Jurnal Teknik Informatika (JUTIF)
Vol. 5, No. 4, August 2024, pp. 1103-1114

DOI: https://doi.org/10.52436/1.jutif.2024.5.4.1930
p-ISSN: 2723-3863
e-ISSN: 2723-3871

IMPLEMENTATION OF STREAMLINE REALTIME STOCK USING AUTO-SCALING THROUGH GOOGLE CLOUD PUB/SUB AT PT XYZ

Joseph Heykel Prabawa[*1], Adi Nugroho[2]

[1,2]Department of Informatics Engineering, Faculty Of Information Technology, Universitas Kristen Satya Wacana, Indonesia
Email: [1]672020057@student.uksw.edu, [2]adi.nugroho@uksw.edu

(Article received: March 14, 2024; Revision: March 25, 2024; published: July 29, 2024)

Abstract

Shortening the updating and inputting of accurate and real-time stock data is crucial for smooth retail business operations at PT XYZ. The existing system had low availability, lacked scalability, and incurred high costs in managing inventory in real time. Implementing real-time stock streamline with automatic scaling and Google Cloud Pub/Sub can help achieve this goal. This system utilizes Google Cloud Pub/Sub as a message delivery platform to distribute stock information from sender to receiver in real-time. Auto-scaling is used to automatically increase or decrease the number of servers processing stock data based on demand. The system is designed using Python and integrated through libraries with the Google Cloud Platform. The results of this research prove that the system is capable of providing optimal performance and scalability with high availability, good security, and cost savings.

Keywords: *auto-scaling, Google Cloud Platform, pub/sub, real-time, stock.*

1. INTRODUCTION

The research begins with understanding the complexity of inventory management in today's business context. Conventional systems often need help with slow data updates, difficulty in handling real-time information flow, and inability to respond quickly to changes in market demand. In a rapidly changing business environment, the need for solutions that can overcome delays in data updates, limitations in real-time information flow, and the inability to adjust inventory stocks quickly has become increasingly pressing. A real example from PT XYZ is when the company encountered difficulties in keeping up with rapidly changing sales trends in their industry. The conventional infrastructure they previously used couldn't provide real-time stock information, leading to frequent stockouts of popular products or excessive inventory of less-demanded items. This resulted in decreased customer satisfaction and financial losses for the company. In this context, cloud-based technology offers a robust platform for improving inventory management with a more adaptive and efficient approach.

Adopting cloud technology by PT XYZ for real-time inventory management is a transformative step in improving operational efficiency. Previously, the company had experienced limitations from conventional infrastructure that prevented it from providing the fast response and scalability needed for inventory management. This research is necessary because it reflects the need for improved operational efficiency and competitiveness. The adoption of cloud technology, such as Google Cloud Platform, offers the company the opportunity to have a competitive advantage. By implementing streamline real-time inventory using auto-scaling through Google Cloud Pub/Sub, PT XYZ can improve flexibility, scalability, and speed in real-time inventory data processing.

2. LITERATURE REVIEW

In research titled "Design and Implementation of Data Synchronization System Using Google Cloud Pub/Sub and Flask at PT XYZ," the development of a system capable of maintaining data synchronization using the Publish-Subscribe technology provided by Google Cloud Pub/Sub and Flask is discussed. By utilizing Google Cloud Pub/Sub, data delivery will occur once, and the system will replicate data to multiple zones to ensure data/messages are delivered. This research employs the Research and Development method. Based on the research results, the system is deemed capable of synchronizing offline data to the cloud and maintaining data integrity in CloudSQL. Additionally, the Dead Letter Queue feature can assist the Backend Developers of PT XYZ in viewing and rectifying data [1].

In the previous research titled "A Cloud Pub/Sub Architecture to Integrate Google BigQuery with Elasticsearch using Cloud Functions," the discussion revolves around a cloud services architecture that integrates BigQuery with Elasticsearch through the utilization of Pub/Sub and

other cloud functions. By combining these platforms, the researchers aimed to leverage the capacity and ease of ingesting documents into Elasticsearch, its search speed, and comprehensive catalog options to create visualizations within a dashboard. This approach results in a robust and scalable application that handles large amounts of data stored in BigQuery and transfers it into analytical and visualization engines in Elasticsearch [2].

The previous research titled "A Dynamic Scalable Auto-Scaling Models as a Load Balancer in the Cloud Computing Environment" revolves around a virtual cluster architecture that enables cloud applications to scale dynamically in a cloud computing virtualization environment. This allows resources to be dynamically adjusted to meet various customer demands using a load balancer. Based on the research, it can be concluded that auto-scaling features are available in size within the sequence group, enabling users to add or automatically remove instances from managed instance groups based on load changes [3].

Based on previous research related to streamlining real-time inventory, the retail industry in the era of Retail 4.0, the use of Google Cloud Pub/Sub, and auto-scaling, a study will be conducted to discuss the implementation of real-time inventory streamlining using auto-scaling via Google Cloud Pub/Sub at PT XYZ. The expected outcome of the implementation is to assist PT XYZ in designing and optimizing infrastructure in the field of information technology so that it can perform inventory releases and updates quickly, efficiently, and with minimal issues.

Instance Group is a collection of virtual machine instances that can be managed as a single entity. Within the Compute Engine service, two types of virtual machine instance groups are managed and unmanaged. Managed Instance Group (MIG) is chosen because the software being developed requires dynamic scaling, high availability, efficient management, and cost optimization [4]-[5].

Pub/Sub is a publisher/subscriber service that acts as a flexible "middleman," enabling different applications or services to communicate with each other. This fully managed and highly scalable service runs within the Google Cloud Platform (GCP) environment. The core principle of Pub/Sub is asynchronous communication. This means that the message sender (publisher) and receiver (subscriber) do not need to be online or active at the same time for the system to function (decoupling) [6].

Automatic scaling, or auto-scaling, is a dynamic technology used in cloud computing to automatically adjust the number of computing resources based on real-time demand. Auto-scaling aims to ensure service availability and optimal application performance with effective and efficient resource utilization. Auto-scaling is required when applications experience fluctuating traffic spikes, process large volumes of data periodically, analyze data in real-time, mitigate service downtime, and save costs for business needs. By using auto-scaling, the streamline real-time stock system can automatically handle spikes or decreases in traffic, enhance resilience to failures, simplify management, and optimize costs [5]-[8].

3. RESEARCH METHODOLOGY

In the implementation of streamline real-time stock using auto-scaling through Google Cloud Pub/Sub, several stages are carried out, including problem identification, system design, implementation, deployment, testing, and report writing.

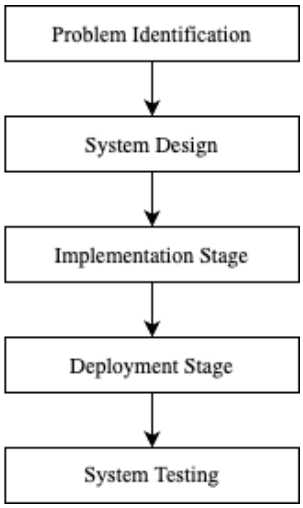

Figure 1. Research Methodology

Based on Figure 1, the research begins by identifying the problems at PT XYZ and finding solutions by implementing Streamline Real-time Stock using auto-scaling through Google Cloud Pub/Sub. In the problem identification stage, data collection and analysis are conducted according to the requests and issues brought by the users. Based on the information and data obtained, the system is designed to provide visualization of the built system. The system is designed using UML (Unified Modeling Language) as a use case diagram. It visualizes the system processes working within the Google Cloud Platform using Google Architecture Diagram. In the next stage, the implementation of Streamline Real-time Stock using Python programming language and integration with Google Cloud Platform. After the software has been developed, the system implementation results are deployed to the production environment, which is the Google Cloud Platform, by configuring and integrating the services used in the Google Cloud Platform. In the production environment, end users can use the software developed. The entire system that has been successfully built is tested before being delivered to end users. Several testing methods are used to

determine the system's performance and the functioning functionality.

4. RESULT AND DISCUSSION

The results and discussion are divided into four subsections, starting with system design using UML (Unified Modeling Language), implementation with software development, deployment to the Google Cloud Platform environment, and finally, system testing with three methods (performance testing, publish testing, and automatic scaling testing.

4.1. System Design

The system is designed to streamline real-time stock using Unified Modeling Language (UML) and Google Cloud Architecture Diagram to provide visualization of the design, architecture, and layout of the implemented service system. The use case diagram is chosen to provide visualization of the nature of the situation and explain the use case scenarios for the implementation of the streamline real-time stock system.

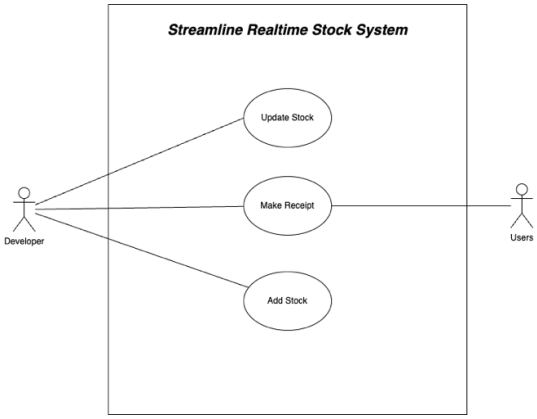

Figure 2. *Use case diagram streamline real-time stock system*

Figure 2 is a use-case diagram of the streamline real-time stock system. The diagram explains the use cases of the streamline real-time stock system between the developer and several users. The developer performs stock update operations and adds stock to the built system. The system is designed to accept receipt creation operations from multiple users. After a user has created a receipt and entered it into the system, the system will transform it into detailed stock updates or additions of new stock. The developer receives detailed stock updates and performs stock updates or adds new stock.

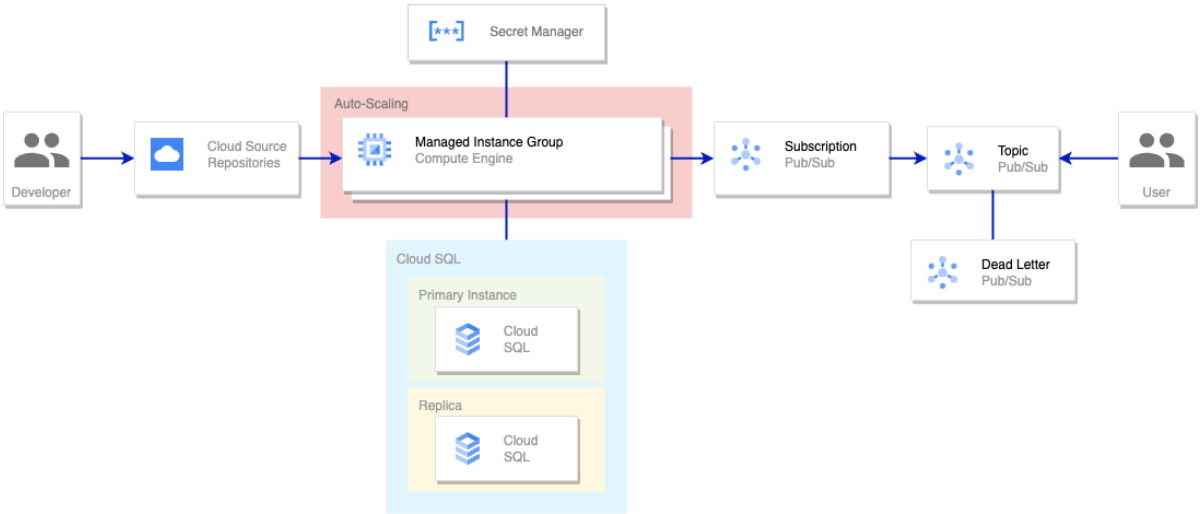

Figure 3. Google Cloud architecture diagram streamline real-time stock system

Figure 3 is the Google Cloud Architecture Diagram of the streamline real-time stock system using auto-scaling through Google Cloud Pub/Sub. The system is built within the scope of the Google Cloud Platform architecture. Several features and services of the Google Cloud Platform are utilized to run the streamline real-time stock system. The Google Cloud Architecture Diagram provides an overview and documents the system's structure built on the Google Cloud Platform infrastructure. During the development phase, developers store the implementation code of the streamline real-time stock system in Cloud Source Repositories. This fully managed Git repository management service provides securely managed Git repositories within the Google Cloud Platform environment. Users have the role of publishers, who will send messages to the system. Users interact with the application programming interface, which sends requests with data and receives responses.

The implementation code is executed within a Compute Engine instance. The Managed Instance Group ensures that the application automatically scales up or down (auto-scaling) based on the number of messages received. All configurations and credentials related to database configuration and others are stored and managed in Secret Manager, a secret management service designed to store,

manage, and secure sensitive information. Secret Manager provides database credential information to the Managed Instance Group to establish connections to the database. The application uses Cloud SQL as its database backend. Cloud SQL is a managed database service provided by the Google Cloud Platform. Data is stored in the primary Cloud SQL instance, with replicas ready for failover if needed, ensuring high availability and protection against data loss in case the primary instance fails. Pub/Sub performs message handling. Google Cloud Pub/Sub provides a communication layer separating various application parts. User actions or events can trigger messages (published to topics), which are then received by the Managed Instance Group (subscription) using a message pull scheme. The dead letter provides a mechanism to handle messages that fail to be processed after a number of retries.

4.2. Implementation Stage

From the system design results of the streamline real-time stock using auto-scaling through Google Cloud Pub/Sub, the implementation uses Python scripts with the following software design patterns: controller, service, model, interface, and implementation. The controller acts as the entry point to handle incoming requests, coordinates interactions with other components, assigns tasks to the appropriate services based on the nature of the request, and invokes services to perform specific business logic or data operations. The controller also handles receiving, preparing, and sending responses or replies.

```
subscriber = pubsub_v1.SubscriberClient()
credentials, project_id =
google.auth.default()
stock_path =
subscriber.subscription_path(project_id,
os.environ['STOCK_SUBSCRIBER'])
flow_control =
pubsub_v1.types.FlowControl(max_messages=
int(os.environ['MAX_MESSAGES']))
```

Code Program 1. Initialization of subscriber and database connection

Code Program 1 aims to create a client subscriber to interact with Google Cloud Pub/Sub, fetch authentication credentials and project identity, construct the address of the subscription created within the Google Cloud Platform using environment variables, and set up control flow settings to handle incoming messages.

Code Program 2 is a callback function designed to process each message received by the subscription. The callback function will extract attributes and content from the incoming message and send stock data to the database using a service. If the process succeeds, the callback function will acknowledge the message to Google Cloud Pub/Sub. If an error occurs, the log will be sent with detailed traceback, indicating the failure to acknowledge messages for potential retry.

```
def callback(message:
pubsub_v1.subscriber.message.Message) ->
None:
    try:
        attributes =
dict(message.attributes)
        content =
json.loads(message.data.decode('utf-8'))

StockService(StockPostgresImplementation(
)).add_stock(content, attributes)
        message.ack()
    except Exception as e:
        exception =
traceback.format_exception(type(e), e,
e.__traceback__)
        stack = traceback.format_stack()
        exception[1:1] = stack[:-1]
        traceback_str =
''.join(exception)
        print(traceback_str, flush=True)
        message.nack()
```

Code Program 2. Define message callback function

```
stock_pull_future =
subscriber.subscribe(stock_path,
callback=callback,
flow_control=flow_control)
```

Code Program 3. Start subscription with the pull method

If the connection to the database has been successfully established, Python will initiate a subscription to Google Cloud Pub/Sub with a method similar to that in Code Program 3. The controller will handle potential timeout errors by canceling the subscription and publishing an exception. The controller will continuously attempt to connect to the database by performing query testing and will retry with a five-second delay if the connection fails, printing an error message.

The service plays a crucial role as an intermediary between the controller and the model and interface. In the built system, the service is responsible for encapsulating logic and business functionality, serving as an abstraction layer that hides the complexity of data access and manipulation from the controller and orchestrating interactions with various models and data sources based on controller requests. The service can validate and transform data received from the controller or retrieved from the model before further utilization.

```
def __init__(self, repo: StockInterface):
    self.repo = repo
```

Code Program 4 Setting up the service with the interface

Code Program 4 is a constructor for configuring the service with the Interface. With the method in Code Program 4, the service is initialized by receiving an interface instance and sending it to the attribute. When adding stock data, the Interface acts as a bridge

to the database, allowing the service to access the database.

```
def add_stock(self, data: dict,
attributes: dict):
    conn = self.repo.get_connection()
    try:
        stock = Stock()

    conn = self.repo.get_connection()
    self.repo.add_stock(conn, stock)
    self.repo.commit(conn)
    return {'message': 'Data inserted
successfully.', 'data': data}, 200
except Exception as e:
    self.repo.rollback(conn)
finally:
    self.repo.close_connection(conn)
```

Code Program 5 Add stock

Code Program 5 is a function to add stock. When adding data, the function first establishes a database connection. The function creates an object filled with information from the data dictionary. This object is then passed to the interface to be stored in the database. The service will commit the changes and return a confirmation message with the entered data if successful. However, if an error occurs during any step, the service ensures the database remains consistent by rolling back any changes and closing the connection.

In the software design pattern created, the model has the role of representing and managing application data. The model acts as the core data layer and represents, stores, and manages application data with minimal dependencies on other components. The model interacts with the service layer through the interface. The interface acts as a contract or blueprint that defines methods to be implemented by a class without specifying the actual implementation details. A more flexible, maintainable, and testable architecture is created with the interface.

In the software design pattern formed, the implementation is the final stage that shapes and makes the application functional and adaptive. The implementation is the concrete code that realizes the methods defined in the interface. The implementation provides functionality to interact with the underlying database or system and bridges business logic and the built technology or system. The implementation manages to add stock data to the database. With SQL queries, the function will add a new row to the table and handle if there are data records with the same primary key.

4.3. Deployment Stage

Software deployment is the process of delivering software applications from development to production environments. In the production environment, end users can utilize the created software. Software deployment with the Google Cloud Platform is chosen because it offers various conveniences and flexibility to move and run software from development to production. Google Cloud Platform provides various services and tools that can be tailored to specific needs. Auto-scaling with Google Compute Engine and Google Cloud Pub/Sub is selected because it meets production requirements and offers scalability, reliability, easy integration, and real-time transmission processes.

4.3.1. Managed Instance Group

Managed Instance Group (MIG) has flexible and feature-rich characteristics. With high availability, MIG will automatically repair virtual machines experiencing errors, be preempted (spot VMs), or be deleted by actions initiated by MIG. It features auto-healing, periodically verifying that programs respond as expected in each MIG. When the program requires additional computing resources, autoscaled MIG can automatically increase the number of instances in the group to meet demand. Managed Instance Group can securely deploy new software versions to instances in MIG and support various flexible launch scenarios. Support for stateful workloads allows MIG to perform highly available build deployments and automate software operations with stateful data or configurations, such as databases, DNS servers, and more. MIG also works with load-balancing services to distribute traffic across all instances in the group. The services and features offered by Managed Instance Group make the implementation process of the streamline real-time stock easier and more beneficial.

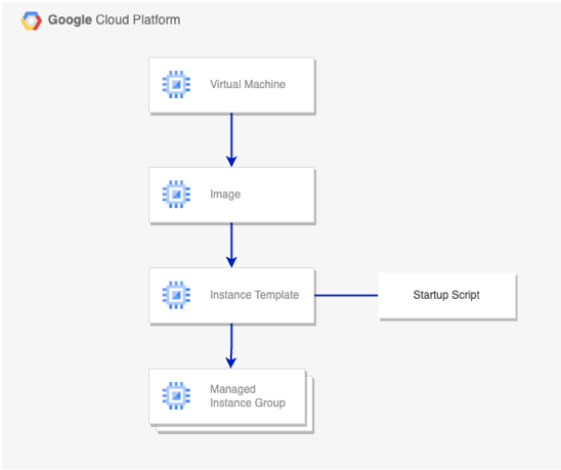

Figure 4. Managed Instance Group build

Figure 5 illustrates the process of forming a Managed Instance Group (MIG). Each virtual machine within the MIG is based on an instance template. An instance template is a container for storing virtual machine (VM) configurations, including machine type, boot disk image, labels, startup scripts, and other instance properties. The formed instance template is constructed using a persistent disk booting image. Before an instance template can be formed, an image or "snapshot" is required, taken from a prepared virtual machine. The

image or "snapshot" is a persistent disk containing the operating system, installed software or programs, and various data necessary to build the system. Virtual machines are prepared to form an image by taking a disk from the virtual machine. Inside the virtual machine's disk, a repository of system implementations, data, and initialized software is stored. All instances are built within one region. The instance template is formed with the disk taken to form the image. A startup script is stored inside the instance template to facilitate continuous development.

```
#/bin/bash
sudo git -C /opt/app/repository pull
origin master
sudo systemctl restart stock.service
```

Code Program 6 The startup script within the instance template.

Code Program 6 is a startup script stored within the instance template. The startup script is a command-line interface for the Linux operating system that will always run when the managed instance group is running based on the instance template. The startup script will be pulled from the system implementation repository. If there are updates to the system implementation script, the clone repository in the boot disk will be updated. However, if there are no updates to the system implementation script, there will be no updates to the clone repository. After pulling from the repository, the service or application will be restarted and continue running.

4.3.2. Pub/Sub

Pub/Sub consists of several service components that make it operational, including:

- Topic: A topic is like a named channel or stream to which messages will be sent.
- Publisher: A publisher is an application that wants to send messages by publishing to a specific topic.
- Subscription: An entity representing interest in receiving messages about a specific topic.
- Subscriber: An application that wants to receive messages makes a subscription to a specific topic and receives messages in the specified subscription.
- Message: Data that flows through the service.

The workflow of Pub/Sub is determined by the method attached to the topic. In Google Cloud Pub/Sub, two methods can occur: push or pull. The method determines how messages are delivered to subscribers. With the push method, messages are actively pushed to the configured endpoint of the subscriber (often a webhook) for immediate processing. The pull method requires subscribers to actively fetch or pull messages from the subscription when they are ready to process them. The publishing and subscribing patterns of Pub/Sub are determined by several factors, including the type of data published to the topic, the number of publishers and subscribers that can communicate with the topic, the availability of topics and messages, and the application's business needs (use-case). The fan-in (many-to-one) publishing subscription pattern is selected because it aligns with the determining factors for selecting the Pub/Sub publishing subscription pattern. The fan-in (many-to-one) pattern allows multiple publisher applications to publish messages on one topic. One topic will be attached to one subscription. This subscription is then connected to one subscriber application that receives all messages published from the topic.

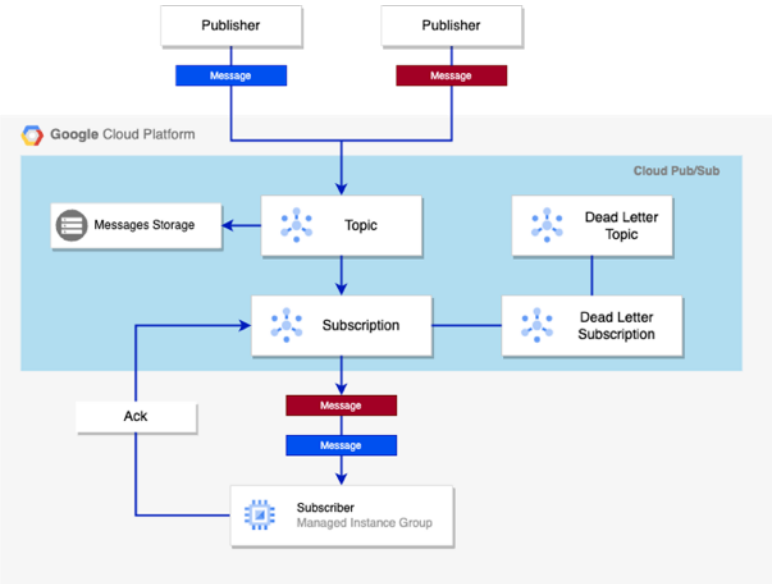

Figure 5. Pub/Sub workflows streamline real-time stock system

Figure 5 depicts the workflow in Pub/Sub with a fan-in (many-to-one) pattern and dead-lettering built for the streamline real-time stock system. The workflow in Pub/Sub follows the message processing

cycle, involving two fundamental entities: multiple publishers and one subscriber, and several Google Cloud Pub/Sub service components. In publishing, multiple publishers independently create messages and send them to the main Pub/Sub topic. Pub/Sub receives and stores each message in the message storage service. In the delivery process, Pub/Sub forwards messages from the topic through subscriptions to the subscriber. In the system built within Google Cloud Platform, the subscriber is a Managed Instance Group (MIG). The pull scheme is the scheme or method involved in the message delivery process to the subscriber. The Managed Instance Group (MIG) will actively fetch or pull messages from the subscription when the subscriber is ready to process them. When the message is delivered, the subscriber, in this case, the Managed Instance Group (MIG), will attempt to process the message.

If the message is successfully processed, the subscriber sends an acknowledgment (ack) back to Pub/Sub. Pub/Sub will then delete the successfully processed message from the topic. If the subscriber fails to process the message (due to errors, timeouts, etc.), the subscriber sends a negative acknowledgment (nack) to Pub/Sub. Pub/Sub will attempt to resend the message based on the retry settings. If the message exceeds the retry limit, Pub/Sub forwards the failed message to the dead-letter topic. The dead-letter topic is a repository for messages that the primary subscriber cannot successfully process. The dead-letter topic has a subscription capable of managing potential monitoring, analysis, or potential reprocessing strategies for failed messages.

4.3.3. Automatic Scaling

There are several critical components required to run auto-scaling for the streamline real-time stock application, including:

- Managed Instance Group (MIG): A collection of identical virtual machines (VMs) created from an instance template. Auto-scaling works by adding or removing VMs from the MIG.
- Auto-scaling policies: Rules that determine when and how scaling is performed. These rules are based on:
  - Metric: Pub/Sub *queue*.
  - Scaling thresholds:
    - Scale out: If the number of messages per virtual machine is greater than or equal to (>=) five (5).
    - Scale in: If the number of messages per virtual machine decreases below (<) five (5).
  - Scaling decisions:
    - Scale out: Adding one (1) virtual machine.
    - Scale in: Removing one (1) virtual machine.
    - Load balancing: A configurable period after scaling actions to prevent rapid fluctuations and give the application time to stabilize.

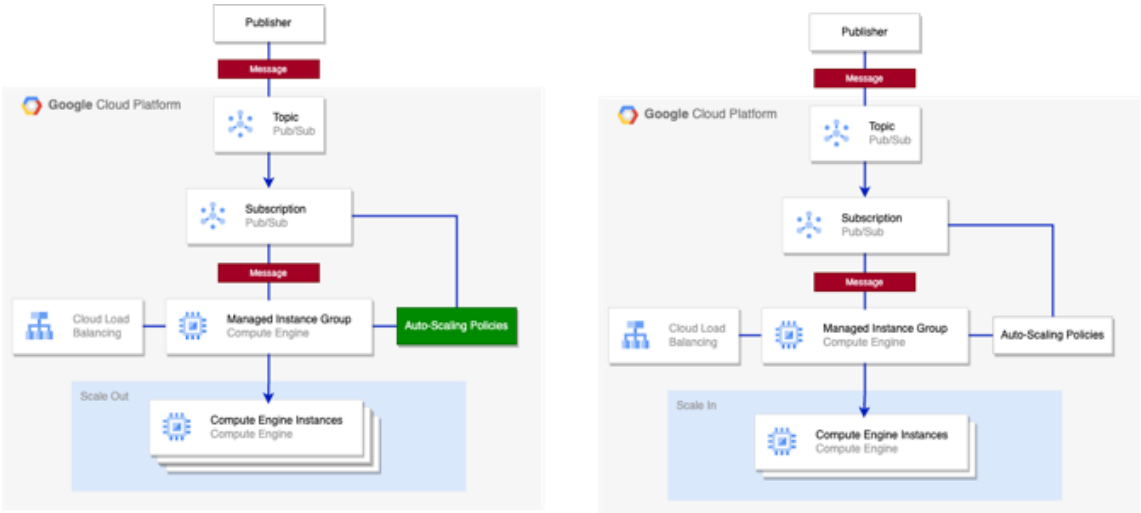

Figure 6. Auto-scaling workflow

Figure 6 illustrates the workflow of scaling out and scaling in in the auto-scaling system. In the task generation phase, the application acting as a publisher sends data generates tasks and publishes them as messages to the Pub/Sub topic. These messages are stored in the Pub/Sub subscription and are awaiting processing. Virtual machines (VMs) within the Managed Instance Group (MIG) process the messages they receive, handling the tasks represented by the messages. The load balancer manages Incoming task workload distribution across all available VMs in the MIG for processing. Auto-scaling continuously monitors the average number of unacknowledged messages per VM in the MIG, representing the current workload on each instance. If the average number of unacknowledged messages per

VM exceeds the previously specified target (i.e., five messages), the VMs are overloaded. Auto-scaling initiates a scale-out operation, adding more VMs to the MIG to increase processing capacity. If the average number of messages per VM consistently drops below the target (i.e., five messages), it indicates underutilized VMs. Auto-scaling may trigger a scale-in operation, removing VMs from the MIG to optimize costs

4.4. System Testing

System testing is conducted using three methods and utilizes tools to demonstrate the performance and functionality of the system. The methods used are performance testing, publish testing and auto-scaling testing.

4.4.1. Performance Testing

Performance testing systematically evaluates and validates the stability, speed, and scalability of software applications, systems, or components under various predefined load, pressure, and resilience conditions. The type of performance testing used is load testing, aimed at determining the target load of a system. Testing is conducted using the Apache JMeter tool. Based on JMeter testing, a good system or application category can handle user load well without significantly affecting response time and throughput.

The testing involves publishing a data value with an HTTP (Hypertext Transfer Protocol) request to an external IP (Internet Protocol) designated for publishing. Factors analyzed in load testing on the streamline real-time stock system include response speed, error rate, response time, and server configuration.

Table 1 Aggregate report result

Label	# Samples	Average	Median	Error %	Throughput	Received KB/sec	Sent KB/sec
HTTP Request	1000	198	171	0%	499.50050	159.02	163.41
TOTAL	1000	198	171	0%	499.50050	159.02	163.41

Table 1 presents the aggregate report from the load-testing conducted with JMeter. An aggregate report is a collection of data from various sources that provides a general overview and high-level analysis of a specific topic. This report does not focus on individual details but summarizes trends, patterns, and key insights. The results of the aggregate report indicate that the system's performance is satisfactory. From 1000 users (threads), the aggregate report shows that all user requests were successful, with an average response time of 198 milliseconds and no errors. The throughput value indicates a fairly high number of 499.5 requests per second, indicating that the server can handle many requests per second.

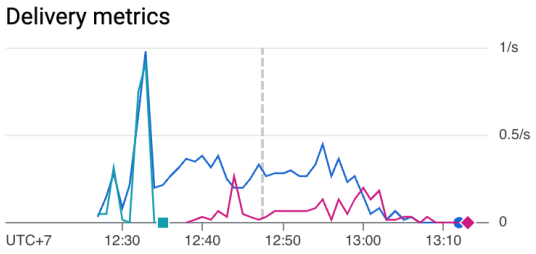

Figure 7. Delivery Metrics Pub/Sub graph

Figure 7 depicts a graph of delivery metrics observed in Pub/Sub. The graph is obtained from Cloud Monitoring, an integrated suite for monitoring, logging, and tracking applications and systems running on the Google Cloud Platform (GCP) and other services. Delivery metrics refer to measurements that track the effectiveness of message delivery between publishers and subscribers. The testing was conducted for approximately one (1) hour by publishing 100 messages to the Pub/Sub topic. Several detailed metrics describe the main deliveries in Pub/Sub.

- The green line represents the publishing rate, indicating the number of messages published to the topic per unit of time. Publishing occurs during the initial ± ten minutes starting from when the publisher publishes the topic. Publishing happens at a rate ranging from 0.917 or 91% messages per second, with a maximum speed of one (1) or 100% messages per second.
- The blue line represents the pulling rate, indicating the number of messages the subscriber pulls from the topic per unit of time. Subscribers begin pulling messages once they enter the topic and continue until all messages have been received. The subscriber can process 100 messages within a ± 40-minute period at an average rate of 0.45 or 45% messages per second from the maximum speed of one (1) or 100% messages per second.
- The red line represents the acknowledgment rate, representing the number of messages successfully acknowledged by the subscriber per unit of time. Acknowledgment informs Pub/Sub that the subscriber has received and processed the message. The acknowledgment action by the subscriber is performed ± 12 minutes after the message is successfully pulled until all messages are acknowledged. The acknowledgment process of 100 messages occurs within a ± 30-minute period at an average rate of 0.15 or 15% messages per second from the maximum speed of one (1) or 100% messages per second.

4.4.2. Publish Testing

The testing was conducted by publishing messages to the Pub/Sub topic so that the Pub/Sub subscription could process the messages and forward them to the subscriber. The subscriber, a virtual machine inside the Managed Instance Group (MIG), would then process the messages and insert them into the database according to the data parameters and commands received (update or insert). The testing was performed using the Postman tool. Postman is a software application that tests APIs (Application programming interfaces). Postman sends API requests to the Managed Instance Group (MIG) server via the Pub/Sub intermediary and receives responses.

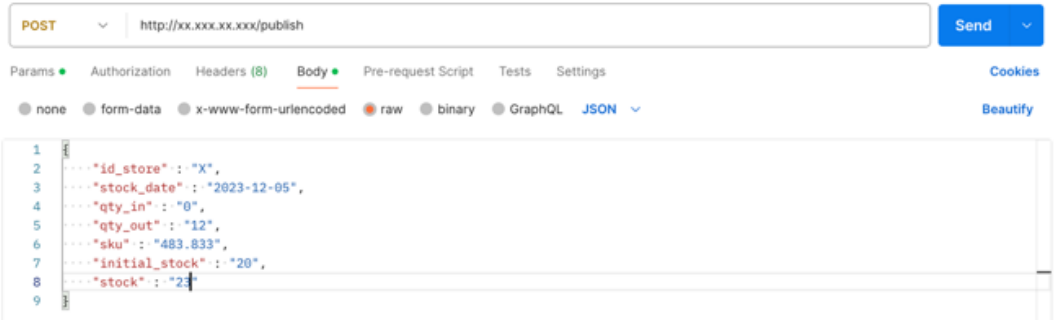

Figure 8. API with Postman

Figure 9. Server response

Figure 8 illustrates the process of sending message data using an API in Postman. The HTTP endpoint for publishing is included in Postman, along with the route using the POST method for sending messages via the API. The message from the publisher is placed in the message body. If the process is successful, the server will send a response containing the message ID and success status, as shown in Figure 9.

4716	X	2023-12-05	0	14	180,506	318	304
4717	X	2023-12-05	8	1	38,183	8	15
4718	X	2023-12-05	0	1	639,818	42	41
4719	X	2023-12-05	0	10	393,851	49	57
4720	X	2023-12-05	0	1	623,969	5	4
4721	X	2023-12-05	0	12	483.833	20	23

Figure 10. Database response

Figure 10 shows the response from the database, indicating the message (data) that has been successfully inserted into the database. If the message (data) being inserted is new or the value of the message is not already in the database, an insert process will occur. However, if the value of the message already exists in the database, an update process will occur.

4.4.3. Auto-scaling Testing

Automatic scaling testing was conducted by performing load testing on the Managed Instance Group (MIG) for approximately one (1) hour. The load testing aimed to determine the maximum and minimum performance of the automatic scaling system on the Managed Instance Group. The testing involved publishing 100 messages to the Pub/Sub topic. The Managed Instance Group, acting as the subscriber, received and processed these messages according to its configuration. Automatic scaling would scale up based on the maximum number of messages it can handle and scale down when the messages have been successfully processed (acknowledged).

Figure 11. Group Size graph

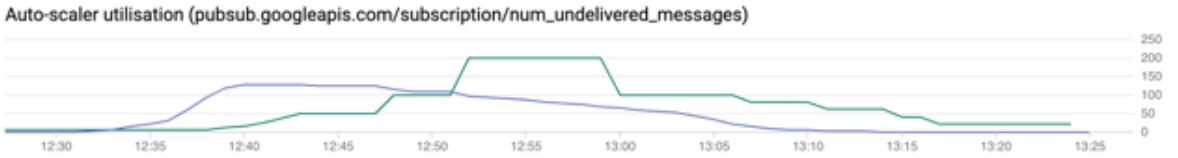

Figure 12. Auto-scaler utilisation graph

Figure 11 represents a graph obtained from Cloud Monitoring when the Managed Instance Group scales up and down. Figure 12 represents the auto-scaler utilization graph obtained from Cloud Monitoring. The graph shows the auto-scaler utilization values with signals for auto-scaling, which are Pub/Sub messages. The testing was conducted with three batch configurations for the Managed Instance Group,

- In the first batch, the Managed Instance Group was configured with a maximum message count of five (5), a maximum instance count of ten (10), and a minimum instance count of one (1). Gradual scaling up to the maximum instances (ten instances) occurred in this batch, and the auto-scaler utilization reached its peak value of 126 with a serving capacity of ten (10) instances within a ± eight (8)-minute timeframe.
- In the second batch, the Managed Instance Group was configured with a maximum message count of ten (10), a maximum instance count of 20, and a minimum instance count of 1. Although the maximum message count was increased to ten (10), the auto-scaler maintained ten instances, and there was an increase to the maximum instances (20 instances) within a ± ten minute timeframe. In this batch, the auto-scaler raised the serving capacity to 200, resulting in a decrease in auto-scaler utilization to 67.52 within a ± ten (10)-minute timeframe.
- In the third batch, the Managed Instance Group was configured with a maximum message count of 20, a maximum instance count of ten, and a minimum instance count of one (1). In the third batch, the instances gradually decreased to the maximum instance count (ten instances) within a ± eight (8)-minute timeframe. The graph indicates that the performance of auto-scaler utilization also decreased to 20.03 with a serving capacity of 100 within a ± eight (8)-minute timeframe.
- After all messages were successfully processed (acknowledged) gradually, the instances scaled down to the minimum instance count (one instance) within a ± 17-minute timeframe. Auto-scaler utilization also decreased to zero (0) with a serving capacity of 20 within a ± 17-minute timeframe.

The testing indicates that the automatic scaling system in the Managed Instance Group functions effectively and can handle changes in workload effectively. The automatic scaling system has demonstrated scalability and flexibility in handling high workloads with efficient resource utilization. The time taken to increase the instance count ranges between 8-10 minutes, while the time taken to decrease the instance count ranges between 8-17 minutes. In managing high workloads, auto-scaling utilizes available resources to the maximum to expedite the acknowledgment process. However, auto-scaling remains flexible and performs optimally even with minimal resources.

5. DISCUSSION

Implementing real-time stock streamlining using auto-scaling via Google Cloud Pub/Sub has been successful and has shown high performance. The utilization of auto-scaling through Google Cloud Pub/Sub is capable of handling a workload of 1000 users with an average response time of 198 milliseconds and a throughput of 499.5 requests per second. In previous research, using auto-scaling through CPU utilization to handle a simple web server with the same workload yielded an average response time of 90 milliseconds and a throughput of 1.7 requests per second [9]. This indicates that in handling high workloads, auto-scaling can maximize the utilization of available resources and execute various forms of automatic scaling signals efficiently.

The utilization of Google Cloud Pub/Sub as a scalable asynchronous messaging service, along with its dead-letter feature, provides high satisfaction in scalability [1]. With its features and reliability, Google Cloud Pub/Sub ensures real-time message delivery and that failed messages are not lost, allowing them to be resent. This benefits the business processes at PT XYZ. Real-time data aids in faster and more accurate decision-making related to inventory. The auto-scaling system through Google Cloud Pub/Sub for real-time inventory streamlining implementation ensures that services are always available with automatic scalability and high reliability while providing secure data.

6. CONCLUSION

Based on the results and discussion from the conducted research, several conclusions can be drawn as follows: 1) The streamline real-time stock system is capable of handling high workloads, as evidenced by load testing with 1000 users (threads). The system can handle user requests with an average response time of 198 milliseconds per thread, and no errors occur. 2) The use of Google Cloud Pub/Sub as an

asynchronous and scalable messaging service, along with its good integration with the streamline real-time- stock system built within the Managed Instance Group, successfully handles message publishing with a large volume. This results in metrics such as a publish rate of 0.917 or 91% messages per second, a pull rate with an average value of 0.45 or 45% messages per second, and an ack rate with an average value of 0.15 or 15% messages per second. 3) As demonstrated during user testing, the system's functionality has been running smoothly. It successfully publishes messages to the Pub/Sub topic and inserts them into the database. 4) The automatic scaling system is capable of handling high workloads and changes in workload with scalability and flexibility. With the publishing of 100 messages, the auto-scaler optimizes performance and increases service availability. The time taken to increase the instance count (scale-up) ranges between 8-10 minutes, while the time taken to decrease the instance count (scale-down) ranges between 8-17 minutes. The implementation of the streamline real-time stock system with auto-scaling via Google Cloud Pub/Sub can provide optimal performance and scalability with high availability, good security, and cost savings.

Developing and adding features to the streamline real-time stock system with auto-scaling through Google Cloud Pub/Sub would be highly beneficial. Implementing system monitoring and logging capabilities would track system performance and health and record events and important information for debugging and troubleshooting purposes. Other features would also be valuable, such as developing a user interface (UI) to visualize real-time stock data and gain insights into inventory levels, trends, and potential issues.

REFERENCES

[1] G. Schumy and Y. A. Susetyo, “Rancang Bangun Sistem Sinkronisasi Data Menggunakan Google Cloud Pub/Sub Dan Flask Di PT XYZ,” in Jurnal MNEMONIC, 2022, vol 5, no. 2, pp 85-92, 2022, doi: 10.36040/mnemonic.v5i2.4645.

[2] S. L. Gutiérrez and Y. P. Vera, “A Cloud Pub/Sub Architecture to Integrate Google Big Query with Elasticsearch using Cloud Functions,” International Journal of Computing, vol. 21, no. 3, pp. 369–376, 2022, doi: 10.47839/ijc.21.3.2694.

[3] S. K. Rout, J. V. R. Ravindra, A. Meda, S. N. Mohanty, and V. Kavididevi, “A Dynamic Scalable Auto-Scaling Model as a Load Balancer in the Cloud Computing Environment,” EAI Endorsed Transactions on Scalable Information Systems, vol. 10, no. 5, pp. 1–7, 2023, doi: 10.4108/eetsis.3356.

[4] Google Cloud, "Instance Group", 2023. https://cloud.google.com/compute/docs/instance-groups?hl=id (accessed Nov. 1, 2023).

[5] S. P. T. Krishnan, Jose L. Ugia Gonzalez, *Building Your Next Big Thing with Google Cloud Platform*. Apress Berkeley, CA.

[6] Google Cloud, "Overview of the Pub/Sub service", 2023. https://cloud.google.com/pubsub/docs/pubsub-basics (accessed Nov. 1, 2023).

[7] Google Cloud, "Event-driven architecture with Pub/Sub", 2023. https://cloud.google.com/solutions/event-driven-architecture-pubsub (accessed Nov. 1, 2023).

[8] Google Cloud, "Autoscaling groups of instances", 2023. https://cloud.google.com/compute/docs/autoscaler (accessed Nov. 5, 2023).

[9] D. Gustian, Y. Fitrisia, S. Purwantoro ESGS, W. Novayani, P. Caltex Riau, and J. Umbansari No, “Implementasi Automation Deployment pada Google Cloud Compute VM menggunakan Terraform,” Jurnal Inovtek Polbeng, vol. 8, no. 2, pp. 50–62, 2023.

[10] A. A. Wibowo Putri and Y. A. Susetyo, “Implementation Of Flask For Stock Checking In Distribution Center & Store On Monitoring Stock Application In PT. XYZ,” Jurnal Teknik Informatika (Jutif), vol. 3, no. 5, pp. 1265–1274, Oct. 2022, doi: 10.20884/1.jutif.2022.3.5.334.

[11] B. P. Putra and Y. A. Susetyo, “IMPLEMENTASI API MASTER STORE MENGGUNAKAN FLASK, REST DAN ORM DI PT XYZ,” Jurnal Sistem Informasi (SISTEMASI), vol. 9, no. 3, pp. 543-556, 2020, doi: 10.32520/stmsi.v9i3.89.

[12] R. Bayu, A. Pradana, and A. Bhawiyuga, “Pengembangan Platform IoT Cloud berbasis Layanan Komputasi Serverless Google Cloud Platform (GCP),” Jurnal Pengembangan Teknologi Informasi dan Ilmu Komputer, vol. 6, no. 4, April. 2022.

[13] A. A. Wibowo Putri and Y. A. Susetyo, “Implementation Of Flask For Stock Checking In Distribution Center & Store On Monitoring Stock Application In PT. XYZ,” Jurnal Teknik Informatika (Jutif), vol. 3, no. 5, pp. 1265–1274, Oct. 2022, doi: 10.20884/1.jutif.2022.3.5.334.

[14] M. C. Silva Filho, C. C. Monteiro, P. R. M. Inácio, and M. M. Freire, “A Distributed Virtual-Machine Placement and Migration Approach Based on Modern Portfolio Theory,” Journal of Network and Systems Management, vol. 32, no. 1, Mar. 2024, doi: 10.1007/s10922-023-09775-8.

[15] S. L. Gutiérrez and Y. P. Vera, “A Cloud

Pub/Sub Architecture to Integrate Google Big Query with Elasticsearch using Cloud Functions," International Journal of Computing, vol. 21, no. 3, pp. 369–376, 2022, doi: 10.47839/ijc.21.3.2694.

[16] C. Mustafa Mohammed and S. R. M Zeebaree, "Sufficient Comparison Among Cloud Computing Services: IaaS, PaaS, and SaaS: A Review," International Journal of Science and Business, vol. 5, no. 2, pp. 17–30, 2021, doi: 10.5281/zenodo.4450129.

[17] A. J. Budianto, P. Ocsa, and N. Saian, "Pengembangan Modul Inventory Management pada Aplikasi Master Distribution Centre System Menggunakan Framework Flask di PT XYZ," Jurnal Teknologi Informasi dan Komunikasi), vol. 7, no. 2, pp. 201–207, 2023, doi: 10.35870/jti.

[18] J. Nam, Y. Jun, and M. Choi, "High Performance IoT Cloud Computing Framework Using Pub/Sub Techniques," Applied Sciences (Switzerland), vol. 12, no. 21, Nov. 2022, doi: 10.3390/app122111009.

[19] P. M. Tobing, M. Ariance, and I. Pakereng, "Migrasi Aplikasi Stock Opname Platform Desktop Ke Android Menggunakan Kivy Framework (Studi Kasus Di PT Sumber Alfaria Trijaya Tbk)," Indonesian Journal on Computer and Information Technology, 2020, vol. 6, no. 2, pp. 151-159. Nov. 2021.

[20] F. V. L. Dewangga and P. O. Nugraha Saian, "Automatic Git Repository Deployer In Ubuntu Using Python, Jenkins And Cloud Firestore At PT XYZ," Jurnal Teknik Informatika (Jutif), vol. 4, no. 6, pp. 1313–1325, Dec. 2023, doi: 10.52436/1.jutif.2023.4.6.1062.

[21] A. R. Nasution, F. Dewanta, and B. Aditya, "AUTO SCALING DATABASE SERVICE WITH MICRO KUBERNETES CLUSTER," Jurnal Teknik Informatika (Jutif), vol. 3, no. 4, pp. 923–927, Aug. 2022, doi: 10.20884/1.jutif.2022.3.4.484.