Integrating Choreography and Orchestration in a Microservice based Gas Cylinders Tracking Model

Muriithi B. Nyagaki
School of Computing & Informatics
University of Nairobi, Kenya

Agnes N. Wausi
School of Computing & Informatics
University of Nairobi, Kenya

ABSTRACT
Microservice architecture is a style that structures an application as a composition of small services developed, and deployed independently. Each service performs a specific business function. The use of microservice architecture comes with benefits of scalability, less coupling, interoperability but also issues of distributed systems such as discovery, communication and coordination of the services.
In this study, microservice design patterns for decomposing, deploying and coordinating services are explored. Choreography and orchestration mechanisms for coordination of services are looked into and tradeoffs established for an efficient application.
A gas cylinder tracking model is developed using identified microservice design patterns. Agile software development methodology was used as it is light weight with iterative stages; Requirements analysis, design, development, testing, deployment and review.
The model performance evaluated using load balancing metrics show that the system is stable under high loads provides dynamic provisioning of computing resources creating more instances of a service distributing the load. A challenge encountered in the use of different tools for service composition and testing that increased the complexity of the application. Adoption of this prototype is recommended for use since it can track and tracing cylinders in the supply chain across different platforms.

General Terms
Distributed technology, microservice, choreography, and orchestration.

Keywords
Microservice, choreography, orchestration, gas cylinder tracking

1. INTRODUCTION
Gas cylinders tracking systems have been used world over by manufacturers, distributors and retailers. The systems are used for the identification of the cylinders, real time monitoring of the location of the cylinders while on transit, monitoring stock level trends for the purpose of restocking and compliance to regulations and safety of use.
Technologies for identification of the cylinders include Radio Frequency Identification (RFID), bar code and QR codes. Global Positioning Systems locate the position of a cylinder in trucks when on transit [1]. Sensor and RFID based solutions detect gas cylinder leakage and enable automatic booking of cylinder when empty. Tracking of cylinders down the supply chain entails integration of many players such as suppliers, manufacturers, retailers and customers. [2] Presents the use of RFID and IoT to track items down the chain.
The systems in use for tracking gas cylinders are mainly built on monolithic architecture as a single block of code. Modules in the code are tightly coupled leading to issues of scalability and deployment and integration with other systems [3]. The need for autonomy of application has led a move from monolith to Service Oriented Architecture (SOA) then microservice.
In this study, microservice design patterns applied are domain driven design for decomposition, database per service deployed in its own container. Choreography used for service to service integration and a single entry point for client requests at the Application Programming Interface (API) gateway. The patterns were integrated to develop a microservice based gas cylinders tracking model for application in the Kenyan context.

2. LITERATURE REVIEW
In this chapter, a review is done on microservices based systems in four areas: design patterns for splitting of a monolith to a set of functional microservices, services composition patterns, deployment and evaluation of performance. A detailed analysis on the integration of composition patterns is done and how it can improve the performance of microservices.

2.1 Review of Microservice
Microservice is an architectural style for distributed software that structures an application as a set of small application called service which runs its own process. Individual services are developed and deployed in separate environments and communicate using lightweight mechanisms mainly HTTP or REST [4].
Each service performs a specific business functionality and the deployment in environments are mainly containers. Communication between clients and the services are managed by use of Application Programming Interface (API) gateways thereby providing an interface for each client to route requests, transform protocols, and perform authentication, monitoring, and static response handling [5].
[6] For communication to take place, services discover each other through an IP addresses and a port assigned during registration of the services. It can happen at the client end where a client identifies location of a service instance from the registry to send a request, or at server side where client sends a request through a load balancer responsible for querying a service registry then routes the request to the available service instance [4].

2.2 Decomposing microservice application
A monolith is an architectural style where a system is wrapped up into a single application with functionalities as modules. The modules are coded using a single technology stack and communicate through function calls. A module consists of classes, functions, and namespaces. The application is easy to develop, deploy and test but as it grows it becomes complex and hard to maintain. Dependencies between modules makes it difficult to effect changes without affecting the entire application [7].
As a solution to the challenges of a monolith, SOA segregates the functions of a monolith into loosely coupled and coarse-

grained web services. A web service is self-contained software performing specific business functionality and is accessible over a network. An Enterprise Service Bus (ESB) integrates web services to form a composite service. Microservice is a case of SOA with fine grained and autonomous application.
Patterns for decomposing a monolith into a set of microservices patterns are: Domain Driven Design, Business Capability, Static Analysis or dynamic analysis.
Domain Driven Design approach identifies sub domains of the business logic and then constitutes a deployable unit called a bounded context that performs a single business functionality [8]. Business Capability pattern entails identifying business functions that can translate into a single responsibility function which becomes a service. Static Analysis requires project source code as input and analyzes coupling between software classes. Dynamic analysis is done at runtime where functionalities of a system are analyzed using execution traces to cluster source code entities with similar functionality as a service [9]. Decomposing based on business capability leads to stable services with less dependencies easy to develop and scale independently.

2.2 Service Composition patterns

To combine the independent services into a single application, two techniques orchestration and choreography are applied. Orchestration uses a central coordinator known as an orchestrator. It listens to all the events produced by each service transaction and triggers a local transaction in a different micro service. It holds in waiting until it gets a response which is transformed and routed to the next service. The orchestrator coordinates communication and workflows between services [10].
Orchestration patterns are: SAGA where a central coordinator sends messages instructing each service what to do and the order of execution; use of an API gateway utilized as an orchestrator where the clients require a streamlined and consistent interface for interacting with other services. This simplifies client interactions by combining responses from multiple services [4] .
Choreography on the other hand is an event driven approach for managing the interactions and workflows between microservice. Each service knows what events to react to since the logic coordination is built within a service. The events generated by services are coordinated through an event bus or a message broker that acts as a dumb pipe as all the resources and function logic is encapsulated within each service. Other services can consume the same event, process or publish their own events back into the event bus. The publisher of the event does not know anything about consumer service but the consumer is aware of which event to listen to trigger its local transaction [10].
In orchestration the central coordinator brings service dependencies hence less autonomy compared to choreography and less resilient. An orchestrator must wait for a response to continue with the next request hence more latency. It manages the workflows and error detection is easier.
Service coordination in choreography is asynchronous and requires implementation of models to ensure consistence and complex. Low interdependency of services enable easy scaling.

Table 1: Comparison of Orchestration and Choreography

	Orchestration	Choreography
Autonomy	Low	High
Complexity	Low	High
Resilience	Low	High
Scalability	Low	High
Latency	High	Low
Error handling	Easier	Harder

2.3 . Microservice Deployment

The environment for deploying services in the cloud can be Virtual Machines (VMs) or containers. These are virtualized environments in the cloud that allow for isolation of services and for elasticity in the cloud. The VMs focus on hardware virtualization that deals with hardware allocation and management which has the limitation of entangling executables, configuration, libraries, and other dependencies of an application with the underlying host operating system. Containers are packages of software that virtualizes the operating system and runs applications [11].
An evaluation of VMs and containers by [12] shows that containers are preferred for microservices deployment. They run on the host operating system sharing libraries reducing their image sizes. They are light in weight, occupy less space and take less time to deploy. Updating services in containers take less effort and low downtime. Rolling updates require downloading and installing libraries which takes less time as containers share libraries. The lightweight nature also enables many containers to run on a single machine leading to better resource utilization [13].
In a microservice application, each service instance is a process that runs on multiple machines or containers. Communication takes place between a client and a service or between services using direct calls to services, through a gateway or using a message or service bus.

2.4 Evaluation of performance

The performance of microservices is assessed using metrics such as time taken, complexity and load test using choreography and orchestration, [14] establishes that choreography is a good approach when there are less micro services participating in the distributed environment. It is also appropriate when an application or software requires constant updates and addition of new features because it does not interrupt existing events and processes. However, if multiple events are triggered from a transaction, it becomes complex to code and handle event choreography. Though slow, orchestration is suitable when transaction scenarios are complex.
An analysis of orchestration and choreography performance using metrics of time, memory and power consumption by [15] shows that choreography is fast in performance in case of few events though complex to code if multiple events are triggered from each microservice.
Load based tests using quantitative analysis done on the microservices by increasing the frequency of events triggered, shows that choreography responds slowly to high loads, but an orchestrator is able to handle the increased load in a better way.
An empirical study on microservices performance done using tools such as Junit, Jmeter, and Mocha show that load, resource usage, availability and connections in a database are the most common metrics of measuring microservices performance. A systematic Literature Review [16] establishes the frequency of various metrics used to evaluate the performance of microservices are Latency, CPU usage, throughput, network, scalability, memory and input/ output from the highest to the least frequent respectively.

3. METHODOLOGY

3.1 Research Design

Evaluation research design was used as it aims at assessing the effectiveness, efficiency, relevance, and impact of programs,

policies, interventions, or projects, providing evidence-based insights to support informed decision-making, improve practices, and enhance outcomes [17]. A case study of Energy and Petroleum Regulatory Authority (EPRA) done involved qualitative data collection using questionnaires and document analysis to gather rich, detailed information.

EPRA is a state corporation established under the Energy Act, 2019 whose mandate is to regulate petroleum, electricity and renewable energy sectors in Kenya

EPRA has embraced use of technology to perform its mandate as a regulator. It uses an online portal for application of licenses for new applicants and renewals of gas cylinder dealers. The current system in EPRA is monolithic with no module for gas cylinder tracking. This project demonstrates the use of microservice architecture applying both the orchestration and choreography patterns for the EPRA case study guided by the research question: How to integrate appropriate microservices design patterns in a gas cylinder tracking system?

3.2 Development Methodology

Agile software development methodology used is a lightweight approach for software engineering that starts with the planning phase followed by iterative and incremental interactions till the deployment phase. [18] Cites that cloud computing relies on services made up of small parts developed, tested and maintained separately hence the need for a lightweight approach and hence the suitability of the agile software development cycle.

3.3 Requirements analysis

[17] Refers to the target population as a group of individuals from whom data is to be collected. The population target in this study included EPRA staff, brand owners, distributors and retailers of gas cylinders who are part of the primary data collection.

Simple random sampling used for brand owners, distributors and retailers of gas cylinders as it gives every individual in the population an equal chance of being selected [17]. Purposive sampling was used to select 30 EPRA staff targeting individuals with knowledge of information systems.

3.3.1 Sample size

Sample size refers to the number of participants to be selected from the entire population as a sample [17]. A confidence level of 80% used to calculate the sample with a margin error of 0.05% using the Cochran's formula:

$$n = \frac{z^2 p(1-p)}{e^2}$$

$$n = \frac{1.282^2 0.5(1-0.5)}{0.05^2} = 164$$

A confidence level of 80% was used due to resources and time constraints for this study as it resulted in a small sample size [19].

3.4 Data Collection Procedure

3.4.1 Questionnaire

A set of questions typed or printed in order to collect data from the respondents [17]. Questionnaires filled in the sample survey with brand owners, distributors and retailers and EPRA staff. A questionnaire was selected as the primary data source as the respondents were distributed in different geographical locations.

3.4.2 Document Review

The documents reviewed show an estimated 3,500,000 active LPG cylinders in circulation in Kenya, an estimate of 70 licensed LPG brand owners, 500 gas cylinder refilling depots and approximately 7000 retailers.

3.5 Requirements Analysis

The research findings were represented using tables, graphs and charts. Two sets of questionnaires were developed, one for EPRA staff and one for players in the supply chain. A total of 164 questionnaires were administered to brand owners, distributors and retailers selected only from Nairobi County, out of which 117 responded: 6 brand owners, 10 distributors and 101 retailers giving a 71 % response rate which is considered to be an adequate response rate [20]. 30 questionnaires were distributed to EPRA staff and 21 responded. Data analysis was done using tables and graphs to come up with the most suitable requirements. The functional requirements were then modeled with system flowchart for the microservice application.

The respondents identified the major challenges experienced in the tracking of gas cylinders in the supply chain shown in table 2.

Table 2. Responses on Challenges Faced in Distribution Gas Cylinders

Challenge	Yes	No
Loss of gas cylinders	96%	4%
Cross filling of gas	80%	20%
Illegal gas refilling	89%	11%
Unclear cylinder serial numbers	67%	33%
Poor tracking of the cylinders	96%	4%

The respondents agreed that a comprehensive system should incorporate various technologies for tracking individual cylinders when on transit as well as when sold to the consumer, data management and security of the system as shown in table 3.

Table 3. Responses on Technologies for a tracking system

Item	Yes	No
QR code for tracking a cylinder	93%	7%
Sensors to monitor delivery	83%	17%
Cloud based for data management	90%	10%
Block chain for security	84%	16%

An average of 89% respondents suggested that use of cloud computing would lead to faster data processing and high data security and enhance accessibility on different platforms due to integration with other systems as shown in fig 1.

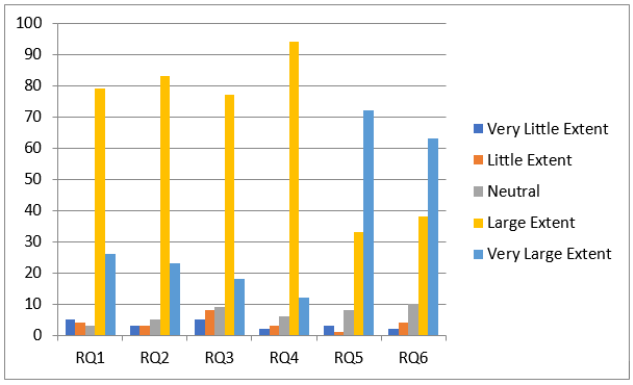

Fig 1: Use of Cloud Based Technology in Gas Tracking

RQ1: Use of cloud computing leads to faster data processing
RQ2: There is security of data since there is controlled access of data by the users

RQ3: Enhanced accessibility since it can be used on different devices such as computer or a Smart phone.
RQ4: Cloud –based systems can be integrated with other systems.
RQ5: A tracking system will impact customer satisfaction and service quality.
RQ6: Cloud technology handles data storage and backup efficiently.
The participants were asked on the effect of microservice application on gas cylinder tracking in the scale (1- Very little extent, 2- Little extent, 3- Neutral, 4 – Large extent, 5 –Very Large extent). An average of 82% agreed to a large extent table 4.

Table 4. Responses on the Effect of Microservice Application on Gas cylinder tracking

Effect	1	2	3	4	5
Simplify development, deployment and scaling	4%	6%	8%	58%	24%
Ensure maximum utilization of resources under high load	8%	6%	10%	46%	30%
Integrate new technology without affecting entire system	4%	2%	6%	56%	31%
Isolation of failures	6%	6%	8%	50%	29%
Integrate with other systems with minimum downtime	6%	6%	6%	42%	40%

3.6 System Design

Domain driven design technique used to identify the services required in the application as it defines clear boundary for each service.

3.6.1 High Level Design

The gas cylinder tracking system is cloud based. New gas cylinders are passed to the brand owner for serialization and branding. They are sealed and then passed to the refilling station. Inspection is done to the cylinder and gas inside either on premise or out of premise. If the cylinder is ok after inspection, they are taken back for distribution, then retailer and finally to consumer. The consumer can scan the details of the cylinder through a public API in the cloud.

3.6.2 System Architecture

Gas cylinder tracking system is a microservice based application deployed in the cloud. System is made of entity manager service and cylinder service Fig 5.
Entity Manager Service: Manages user authentication, registration, and profile management of brand owners, distributors and retailers. It exposes RESTful APIs for user-related operations. It allows the system administrator to create user accounts and register brand owners, distributors, retailers and refillers.
Cylinder Service: Handles cylinder and inventory tracking and pricing. It integrates with the main service through asynchronous messaging.
Each LPG cylinder registered on this service has a unique serial number that's generated during registration of new cylinders. A cylinder also has a QR code that contains information about serial number, brand and other information about the cylinder. With a QR code scanner, anyone can scan and get the cylinder's information, for verification of its authenticity. With the serial number obtained from the QR Code, the cylinder can be retrieved from the portal using a registered and an authenticated account. The system then displays detailed information about the cylinder including all the events that have taken place against it, and by which registered entities.
RabbitMQ enables choreography by allowing for events generated cylinder service are consumed by the entity manager service. It is a message broker providing asynchronous communication for handling events such as manufacturing, refilling, cylinder returns and sales updates and notifications.
Users access the application using a client portal coded using React Js. The LPG Cylinder management portal allows various parties authorized by EPRA to access LPG cylinder services, like tracking cylinders, updating cylinders through events like refill, sale and returns. To achieve this, it limits actions based on user's accounts which implement access control to the various services.
The user requests pass through a load balancer that distributes load and routes the requests to either the entities manager service or the cylinder service. An API gateway is used to combine data from several microservices into a single response for a client request. It translates different protocols and data formats for user requests and responses. It authenticates tokens, verify user access rights based on access control policies. The services are created using Django and REST frameworks.
Database per Service method is used for data management. Entity manager service has its own database created using PostgreSQL while the cylinder database is in MySQL.
Docker containers are used to package each microservice for containerization and deployment.
Security is enforced through user authentication and authorization using JWT tokens issued by the services.

3.7 System Testing

Load testing was done using locust tool. The test parameters were user behavior (number of users). Locust tool simulates users as in a real running environment.

4. RESULT AND DISCUSSION

This research aimed at identifying microservices design patterns to develop a micro service-based gas cylinder tracking application. To create a robust microservice application, design patterns were carefully considered in the decomposition, data management, service discovery, communication, deployment and evaluation.

Table 5. Microservice patterns

Process	Design Patterns
Decomposition	Domain driven Design [21] Static Analysis [22] Dynamic analysis [23]
Data management	Database per service Shared database [24]
Communication	Synchronous Asynchronous [25]
Service Composition	Orchestration Choreography [10]
Service Deployment	Containers Virtual machines [13]
Evaluation metrics	Load balancing Use of resources-RAM, Processor Latency Throughput [13], [16], [26]

Domain driven design pattern was used to decompose the services based on their functionality as it is appropriate for defining services through analysis of business logic [27], [10]
The choice of data management pattern is determined by the performance or data persistence of a system. Database per service pattern used in this model as it ensures loose coupling hence autonomy of services [24].
Two microservices Entity manager service and Cylinder service were deployed using Docker containers because of their lightweight. They take less effort and low downtime when updating or rolling. Many containers can run on a single machine with better resource utilization [15].
Services crated in Django and Django Rest framework due to its compatibility with React.js for front end user interface Fig 4. Events generated by the services are:

1. Manufacturing <> At the Brand Owners' domain
2. Refill <> At the Refiller's domain
   (Implications: Inspection status: Good or Out of service)
3. Sale <> At Brand owner, Distributor, or Retailer's domain
   * UI is expecting sale event to mark the seal code as invalid & cylinder empty to allow for returns *
4. Cylinder Returned (from customers) <> At Brand owner, Distributor, or Retailer's domain
   Implications: seal bar code should already be invalid & cylinder should already be empty.
5. Refilling: Repeat cycle 2-4 for the usable lifetime of the cylinder.

Orchestration and choreography service composition patterns can be integrated to achieve the strengths of both [10]. In this hybrid collaboration pattern, Choreography occurs through RabbitMQ a message broker for the service. An API gateway orchestrates the responses of requests from different services and sends to client.
One of the limitations of a monolith occurs when scaling due to high load. It requires adding more computing resources by redeploying the entire codebase increasing the downtime. An evaluation of this model show that the system is stable with high loads. Two tests done, at first 1 server node and 1 database host simulated 100 users making random requests on the backend service. In the second test, 1 server node, 1 database host simulated 200 users making random requests on the backed service. Server started throttling requests because of too many connections in the database connection pool. Database engine also complained of too many connections affecting its performance but after adjusting by increasing the maximum number of connections allowed, the system stabilized Fig 2. This means that the system is stable even when the load is high.

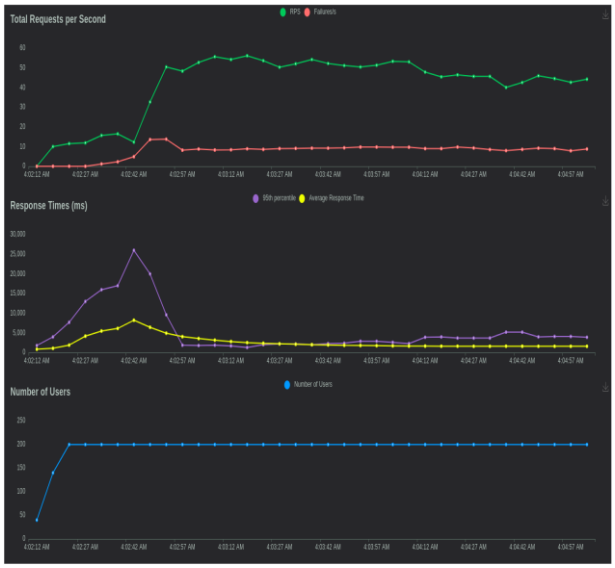

Fig 2: Performance handling spike 200 users in 2 minutes

5. CONCLUSION

The study shows that microservices architecture style can be used to solve problems of monolithic systems such as limited scalability, complexity as the system grows, adaptability to changes. The model developed and evaluated enables tracking of gas cylinders from manufacturer to retailers. It is stable under high load when being accessed by many users with less downtime as shown in the load test Fig 2. The public APIs allow for integration other systems. The system is deployed in the cloud with less infrastructure implemented on the premise where payment is made based on demand thus reducing on cost. Scaling of the system horizontally is possible due to high load and vertically by adding computing resources. Microservices dynamic provisioning enables automated deploying and scaling services based on demand, without manual intervention and with less downtime.

We envisage for the adoption and use of this system which shall in the future incorporate other services such as payments using an orchestrator tool.

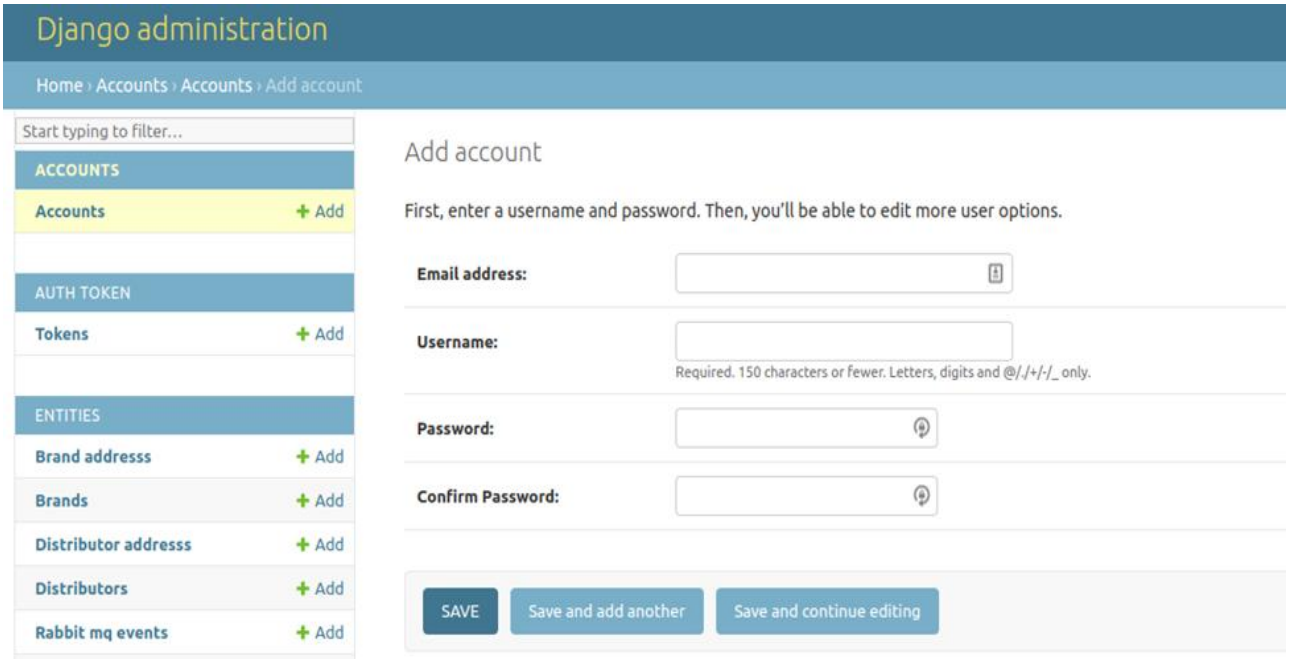

Fig 3: Entity Manager Service

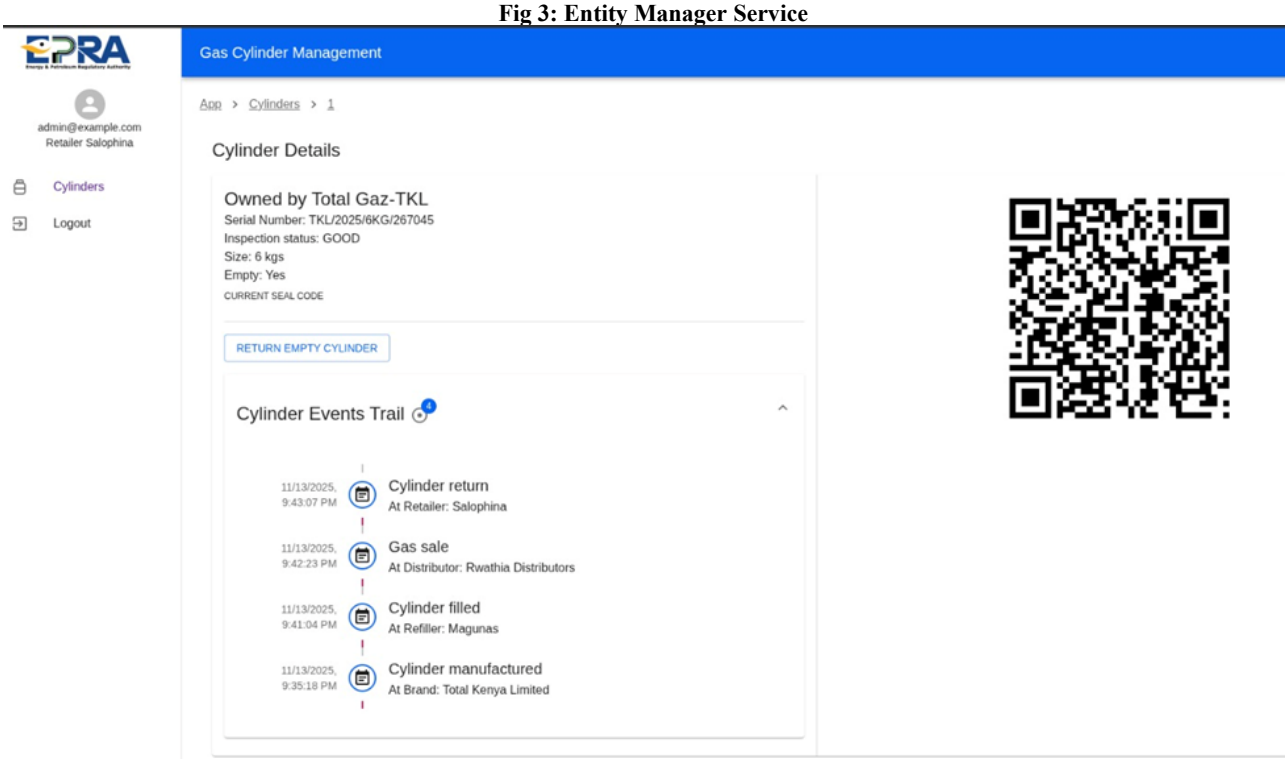

Fig 4: LPG Cylinder Management Portal

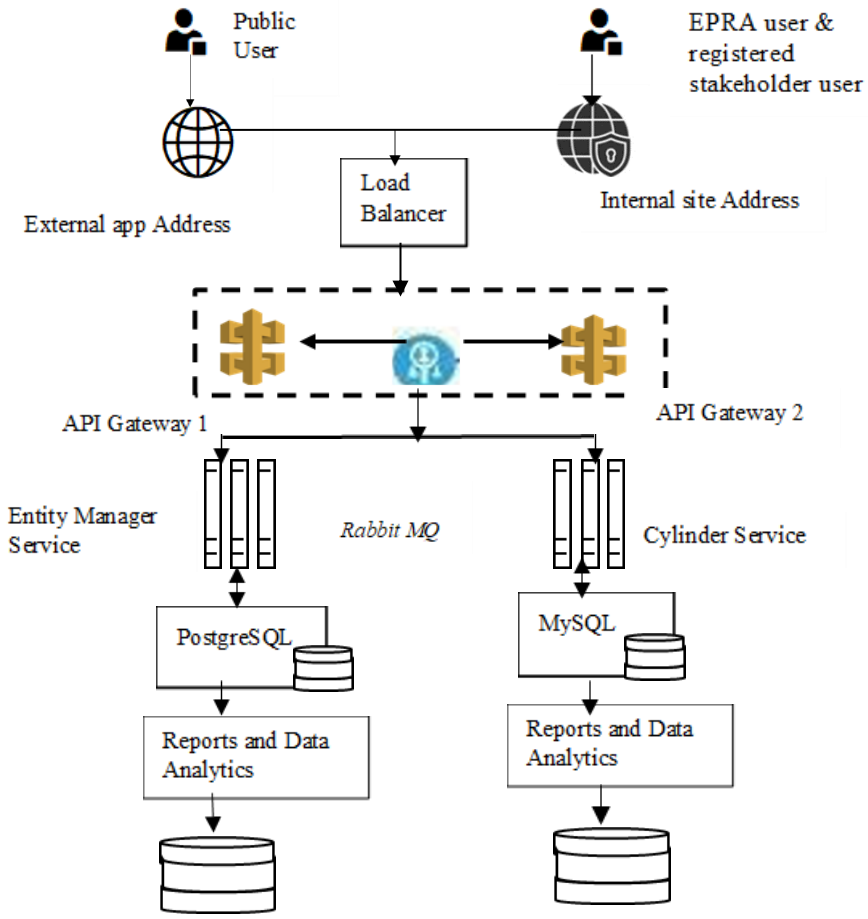

Fig 5: System Architecture

6. REFERENCES

[1] T. Rohmat , D. N. Ramadan, H. R. Sugondo and Z. Rahmana , "Fuel Truck Tracking for Real-Time Monitoring System Using GPS and Raspberry-Pi.," in Proceedings of the 1st International Conference on Electronics, Biomedical Engineering, and Health Informatics, Surabaya, Indonesia, 2021..

[2] W. C. Tan and M. S. Sidhu, "Review of RFID and IoT integration in supply chain management," Operations Research Perspectives, 2022. Fröhlich, B. and Plate, J. 2000. The cubic mouse: a new device for three-dimensional input. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems

[3] F. Ponce, G. Márquez and H. Astudillo, "Migrating from monolithic architecture to microservices: A Rapid Review," in 38th International Conference of the Chilean Computer Science Society (SCCC).

[4] J. Lewis and M. Fowler, "Microservices -a definition of this new architectural term," March 2014.

[5] P. Valderas, V. Torres and V. Pelechano, "A microservice composition approach based on the choreography of BPMN Fragments," in Information and Software Technology, 2020.

[6] N. Singh, Y. Hamid, S. Juneja, G. Srivastava, G. Dhiman, T. R. Gadekallu and M. A. Shah, "Load balancing and service discovery using Docker Swarm for microservice based big data," Journal of Cloud Computing:Advances, Systems and Applications, 2023.

[7] S. Baškarada, V. Nguyen and A. Koronios, "Architecting Microservices: Practical Opportunities and Challenges," Journal of Computer Information Systems, vol. 60, no. 5, pp. 428-436, September 2018.

[8] O. Özkan, Ö. Babur and M. v. d. Brand, "Domain Driven Design in Software Development: A Systematic Literature Review on Implementation, Challenges and Effectiveness," Journal of Systems and Software, 2023.

[9] F. Ponce, G. Márquez and H. Astudillo, "Migrating from monolithic architecture to microservices: A Rapid Review," in 38th International Conference of the Chilean Computer Science, 2019.

[10] A. Megargel, C. M. Poskitt and V. Shankararaman, Microservices Orchestration vs Choreography: A Decision Framework, MEGARGEL, Ali MADJELISI; POSKITT, Christopher M.; Venky, SHANKARARAMAN: IEEE, 2021.

[11] K. A. Işıl, Ç. Turgay, A. B. Can and T. Bedir, "Deployment and communication patterns in microservice architectures: A systematic literature review," Journal of Systems and Software, vol. 180, October 2021.

[12] G. Liu, H. Bi, Z. Liang, M. Qin, H. Zhou and Z. Li, "Microservices: architecture, container, and Challenges," in IEEE 20th International Conference on Software

Quality, Reliability and Security Companion (QRS-C), 2020.

[13] M. Waseem, P. Liang, M. Shahin, A. Di Salle and G. M´arquez, "Design, monitoring, and testing of microservices systems: The practitioners' perspective," Journal of Systems and Software, 2021.

[14] C. K. Rudrabhatla, "Comparison of Event Choreography and Orchestration Techniques in Microservice Architecture," International Journal of Advanced Computer Science and Applications, vol. 9, no. 8, 2018.

[15] N. Singhal, U. Sakthivel and P. Raj, "Selection Mechanism of Micro-services Orchestration vs. Choreography," International Journal of Web & Semantic Technology (IJWesT), vol. 10, no. 1, pp. 1-13, 1 January 2019.

[16] N. Bjørndal, M. Mazzara, A. Bucchiarone, N. Dragoni and S. Dustdar, "Migration from Monolith to Micro services: Benchmarking a Case Study.," Technical University of Denmark., 2020.

[17] R. Kothari, Research Methodology - Methods and Techniques (Second ed.), New Age International Publishers. 2004.

[18] S. Al-Saqqa, S. Sawalha, and H. Abdelnabi, "Agile Software development: Discovery from Enterprise Systems," International Journal of Interactive Mobile Technologies, vol. 14, no. 11, 10 July 2020.

[19] J. E. Bartlett and C. C. Higgins, "Organizational Research: Determining Appropriate Sample Size in Survey Research," Information technology, learning, and performance, pp. 43-50, 2001.

[20] D. D. Nulty, "The adequacy of response rates to online and paper surveys: what can be done?," Assessment & Evaluation in Higher Education vol. 14, no. 11, 10 July 2020.

[21] A. Rahmatulloh, D. W. Sari, R. N. Shof and I. Darmawan, "Microservices-based IoT Monitoring Application with a Domain-driven Design Approach," in 2021 International Conference Advancement in Data Science, E-learning and Information Systems (ICADEIS), 2021.

[22] V. Bushong, D. Das, A. A. Maruf and T. Cerny, "Using Static Analysis to Address Microservice Architecture Reconstruction," in 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2021.

[23] A. Krause, C. Zirkelbach, W. Hasselbring, S. Lenga and D. Krger, "Microservice Decomposition via Static and Dynamic Analysis of the Monolith," in IEEE International Conference on Software Architecture Workshops (ICSAW), 2020.

[24] K. Munonye and P. Martinek, "Evaluation of Data Storage Patterns in Microservices Archicture," in IEEE 15th International Conference of System of Systems Engineering, Budapest, 2020.

[25] B. Shafabakhsh, R. Lagerström and S. Hacks, "Evaluating the Impact of Inter Process Communication in Microservice Architectures," in 8th International Workshop on Quantitative Approaches to Software Quality (QuASoQ 2020), 2020.

[26] B. Goossens, "Decision Making in a Microservice Architecture," 2019.

[27] A. N. Fajar, E. Novianti and Firmansyah, "Design and Implementation of Microservices System Based on Domain-Driven Design," International Journal of Emerging Trends in Engineering Research, vol. 8, no. 7, July 2020.