social Volunteer Computing. *J. Syst. Cybern. Inform.* **2011**, *9*, 34–38.
31. Anderson, D.P. Globally Scheduling Volunteer Computing. *Future Internet* **2021**, *13*, 229. https://doi.org/10.3390/fi13090229.
32. Xu, L.; Qiao, J.; Lin, S.; Qi, R. Task Assignment Algorithm Based on Trust in Volunteer Computing Platforms. *Information* **2019**, *10*, 244. https://doi.org/10.3390/info10070244.
33. University of California at Berkeley. SETI@home. 2020. Available online: https://setiathome.berkeley.edu (accessed on 1 January 2022).
34. Fundación Vodafone. DreamLab. Available online: https://www.vodafone.com/dreamlab/spain (accessed on 1 January 2022).
35. Waheed, A.; Shah, M.A.; Khan, A.; ul Islam, S.; Khan, S.; Maple, C.; Khan, M.K. Volunteer Computing in Connected Vehicles: Opportunities and Challenges. *IEEE Netw.* **2020**, *34*, 212–218. https://doi.org/10.1109/MNET.011.1900603.
36. Cao, X.; Wang, F.; Xu, J.; Zhang, R.; Cui, S. Joint Computation and Communication Cooperation for Energy-Efficient Mobile Edge Computing. *IEEE Internet Things J.* **2019**, *6*, 4188–4200. https://doi.org/10.1109/JIOT.2018.2875246.
37. Twitter. Twitter—Red Social. 2021. Available online: https://twitter.com/ (accessed on 1 January 2022).
38. Richards, R. Representational State Transfer (REST). In *Pro PHP XML and Web Services*; Apress: Berkeley, CA, USA, 2006; pp. 633–672. doi:10.1007/978-1-4302-0139-7_17.
39. Tanenbaum, A.; Van Steen, M. *Distributed Systems: Principles and Paradigms*; Pearson Educación: London, UK, 2008.
40. Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M. *Pattern-Oriented Software Architecture—Volume 1: A System of Patterns*; Wiley Publishing: Indianapolis, IN, USA, 1996.
41. Taylor, H.; Yochem, A.; Phillips, L.; Martinez, F. *Event-Driven Architecture: How SOA Enables the Real-Time Enterprise*, 1st ed.; Addison-Wesley Professional: Boston, MA, USA, 2009.
42. VMware Inc. RabbitMQ—Messaging That Just Works. 2007–2020. Available online: https://www.rabbitmq.com (accessed on 1 January 2022).
43. Newman, S. *Building Microservices: Designing Fine-Grained Systems*, 1st ed.; O'Reilly Media: Newton, MA, USA, 2015; p. 280.
44. Kumar, A. *Cqrs (Command Query Responsibility Segregation)*; Independently Published: 2019.
45. OASIS. AMQP—Advanced Message Queuing Protocol. 2020. Available online: https://www.amqp.org (accessed on 1 January 2022).
46. Evans, E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*; Addison-Wesley: Reading, MA, USA, 2004.
47. Ongaro, D.; Ousterhout, J. In search of an understandable consensus algorithm. In Proceedings of the USENIX, Philadelphia, PA, USA, 19–20 June 2014; pp. 305–320.
48. Chen, Y.; Li, H.; Li, K.; Zhang, J. An improved P2P file system scheme based on IPFS and Blockchain. In Proceedings of the 2017 IEEE International Conference on Big Data (Big Data), Boston, MA, USA, 11–14 December 2017, pp. 2652–2657.
49. Tahirkheli, A.I.; Shiraz, M.; Hayat, B.; Idrees, M.; Sajid, A.; Ullah, R.; Ayub, N.; Kim, K.-I. A Survey on Modern Cloud Computing Security over Smart City Networks: Threats, Vulnerabilities, Consequences, Countermeasures, and Challenges. *Electronics* **2021**, *10*, 1811.
50. Mysliwiec, K. NetsJS—A Progressive Node.js Framework for Building Efficient, Reliable and Scalable Server-Side Applications. 2017–2020. Available online: https://nestjs.com (accessed on 1 January 2022).
51. OpenJS Foundation. NodeJS. 2020. Available online: https://nodejs.org/ (accessed on 1 January 2022).
52. Microsoft. TypeScript—Typed JavaScript at Any Scale. 2012–2020. Available online: https://www.typescriptlang.org (accessed on 1 January 2022).

53. Alistair Cockburn. The Pattern: Ports and Adapters ("Object Structural"). 2020. Available online: https://alistair.cockburn.us/hexagonal-architecture/ (accessed on 1 January 2022).
54. Open Source—Supported by Sponsors. TypeORM—Object-Relational Mapping. 2020. Available online: https://typeorm.io/#/ (accessed on 1 January 2022).
55. Fowler, M.; Rice, D.; Foemmel, M.; Hieatt, E.; Mee, R.; Stafford, R. *Patterns of Enterprise Application Architecture*; Addison-Wesley Professional: Boston, MA, USA, 2002.
56. Obe, R.O.; Hsu, L.S. *PostgreSQL: Up and Running A Practical Introduction to the Advanced Open Source Database*, 2nd ed.; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2014.