A Novel Cloud Computing Software Engineering Strategy Based on "Socialised Architecture"

Rajesh Kumar[*]

Department of Computer Science and Technology, Radha Govind University, Ganrke, India

Corresponding Author[*]

Rajesh Kumar
Department of Computer Science and Technology
Radha Govind University,
Ganrke, India,
Tel: 8396945480
E-mail: rajeshasuszen@gmail.com

Received: June 06, 2023, Manuscript No. IJIRSET-23-100870; **Editor assigned:** June 08, 2023, PreQC No. IJIRSET-23-100870 (PQ); **Reviewed:** June 22, 2023, QC No. IJIRSET-23-100870; **Revised:** August 07, 2023, Manuscript No. IJIRSET-23-100870 (R); **Published:** August 14, 2023, DOI: 10.35248/IJIRSET.23.4(2).007

Abstract

The cloud represents a revolution within the revolution of the internet today. However, an oligopoly supporting the cloud may not be the best solution, given the potential for ethical issues like privacy and even the transfer of data sovereignty. Our study, which we call the "socialised architecture," proposes a new disruptive approach that has the potential to fundamentally alter the current state of the cloud. This method is similar to those used in volunteer computing in that it attempts to pool unused computing resources present in the hardware owned by institutions and regular people. However, unlike existing options, ours does not involve developing hyper-specialized muscles in client machines. The novel aspect of this solution is the software engineering approach it proposes for building "socialised services," which, by means of an asynchronous interaction model, generates a system of lightweight micro services that can be dynamically allotted and replicated across the network. While an API gateway takes care of communication, modern design patterns like command query responsibility segregation help to compartmentalize domain events and persistence requirements. A fully functional proof of concept, a prototype called circle that implements a social network, was used to evaluate all of the preceding ideas. Circle has helped bring to light issues that need fixing. We agree with the evaluation's conclusion that venturing into this area of expertise is worthwhile.

Keywords: Software architecture • Micro service • Cloud • Distributed systems • Socialised services

Introduction

It's common knowledge that we reside in a highly digitised world, where computers form an integral part of our daily lives and are required for the growth of a respectable existence. It's not hard to see the positive effects they've had, such as the enhancement of countless processes, economic expansion, and increased wealth. The loss of jobs to automation, the gap between the sexes and generations in terms of technology access, and the proliferation of privacy scandals in the media are all shortcomings that are not hidden [1,2]. While these are all very valid concerns, there is also the issue of how dependent these cloud hosted systems are. In 2018, in house deployments, or those hosted on the company's own infrastructure, made up 37% of all deployments.

In 2020, only two years from now, 83% of deployments will be made using cloud based solutions [3]. The cloud's market share has grown by 10% in just the past two years. By 2025, cloud services will have replaced all others. The advent of cloud hosting, which itself represents a revolution within the internet revolution, is to blame for this development.

Companies make the move to the cloud primarily because of the cost savings afforded by the cloud computing revolution [4]. When compared to on premises solutions, the cloud provides levels of scalability, replication, and fault tolerance that are simply not possible. Because of its ease of use, infrastructures that would have taken several systems engineer's months to set up can now be set up with just a few mouse clicks. However, Amazon, Microsoft, And Google control approximately 74% of the cloud hosting market [5]. They monopolise a platform used by 93% of the world's corporations. The numbers add up: 68 out of every 100 cloud users businesses will rely on cloud storage for some kind of data. The power we are giving these corporations is comparable to that of states if, as we have said, those 68 systems are nuclear components in the lives of millions of people. Without question, half of all government agencies currently use cloud based solutions in their operations [6].

There may be moral issues due to the concentration of power in the hands of so few companies [7]. In addition to providing the processing power, cloud service providers also host the information that is used by the applications. Even though users are shielded from unauthorised access to their data by legal frameworks, nothing stops the operator or the same company from accessing the contents of the infrastructure, which they ultimately control, without leaving any traces. Although privacy is a pressing concern, we can't overlook the possibility of other issues, such as the blocking of access to vital resources, the alteration of previously collected data, or the interruption of service for nefarious reasons. One potential downside of moving to the cloud is that it could mean giving up control over our data and the reliability of our systems to multinational conglomerates.

However, just because these problems are theoretically conceivable doesn't mean that they will actually occur. They're not inevitable, but they're worth thinking about and, ideally, avoiding anyway. Of course, safety should not be sacrificed for the sake of abandoning the cloud's many advantages. Typically, for profit businesses are responsible for creating new computer systems. Fighting oligopolies for social and ethical reasons will never be a convincing argument for abandoning that fight. If you want to change things and start hosting your own systems again, one strategy is to work hard to make those systems profitable in the long run, by providing viable alternatives to the cloud's "status quo" that can compete on price and reliability. This was meant to be the outcome of the study. The most important results of this study are:

- The central ideas for a new cloud architecture based on hosting "socialized services" are explored. The solution relies on micro services and an asynchronous interaction model.
- Introduces a plan for creating "socialised services" by leveraging advanced software engineering patterns like Command Query Responsibility Segregation (CQRS) and an Application Programming Interface (API) Gateway [8].
- Creates a working model, named circle that demonstrates the viability of the proposed solutions and highlights any remaining issues.

This article begins by highlighting some of the many problems to address and the many disciplines involved in developing a viable alternative to the cloud, including but not limited to software engineering, distributed systems, and security. The study's preliminary ideas are a small step in the right direction, especially for software engineers. Because of this, subsequent studies and businesses can build upon this foundation by refining and envisioning even better solutions, which can eventually lead to a truly sustainable new cloud. However, the open innovation field proposes an alternative innovation model wherein businesses commercialise both external and internal ideas by employing external and internal routes to market. Therefore, ideas from outside the company have a variety of options for commercialization, including licensing deals and new ventures. Thus, the work presented here can form the basis for a new product, namely, this new sustainable cloud that fixes the issues with the current cloud [9,10].

This article's results should be interpreted as both a vision statement and a "proof of concept." Regarding this, we propose an architecture that, at a reasonable price, permits dependable and secure deployments outside the current cloud without incurring the unsustainable on premise approach, which has drawbacks such as high maintenance costs and equipment obsolescence. The term "socialised architecture" describes the system we developed.

The rest of the article continues below. The article's research is contextualised and a literature review is presented in section 2. The methodology used in this study is discussed in section 3. Our findings and evaluations are presented in section 4. In section 5, we'll look at some of the most relevant aspects of this study.

Purpose statement and review of related work

The term "socialised computing" refers to the practice of sharing a system's computing resources and, eventually, its storage, among its users in the interest of working with businesses looking to take actions with a transfer motive impact, or with any entity betting on a decrease in cloud hosting. Whether out of altruism or self-interest, "socialization" entails supporting a portion of the system by hosting its components. This results in highly dispersed settings running on a varied set of servers.

Instead of the centralized approach that has been taken in the past which is discussed and critiqued in section 2.1 we propose decomposing a multipurpose system (the core of any contemporary system) into a collection of decoupled services. As a result, any user who wants to take part in the "socialization" process can host a copy, or copies, of any of the "socialised services" that make up the system on their own local machine. These socialised replicas would function as equals to those hosted locally or in the cloud by the system's owner. Depending on the current demand, the company could dynamically increase or decrease the number of socialised replicas. A critical mass of users would make it possible to eliminate cloud reliance entirely. Although the concept of sharing computational resources equitably is novel, several established technologies already exist to put it into practice. Our proposal is supported by the following primary tools, methods, and technologies:

- Virtualization. In particular, container based lightweight virtualization likes docker [11].
- Non real time conversation. New communication patterns made possible by developments in distributed systems enable low latency, dependable transmissions.
- Micro services. A pattern for designing systems that encourages scalable, cohesive services with low coupling between them.
- Patterns like command query responsibility segregation help developers keep their asynchronous code organized.

Late in the 1980's, a phenomenon known as Volunteer Computing (VC), also called cycle stealing system or public resource computing, emerged. The idea of helping out a good cause by providing access to local computing resources is not new; the review in updates dozens of works in the field of VC, proposed in the last few years [12]. In VC, anyone with access to a computer can pool their unused processing power to complete data and computation intensive tasks [13].

According to Mengistu, T.M, a VC system consists of resource nodes, which are volunteer nodes that donate unused resources, and a resourcecontroller,which manages the donated resources and acts as an entry point for both volunteers and VC system users. This section of summarizes the operation of VC systems. The user submits the task(s) to the resource controller, which, after preprocessing, chooses an appropriate resource node(s) to deploy the task(s) to; the task(s) are then executed by the resource node(s) and the results are sent back to the Controller. VC systems can have a centralized (client/server, C/S), decentralized (P2P), or hybrid architecture, depending on how they are deployed. In a C/S setup, one or more dedicated computers serve as servers, managing other computers' access to shared resources. Without a central authority, communication can be coordinated thanks to P2P's reliance on volunteers acting as resources and controllers. The advantages of C/S in terms of security and trust leverage are combined with the scalability and adaptability of P2P in hybrid architecture. An advantage of a central server is the global resource directory it provides.

It's worth emphasizing that all three models (C/S, P2P, and hybrid) work well with the "socialised architecture" and can be used to implement it. This is because the infrastructure layer imposes no limitations on deployment and instead merely provides an API with a subset of features. In reality, this is a resource controller, albeit a lightweight one, whose responsibilities include resolving service addresses, balancing loads, collecting data, and translating protocols. The infrastructure layer then functions as a decoupled, asynchronous component, communicating with clients and resources. Therefore, we can pick an appropriate implementation based on the features that need to be prioritized, such as scalability versus security. Circle's resource controller serves as an API Gateway thanks to its use of the graph query language [14].

The "socialized architecture" aims to propose a comprehensive alternative to the current "status quo" of the cloud, but we have not found any works in the literature that share this objective. Nonetheless, in the VC field, Kirby, et al. and Che and Hou discussed initial models for desktop clouds, then proposed alternates of possible architectures and discussed some of the challenging aspects to address, all of which are similar to our work. Multiple projects have taken inspiration from these prototypes and proofs of concept to implement their own solutions [15-17]. In, we find the Cloud project, which is based on. Clients, similar to what happened in BOINC related projects, run guest Virtual Machines (VMs) and install a middleware that controls the node, in this case for monitoring its utilization and QoS. Cloud Stack was used to create this product [18].

The Virtual Machines (VMs) used by AdHoc cloud are hosted by volunteer computers running BOINC and virtual box [19].

The server schedules, monitors, and manages the jobs, the entire system, and the VMs in C/S architecture. To communicate with the server and carry out the tasks, the client sets up some kind of middleware. Both paid and unpaid nodes are used in nebula [20]. Chrome's native client sandbox is used to perform the computation, eliminating the need for Virtual Machines (VMs) and allowing full advantage to be taken of chrome's security features. Volunteers are tracked, given information, and assigned tasks while the system manages load balancing. To achieve a fully distributed cloud system managed by an underlying network, P2PCS employs P2P architecture (Table 1). To make requests to the system and communicate with peers, middleware must be installed on every participating node, in this case as a daemon. For P2PCS, a Java based prototype has been created. Finally, the key features of these desktop clouds are summarised and compared to the "socialised architecture" in the Table. Whether or not this method defines a software engineering approach for creating client applications is indicated in the "Sw Eng. Approach" column [20,21].

Table 1. Comparison with desktop clouds.

Approach	Architechture	Only volunteer	Virtualization	Middle ware in client host	Sw. eng approach	Related projects
Socialized architechture	C/s, P2P, hybrid	No	Containers	No	Yes	
Cucloud	C/s	Yes	VMs	Yes	No	BIONIC, could stack
Adhoc cloud	C/s	Yes	VMs	Yes	No	BIONIC, could stack
Nebula	C/s	No	No	No	No	Chrome
P2PCS	P2P	Yes	No	Yes	No	

Storage home, Fatman, STACEE, and SAS cloud are just some of the projects that focus on a specific aspect of the cloud, such as cloud storage. They are community driven cloud storage initiatives that offer substitutes for paid services like Amazon's simple storage service (S3) [22-25]. With C/S architecture, storage home pools the space that its users donate and offers backup features. Fatman is an archival system that utilises tens of thousands of idle servers in C/S architecture. To achieve its goals of reducing energy consumption and increasing user participation, STACEE proposes four layer architecture (backend, services, adaptation, and economic indicators) based on economic metrics such as energy [26]. It makes use of a scheme in which resource provision is handled in a dynamic fashion. SAS cloud is an ad hoc, mobile cloud storage service that guarantees the safety of your data. The hybrid architecture in use here features both a centralized administration and node registering clients. In addition, there is a sizable body of work that focuses on enhancing cloud based services, such as social networking. SoCVC and SOCIALCLOUD are two examples. The first is a proposal in a paradigm that uses trust relationships prevalent in social networks like Facebook and twitter to construct cloud computing services. To verify the identities of its users, SoCVC (Social Cloud for Volunteer Computing) taps into the APIs of popular social media platforms like Facebook. In addition, it suggests algorithmic indicators of users' social reputations. At last, references discuss some other systems worth considering in this group [27-30].

There are fully functional projects in the field of VC that can be compared to the "socialised architecture," despite the fact that their goals and features differ significantly from our own. They provide technological answers that could be useful down the line, and could even be recycled here. The scientific community employs BOINC for compute-intensive tasks. It's a multipurpose piece of middleware that can be downloaded and installed on any willing computer. New task assignment algorithms that claim to minimise completion time are given in, while explains the BOINC architecture and the scheduling problem for assigning tasks to volunteers. Conventional technologies like relational databases, web services, and daemon processes are all used by BOINC. We can see that the project's intended architecture and technological components are very different from what we've proposed. The Search for Extraterrestrial Intelligence at home (SETI home) is a BOINC hosted signal processing project. Using C/S architecture, the clients pull down the work units, run them locally, and then send back the results. The redundant computation used to identify malicious users is the most notable feature. Again, our proposal is very different from SETI home in terms of both its purpose and its design. Mobile app dream lab facilitates collaborative computing for COVID-19 projects. For BOINC and SETI home, this is accomplished by installing a programme on regular computers so that they can carry out computationally demanding tasks. The users in all three cases are acting altruistically [31-34]. In these projects, however, the local software components always serve as mere slave executors of a single, highly specialized task that requires extensive computing resources. In contrast to our proposal, the software components are not integral parts of a single supercomputer but rather are orchestrated from afar. A muscle that has become overly specialized for a single task.

We found the following to be current among VC proposals. Combining Virtual Communication (VC) with Vehicular Ad Hoc Networks (VANET) is proposed in Waheed, A. The plan is to put unused computer power in vehicles to good use. Whether vehicles are parked, waiting at a light, stuck in traffic, or moving along smoothly, the work defends the existence of a large pool of available resources in each of these situations [35]. Instead of suggesting a specific architecture, we will discuss the principles behind using a VANET for virtualization computing (master slave computation). Specifically, taxonomy for this emerging field is discussed, along with examples of its potential application and the difficulties that may arise from doing so. High performance computing, autonomous vehicles, intrusion detection, content distribution, connectivity, and efficient communication are just some of the standard uses for this computing surplus. For Mobile Edge Computing (MEC) systems, Cao, et al. proposed a new user cooperation approach for computation and communication within the realm of 5G cellular technologies [36]. The goal is to enhance energy efficiency for computations with limited tolerance for delay. The work is predicated on the idea that nearby access points and cellular base stations can take over computation-intensive and latency critical tasks from nearby wireless devices.

Materials and Methods

Our research methodology consisted of first creating a proof of concept. In order to evaluate the "socialised architecture" concepts presented in the previous section, we decided to create a working prototype. Due to the dearth of literature on the topic, a purely theoretical proposal and discussion may not be adequate treatment of such a revolutionary idea. As an added bonus, when confronted with a real development, details that would normally remain concealed become apparent.

To fully develop the "socialised architecture" would require the invention of numerous difficult features. New technologies would have to be created, complex security issues would have to be solved, and micro-services tailored to specific use cases would have to be designed. Next, we suggest putting into action a prototype system at a small scale that is robust enough to take into account as many possible scenarios and features while still providing a solid foundation upon which to reason and draw actionable conclusions.

On January 1, 2022, we launched circle, a social network with features and aesthetics not dissimilar to Twitter's [37]. Circle complies with a comprehensive grading scheme. Short texts written by registered users and rated by other users' favorability are stored and can be retrieved by the most recent or highest rated publications or by the most popular users. Subscriptions and alerts are another feature provided by circle. The most important features are outlined in Table 2. The domain may seem simple at first glance, but it actually has a lot of depth and can accommodate many different applications. It can also be broken up into smaller, more manageable pieces that can be handled by their own micro services without requiring any heavy duty cross communication.

Table 2. Requirements for circle.

Users
FR1 A user can sign up for the system with just a user name and email address.
FR2 users are able to update their information in the system.
FR3 you can look up other user's profiles on the system by searching for their username in a list of all users, or you can query the n-top users who have the most publications or the most subscribers.
Contents
FR4 publications, complete with title and text, can be added to the system.
FR5 users are able to "like" other users' posts on the platform.
FR6 you can use the system to look at the n most popular recent publications or the n most popular liked publications.
Subscriptions
FR7 users can subscribe to one another on the system.
Notifications
FR8 subscribed users are notified via email whenever new publications are made available.
FR9 when another user subscribes to a user's publications, the system notifies that user *via* email.
FR10 when another user "likes" one of a user's publications, the system sends that user an email.

Circle is queried *via* an API in GraphQL, and it provides services to support user needs, persistence, and the requisite technical infrastructure. In order to consistently meet the requirements for experimentation and analysis, the richness of the system necessitates dealing with the following aspects, despite the lack of a user interface:

- The importance of enforcing integrity checks while responding to events in the domain. When we run a query like "who publishes the most?" the users and contents micro services must be in sync, despite the fact that they each manage their own sets of data.
- The necessity of coordinating with third party services in response to internal triggers. For instance, when notifying subscribed services of newly published content via email.
- The requirement for combining data from multiple service queries. For instance, when pulling up a user's profile and seeing a list of all of their published works.
- Services need to be scaled up. Replicas for a service should be easily added.

Results

The proposals being considered at the moment to actualize the ideas behind the "socialised architecture" are presented and discussed in this section. Our ultimate goal was, as was mentioned earlier, to pioneer new research territory. Thus, these are preliminary ideas, a jumping off point for additional study. In our opinion, the best way for researchers to grasp and understand the underlying concepts is to provide an initial, albeit modest, body of solutions. Therefore, the reader should not assume that all the solutions explored here are the best; what we defend is that they work to implement the "socialised architecture", merely to demonstrate the idea's feasibility.

A model of communication that is not real time

In the modern cloud, every aspect of the underlying system is under direct management. Because of this high degree of replication, service level agreements typically guarantee cloud operations at rates higher than 99%. To further ensure low latencies in their data centers, these companies spend vast sums on the installation of submarine cables and optic fiber. In a nutshell, this top notch infrastructure is responsible for the reliability and safety of cloud deployments [38]. But the "socialised architecture" can't reap the benefits of the cloud because it runs on a heterogeneous leased infrastructure comprised of different hardware solutions. Instead, there are new issues that have emerged:

- Disparate lag durations. A raspberry Pi connected to a home network or a high performance data centre at a university could both host the same service.
- Poor connection stability. Quality of Service (QoS) policies in home networks vary from those in ad hoc networks in data centres. Additionally, they are impacted by the actions of other users who consume bandwidth. A user who is also hosting a copy of a "socialised service" might watch a video-streaming service at the same time.
- IP obfuscation and network address translators. These days, public IP addresses for private networks are rare.
- Most of them are hidden behind a NAT4, and their ISP can swap out their shared IP address at any time. As a result, it's possible that other services won't be able to send messages to your "socialised service" instance, as they won't know how to find it. Therefore, it's important to acknowledge that the "socialised architecture" as a whole can't be relied upon. Since synchronous communication protocols like REST and RPC require low latency, network stability, and a means to enroute messages, we ruled them out due to the difficulties we encountered when attempting to implement them [39]. Thus, we expect that interactions will be required among "socialised service" without the need for all interlocutors to be present at the same time. Communication patterns that are asynchronous are the obvious choice for such situations. Message passing makes sense here, and when coupled with the idea of a message broker, the issue can be managed [40].
- Since senders can keep working until a response arrives, if necessary, latency times become less of a concern.
- As long as the broker is operational, the network's instability is mitigated. If a replica that could receive a message is no longer connected to the network, the message will be forwarded to the next available replica in the queue. For instance, if a network partition prevented any replica from reaching the broker, the message would still be stored locally until a replica became available to process it.
- The broker acts as a centralised agent, ensuring that messages are properly routed to the appropriate services. The services can always find the broker, which is hosted at a static address, even if they are behind a NAT. Furthermore, the public IP change procedure is made entirely open and accessible.

By using asynchronous pattern interactions, the system is effectively transformed into a reactive one. Then, each service can function autonomously while still responding to system wide domain events. The "socialised architecture" is an example of what is known as a "event driven architecture". Therefore, there is now total separation between the various services [41]. The use of a broker to aid in the architectural design also has the following additional benefits:

- Provides a centralised location for configuring various aspects of service delivery, such as delivery policies and timeouts.

- Facilitates fewer service endpoint security measures by limiting attack surface to the broker.
- Offers a consolidated approach to gathering analytics data.
- Multiple protocols and implementations are available as open source for the broker's operation. In particular, the RabbitMQ protocol will be highlighted [42].

However, we can also point out some drawbacks:

- The broker then becomes the system's single weak link; if it goes down, everything else in the system fails with it. However, this is a well-studied issue, and replication and clustering mechanisms are available in most implementations to help lessen the impact.
- Adding the level of replication required to prevent the aforementioned risk increases the cost of running the broker. There are numerous ways to plan and carry out the implementation of an asynchronous model. We are confident that queuing systems are now a viable option due to advances in technology. Messages, which are relatively simple data structures, can be published and consumed in queuing systems thanks to a queue that is managed by a broker. There are many possible queue management policies; in Circle, we used First-In, First-Out (FIFO). According to the CQRS pattern, a message in the "socialized architecture" is a serialized object that represents a command, a query, or an event. A queue will stand in for each individual service [43-45].

Using queues to implement an asynchronous model requires several design choices:

- Each service's commands, queries, and events will be broadcast as messages. They will be released in a special forum.
- If multiple services subscribe to the same message, that message will be replicated that many times. Each replica will add items to the queue that represents this type of service, and this queue will be linked to the publisher service's exchange. This prevents users from competing services from intercepting each other's messages.
- Each type of command, query, or event will have its own dedicated queue, but all replicas representing the same service will use the same queue. In this way, multiple copies of an item compete for the same message, with only one ultimately receiving it. Therefore, adding more replicas to the queue that represents the service being scaled eliminates the need for load balancing. This is what is meant by "scalability by design."
- Each copy of the service has its own queue for receiving responses. This is because the name of the queue where the executor service must respond is appended to the publication of the query by each replica [46].

The queue layout for implementing the suggested asynchronousis model shown in Figure 1.

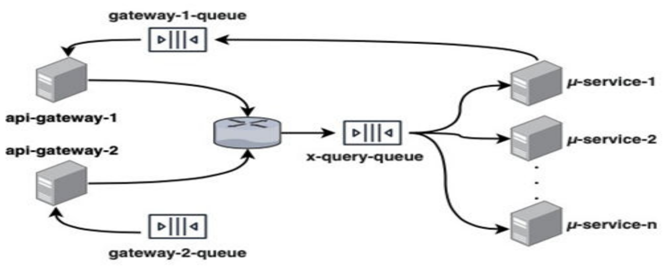

Figure 1. The system of queues.

The method can be illustrated with ease using FR4 of circle. The contents service will directly communicate with the users service in a classic REST (Representational State Transfer) architecture to increase the author's publication count. However, in our reactive proposal, the contents service only announces a new post *via* an event upon receiving the publication. All subscribed services will respond appropriately per the event's business logic. This allows for the greatest possible service decoupling while eliminating any delays caused by direct communication [47-50].

Constructions based on micro services

By "partitioning the system" into modular software components that can be independently "containerized" and hosted on external servers, "socialised services" can be achieved. A key motivation for the socialisation process is to have as small an impact as possible when hosting a replica on the publicly leased hardware, so the smaller the pieces, the better. No one is going to host a container if it requires a significant amount of effort or resources. Therefore, it is critical to "minimise" the replicas in terms of both physical size and the resources they require.

Since "partition of the system" and "minimization of such partitions" are both must haves, the obvious architectural solution is micro services. According to Newman, they're "an approach to distributed systems that promotes the use of small independent services with their own life cycles, which collaborate jointly." Micro services advocate a decoupled architecture in which individual services talk to one another while striving for maximum cohesion and minimum coupling. For example, the "socialised architecture" aims for small services, which can be obtained by narrowing the focus of the services to specific domains and use cases.

We are cognizant of the fact that size reduction is incidental to the principles underlying the micro services architecture rather than its primary focus. For us, however, this kind of simplification is what ultimately decides which pattern to use. Additional benefits of micro services for "socialised architecture" include:

- **Resilience:** When one service fails in micro services architecture, only the dependent services are impacted, rather than the entire system. This is significant for the "socialised architecture" because, as we've established, the foundational infrastructure is more prone to breakdown.
- **Scalability:** Compared to monolithic architectures, micro services are more scalable. It is sufficient to horizontally scale only the services associated with an overloaded use case. In our setting, this is preferable because it allows for quicker response times when there are fewer copies of a service.
- **Easier deployment:** A service can be deployed more quickly and easily than a large programme. There are fewer moving parts, and configurations can be simplified with less effort. This is crucial in urging end users to set up replicas in their own communities [51].

There are drawbacks to using micro services. To begin, the inherent communication processes and the variety of failures they may cause make their development significantly more difficult than that of monolithic approaches. Second, there will be an increase in the complexity of the testing procedure. Third, more delays are experienced if synchronous communication patterns the usual method for them are proposed. The latter shouldn't be an issue here.

Methodology for progress

The "socialised architecture" should not impose a particular programming language, development framework, or any other restriction necessary for the environment where "socialised services" will be deployed, in contrast to the methods discussed in section 2.1. In SETI home, for instance, the tasks can only be performed in the specific conditions that the client application is given. Their services need to be built with cutting edge software engineering for the "socialised architecture" to be a viable option. In section 4.4, we'll talk about how virtualization technology is a natural next step after satisfying this mandate. The need to provide design guidelines for the creation of "socialised services" is a second implication that follows.

The "socialised architecture" relies on the concept of replicating services across a network of personal computers to meet the distributed nature of the network's computational demands. Micro services are small, independently deployable software components that provide core system functionality, such as business logic and a persistent data store. However, as discussed in Section 4.1, the asynchronous model and the reactive nature of the architecture must be factored into any such design. The Command Query Responsibility Segregation (CQRS) pattern is one approach we're considering for the micro services architecture. Furthermore, it is helpful in addressing the right "separation of concerns," which is an advantage when developing targeted micro services. We discuss the pattern's main features and its applicability to the "socialised architecture" below.

With CQRS in place, a system can accommodate two distinct modes of user engagement: commands and queries. Actions that alter the state of the system but yield no information are called commands. Queries, on the other hand, do not alter the state of the system but do provide the user with a result. Using distinct processing models, the system will direct queries and commands down distinct paths. However, CQRS introduces the idea of bus. Handlers are subscribed to one or more of several buses that are used to dispatch commands and queries. The bus typically only handles calls to local functions that stand in for the handlers, leading to a unified method. Due to the fact that our system is built on separate modules, or micro services, this cannot be implemented. Instead, we decided to build an API gateway through which users could issue commands and retrieve results. Circle packages commands and queries into an AMQP message, which the API gateway can then help implement. The next step is for services to sign up for these alerts. Classical handlers accomplish their goals through the action taken in response to a message's arrival. The essence of our CQRS implementation is depicted in Figure 2.

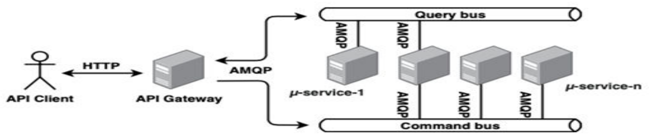

Figure 2. High level view of the CQRS pattern applied to micro services.

In micro service architectures, the API Gateway is a typical design pattern. In particular, it will be responsible for resolving service addresses, balancing workloads, collecting data, and translating protocols. Additionally, it could function as an authentication entity while simultaneously reducing the security exposure of the system. Negatives include that it is dependent on existing infrastructure and has a single point of failure.

Each service, when following CQRS, acts as the "runner" of the logic for its respective "use case." It is also beneficial for each service to be replicable and not state owned. As a result, the system can run multiple copies of the service without any extra configuration. For the architecture to deal with all of those limitations, each service must also perform the tasks below:

- Create a service interface that provides the designated system features. In CQRS terminology, API operations consist of commands and queries.
- Pay attention to the requests (commands) made of the API. After checking the inputs, it'll either approve or decline the request. If the order is relevant, it will be carried out at some point.
- Send out domain events because you've been told to.
- Domain events are generated by other services, so you can listen for them and act accordingly.
- Communicate with a persistence service whenever is necessary for a command or query.

To mimic the primary system features, we built a user service, a content service, and a subscription service into circle. In Figure 4, we see a UML components diagram for one of the circle services. It accurately reflects our definitive proposal for the architecture's various services. Therefore, every service consists of several constituent parts:

- An AMQP controller that coordinates all system wide messaging. It responds to queries, commands, and events to which it has subscribed. Additionally, it broadcasts domain events used by the business logic it enacts.
- The heart of the service, which contains the business logic and controls the persistence of its individual parts. In the past, it would register call backs in the amqp controller to be run as required.

The benefits of implementing CQRS are laid out in detail in Fowler, M. To counteract the "anaemic" nature of CRUD systems, Fowler recommends using CQRS first. In addition, Fowler stresses CQRS' efficacy. However, due to the new way of thinking that is required of the developer, CQRS does add complexity to the system. In the case of a unit session, for instance, a command should be executed whenever the system changes state to set up the session. Therefore, the system should not return anything beyond the order's acceptance. However, the proposed subscription mechanism is required for the system to notify the user of the command's outcome. Our asynchronous model is well-suited to accommodate such a mechanism.

In a nutshell, the implementation of CQRS is expensive. However, its speed and asynchronous character led to its proposal. As was previously mentioned, we have to factor in the inherent hardware infrastructure's inherent unpredictability. Because of this, we assumed the best case scenario and worked backwards to ensure maximum performance. Finally, our case study is not particularly calculation intensive or time sensitive, but we did implement it according to our architectural proposal so that we could get reliable results and learn the appropriate lessons (Figure 3).

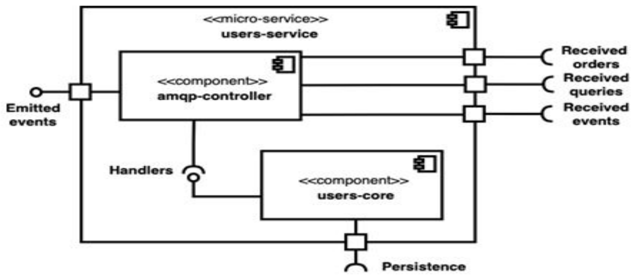

Figure 3. UML components diagram for the user service of circle.

Using virtualization in deployments

Based on section 4.3's premise of not forcing rigid setups, virtualization-based solutions are preferred. The installation procedures we use are also complicated and prone to mistakes because of the presence of dependencies, configurations, heterogeneous environments, or incompatible versions or systems.

Lightweight containers and other forms of virtualization could be the answer to these issues. Virtualization at the OS level creates impenetrable development and production environments. Since it is not necessary to emulate the entire virtualized system, this allows for a high degree of isolation while imposing almost no additional burden on the system. We think this is a good answer for the "socialised architecture":

- It provides a simple approach to packaging software with all of its prerequisites and settings.
- It provides separate environments for programmes to run in.
- It almost never entails additional system overhead compared to what would result from running the service locally.

However, a sizable user base is crucial to the success of the "socialised architecture" model. We can therefore argue that the presence of processes that automatically deploy the "socialised services" is a necessary condition for achieving this goal. The circle rollout teaches us what challenges must be met to reach that level of automation. The approach we took to deploying circle and the problems we encountered are discussed in section 5.3.

Evaluation

Circle's evaluation is crucial because it will shed light on the correctness of the design decisions we made when shaping the architecture and the viability of the technologies we chose. We chose three widely used metrics response time, resource demand, and latency to conduct this kind of evaluation (Table 3).

Table 3. The execution times we measured for each requirement are summarized.

Code	Success	Tmean sequential (ms)	Tmean 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

The execution times we measured for each requirement are summarised in Table 2. Even with multiple requests being sent at once, the average response time was less than 600 ms. We can evaluate the following in light of these findings and a thorough investigation, which is not presented here. With our knowledge, we can assess the viability of any technological option. But here's what we found out about GraphQL. Depending on the parameters chosen, the results could be quite different. In the trials, we only made a single data request per entity type. For instance, we did not inquire about a user's related posts when conducting searches. It would be impossible to make fair comparisons of results if nested queries required the introduction of a second service. We then ran more tests to determine the effectiveness of nested queries. Then, we determined that the response time was exponential. This is emphasised in FR3 in a unique way. It's possible that response times here could go up to 6 seconds. This behaviour for nested queries results from the messages being sent to the microservices in a linear fashion. In accordance with FR4, the post-related query is held back until all user data has been collected. This is the expected behaviour of GraphQL, but there are no simple solutions.

The requirements for the containerized services resources are listed in Table 4. It was determined that the notifications service required the most resources. This is because of how it communicates with the world outside of email. Since the containers execution has little effect on the system as a whole, we can say that the results are satisfactory. The outcomes were accomplished by employing the same docker provided techniques as the original execution.The average latency measured between replicas and the message broker was 35 milliseconds. Considering the sheer number of communications between replicas and the broker, minimising this number is of paramount importance.

Table 4. Resource demands (information obtained from docker).

Service	RAM Mean demand (MB)	CPU peak (%)
Users service	85	4.5
Content service	95	3
Notifications service	110	6

As this is a major contributor to the system's efficiency, we feel confident in the outcome.

Discussion

The iterative development process was used to create circle. Once use cases were defined, we developed a few of them to learn about the technologies at play and hone our proposed model of asynchronous interaction. We then created a fully functional prototype to test out these agile principles. Subsequently, a shared library was established and the prototype was refactored with domain driven design. Part of the project was rethought and redone in accordance with software quality best practices. The current version of our proof of concept is the result of our most recent refactoring.Here we'll go over three major considerations that arose during circle's creation.Decisions about the implementation and design of "socialised services" are central to understanding the evolution of this concept.

However, choosing a number of technologies was a necessary part of developing circle, a "socialised architecture" reference application. The selections had to take into account the potential design benefits of each option. The most intriguing technologies chosen for circle are detailed.

Library

We learned the importance of code sharing between system components while building the first prototype. We discovered some ubiquitous domain objects, in particular the classes responsible for facilitating interaction between services. For this reason, we developed the circle core library, which is used by every service, including the API gateway. There are three distinct levels to their content:

- **Layer of domains:** Application relevant domain-specific objects. One possible classification represents credentials, for instance.

- **Layer of applications:** Services logic is encapsulated in classes. Examples include command, query, and event objects.
- **The foundational system:** Classes controlling service to service communication and encapsulating the use of AMQP.

Using a shared library is a smart design choice that will cut down on unnecessary repetition of code throughout the project. Circle's implementation can be found at https://github.com/Pitazzo/circle. The idea is that "socialised services" can reuse the infrastructural layer, while the implementation of the other layers can be seen as guidelines for designing the application's unique requirements.

GraphQL

Facebook's GraphQL is an API design pattern that aims to succeed REST by fixing its flaws. As a query language for a server provided data model, GraphQL runs on HTTP and has a more robust interaction with web services than REST. GraphQL provides three distinct means of communication:

- **Inquiries are used:** The server allows for compound queries that contain sub queries.
- **By means of mutations:** The internal state of the server is modified by these actions. They have the same concatenation and parameterization capabilities as queries.
- **Utilising memberships:** They're not quite queries, but they're a unique data format all the same. However, information can be dynamically updated by them. By way of illustration, if you sign up for a web service for a "publicly traded" asset, you will always know what that asset is currently worth. Circle employs GraphQL as its API gateway communication protocol.

GraphQL was selected because of its semantic richness and because of its compatibility with the Command Query Responsibility Segregation (CQRS) design pattern. First, the pattern driven structure separates commands from queries. Subscriptions, on the other hand, improve its compatibility with CQRS. CQRS complicates the management of API clients because it offers no alternative for handling asynchronous responses to orders. Adding "polling" mechanisms is a common workaround that often enhances the user experience. The term "polling" refers to the practice of triggering the service at regular intervals in order to create a false sense of synchrony. From a design perspective, this solution is adequate but lacks efficiency and elegance. Subscriptions in GraphQL, however, provide the following workaround. Subscribing to a query in GraphQL that promises the completed result of a command and then receiving it synchronously does not contradict CQRS principles. To achieve this, however, we need to make use of what are called "domain events." The completion of an order's execution then causes the publication of a "domain event" communicating this information. The API Gateway will be alerted to the arrival of this event and will then extract the payload result. When the API Gateway receives the notification, it will update the client's subscription, which was created after the command was sent. Therefore, "polling" mechanisms are unnecessary, as the client can simply wait for the command's result. Figure 4 illustrates the described mechanism.

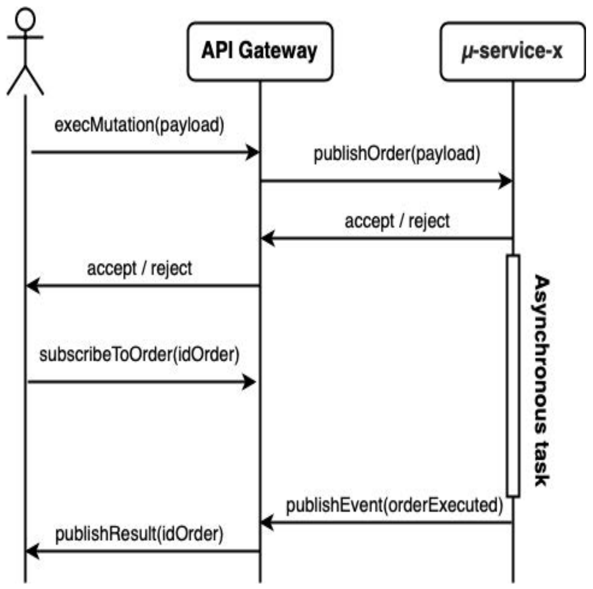

Figure 4. API gateway and client interaction, using GraphQL subscriptions.

Containers and security

A "socialised application" should have its dependencies, configurations, and code compressed to ensure that it runs equally well regardless of the environment and without any intervention from the user. You can run multiple copies of the system with just one shell command using containers. Docker, a widely adopted and relatively lightweight virtualization solution, is what circle employs to host all of its services. The Figure 5 commands can be used to initiate these services. The images will begin running immediately after they have been downloaded. Without any intervention from the user, they are kept in sync with the broker and the database. One service at a time, one copy of each service, or multiple copies of the selected services can be run by the user. Figure 6 shows the system in its deployed state.

```
Docker run -d pitazzo/circle:users-service

Docker run -d pitazzo/circle:content-service

Docker run -d pitazzo/circle:notifications-service
```

Figure 5. Commands for executing system services.

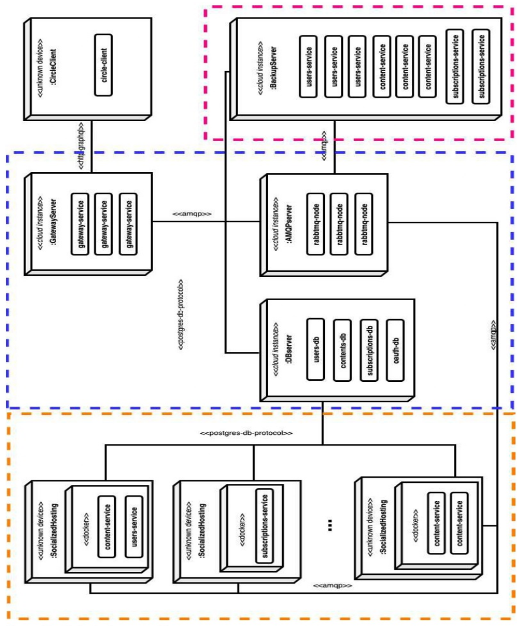

Figure 7. *Circle* deployment view.

Figure 6. Circle deplyoment view.

Using external infrastructure raises security concerns for the proposed deployment model. In fact, we're handing over the source code along with the necessary credentials and business logic. We first suggested obfuscation, or a cryptographic rewrite of the code. In this way, the code can be successfully executed but cannot be obtained or modified *via* reverse engineering. Regarding the credentials, bit wise operations are applied to the literals in the code representing them. It is not recommended practice to incorporate credentials into the code. However, we suggested that in order to employ obfuscation. Good practice dictates that they be stored in a dedicated configuration file alongside the code. At compile time, we used an external file to automatically replace the literals. Obfuscation may serve as a stopgap measure, but it's not enough for widespread use in the industrial sector.

Conclusion

The main contributions of this work are:

- An unrestricted plan for breaking the cloud monopoly, based on software engineering concepts like patterns and asynchronous interactions.
- The creation of a functioning social network that demonstrates most of the theoretical underpinnings and existing issues.
- Despite the fact that a lot of work was put in from both ends, we know that a lot more is needed to make the "socialised architecture" a reality. Consequently, the open innovation field mentioned in the introduction may be an appropriate means of speeding up the suggested ideas as inputs for businesses hoping to increase internal innovation. We address some outstanding, crucial concerns below.
- One of the biggest problems with "socialised architecture" is figuring out how to divide up the persistence. While our work is motivated by a desire to reclaim "data sovereignty," we must rely on cloud services to demonstrate the viability of our proof of concept.
- Designing for user authentication in a "socialised architecture" is a pressing concern. Please provide your responses to the following inquiries. Where, between the API Gateway and the services, should authentication take place? Is it necessary to keep the authentication service private?Moreover, what does this mean?
- At present, as indicated by CQRS, order parameters are validated by the services that receive them. However, incorporating the API Gateway into the procedure would be exciting. There may be two benefits to this. First, response times would increase because there would be no downtime between services. Second, since the broker can persist services, there would be no need to have them ready to process a message immediately upon its arrival. As a result, the system was able to process requests from clients without regard to the availability of replicas.
- Nested GraphQL queries are slow, as discovered in section 4.5 of the prototype evaluation. It would be intriguing to find a way to automatically parallelize such queries, or to provide design guidelines to reduce the number of instances in which such parallelization is not possible.
- Future lines of work must necessarily address the previous pending issues. Concretely:
- Manage distributed transactions and consensus algorithms in a conventional fashion, as in Ongaro, D, and "socialise" the persistence alongside the replicase
- We can use block chain as a distributed ledger and other decentralised hosting solutions like the interplanetary file system to achieve this.
- According to Tahirkheli, A.I, authentication isn't the only security issue that needs to be addressed.
- Another area where development is needed to guarantee the long-term reliability of "socialised services" is fault tolerance.

Finally, it goes without saying that our proposal is inferior to state of the art cloud services in terms of speed, safety, and dependability. The key takeaway from this study, however, is the realisation that an alternative to the current cloud exists and has the potential to grant the internet greater autonomy.

References

1. Florea, D., & Florea, S. "Big data and the ethical implications of data privacy in higher education research." *Sustainability*. 12.20 (2020): 8744.
2. Antonio, A., & Tuffley, D. "The gender digital divide in developing countries." *Future Int*. 6.4 (2014):673-687.
3. Columbus, L. "83% of enterprise workloads will be in the cloud by 2020." (2018).
4. Antova, L., et al. "Rapid adoption of cloud data warehouse technology using datometry hyper-Q." In proceedings of the 2018 international conference on management of data.(2018):825-839.
5. Vargas, C. "Cloud market share report: Aws *vs.* azure *vs.* google cloud 2019: Mcafee." (2020).
6. van der Meulen, R. "Understanding cloud adoption in government." (2018).
7. Faragardi, H. R. "Ethical considerations in cloud computing systems." Proceedings, IS4SI 2017, 12-16 June 2017, Gothenburg, Sweden. 166.
8. Chesbrough, Henry W. "The era of open innovation." Managing innovation and change. 127.3 (2006):34-41.
9. Baierle, I. C., et al. "Influence of open innovation variables on the competitive edge of small and medium enterprises." *J Open Innov: Technol Mark Complex*. 6.4 (2020):179.
10. Docker, I. "Docker: Empowering app development for developers." (2020).
11. Mengistu, T.M., & Che, D. "Survey and taxonomy of volunteer computing." *ACM Comput Surv*. 52 (3) (2019):1-35.

12. Durrani, M. N., & Shamsi, J. A. "Volunteer computing: Requirements, challenges, and solutions." *J Netw Comput Appl*. 39 (2014):369-380.
13. Facebook. GraphQL—A Query Language for Your API. 2012–2021.
14. Kirby, G., et al. "An approach to ad hoc cloud computing." *Arxiv Preprint Arxiv*. 1002. (2010):4738.
15. Che, D., & Hou, W. C. "A novel "Credit Union" Model of cloud computing." Digital information and communication technology and its applications: International conference, DICTAP 2011, Dijon, France, June 21-23, Proceedings, Part I. Springer Berlin Heidelberg, 2011.
16. Mengistu, T., et al. "A "no data center" solution to cloud computing." 25-30 June 2017, Honololu, HI, USA, 2017 IEEE 10th international conference on cloud computing IEEE. (2017):714–717.
17. McGilvary, G. A., Barker, A., & Atkinson, M. "Ad hoc cloud computing." 27 June 2015-02 July 2015, New York, NY, USA, IEEE 8th international conference on cloud computing. 2015: 1063-1068.
18. Ryden, M., et al. "Nebula: Distributed edge cloud for data intensive computing." 11-14 March 2014, 2014 IEEE international conference on cloud engineering. Boston, MA, USA. (2014): 57-66.
19. Babaoglu, O., Marzolla, M., & Tamburini, M. "Design and implementation of a p2p cloud system." Proceedings of the 27th Annual ACM Symposium on Applied Computing. (2012).
20. Beberg, A. L., & Pande, V. S. "Storage home: Petascale distributed storage." 26-30 March 2007, 2007 IEEE International Parallel and Distributed Processing Symposium, Long Beach, CA, USA.(2007):1-6.
21. Qin, A., et al. "Fatman: Building reliable archival storage based on low ost volunteer resources." *J Comput Sci Technol*. 30 (2015):273-282.
22. Neumann, D., et al. "STACEE: Enhancing storage clouds using edge devices." Proceedings of the 1st ACM/IEEE workshop on Autonomic computing in economics. (2011):19-26.
23. Al Noor, S., Hossain, M. M., & Hasan, R. "SAS cloud: Ad hoc cloud as secure storage." 08-10 October 2016, In Proceedings of the 2016 IEEE international conferences on Big Data and Cloud computing (BDCloud), Social Computing and networking (SocialCom), Atlanta, GA, USA 2016: 8-10 October. (2016).
24. Mohaisen, A., et al. "Social cloud: Using social networks for building distributed computing services." *Arxiv Preprint Arxiv*. 1112.2254 (2011).
25. Chard, R., Bubendorfer, K., & Chard, K. "Experiences in the design and implementation of a social cloud for volunteer computing." IEEE 8th International Conference on E-Science, 8-12 October 2012, Chicago, IL, USA (2012).
26. Caton, S., et al. "A social compute cloud: Allocating and sharing infrastructure resources *via* social networks." IEEE transactions on services computing, 28 January 2014. 7.3 (2014): 359-372.
27. Kuada, E. "A social network approach to provisioning and management of cloud computing services for enterprises." (2014). 25-30.
28. McMahon, A., & Milenkovic, V. "Social volunteer computing." *Int J Syst Cybern Inform*. 9.4 (2011):34-38.
29. Anderson, D. P. "Globally scheduling volunteer computing." Future Int. 13.9 (2021): 229.
30. Xu, L., et al. "Task assignment algorithm based on trust in volunteer computing platforms." *Information*. 10.7 (2019):244.
31. Waheed, A., et al. "Volunteer computing in connected vehicles: Opportunities and challenges." IEEE Network. 34.5 (2020):212-218.
32. Cao, Xiaowen, et al. "Joint computation and communication cooperation for energy efficient mobile edge computing." *IEEE Int Thin* J. 6.3 (2018): 4188-4200.
33. Buschmann, F., et al. "Software Patterns." (1996).
34. Richards, R., & Richards, R. "Representational state transfer (rest)." Pro PHP XML and web services. (2006):633-672.
35. Tanenbaum, A. S. "Distributed systems principles and paradigms." (2007).
36. Taylor, H. "Event-driven architecture: How SOA enables the real time enterprise." Pearson Education India, 2009.
37. RabbitMQ, A. "Messaging that just works-rabbitmq." (2020).
38. Newman, S. "Building micro services." Designing fine grained systems, 1st edition. O'Reilly Media: Newton, MA, USA, 2015; 280.
39. Kumar, A. "Cqrs (Command Query Responsibility Segregation)." Independently Published (2019).
40. OASIS. AMQP-Advanced Message Queuing Protocol. (2020).
41. Evans, E. "Domain driven design: Tackling complexity in the heart of software." Addison-Wesley Professional. (2004).
42. Ongaro, D., & Ousterhout, J. "In search of an understandable consensus algorithm." 2014 USENIX annual technical conference. (2014):305-320.
43. Chen, Yongle, et al. "An improved P2P file system scheme based on IPFS and block chain." IEEE international conference on big data (big data), 11-14 December 2017, Boston, MA, USA. (2017):2652–2657.
44. Tahirkheli, A. I., et al. "A survey on modern cloud computing security over smart city networks: Threats, vulnerabilities, consequences, countermeasures, and challenges." *Electronics*. 10.15 (2021):1811.
45. Mysliwiec, K. "NetsJS-A progressive Node. js framework for building efficient, reliable and scalable server-side applications." (2017).
46. OpenJS foundation. "NodeJS." (2020).
47. Microsoft. "TypeScript-typed javascript at any scale." (2012).
48. Alistair, C. "The pattern: Ports and adapters (Object structural)." (2020).
49. Open Source-Supported by Sponsors. "TypeORM object relational mapping." (2020).
50. Fowler, M. "Patterns of enterprise application architecture: Pattern Enterpr Applica Arch." Addison-Wesley. (2012).
51. Obe, R. O., & Hsu, L. S. "PostgreSQL: Up and running: A practical guide to the advanced open source database." O'Reilly Media, Inc. (2017).

Cite this article: Kumar R. "A Novel Cloud Computing Software Engineering Strategy Based on "Socialised Architecture". Int J Innov Res Sci Eng Technol, 2023, 4(2), 1-10.