International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 7
DOI: https://doi.org/10.17762/ijritcc.v11i7.7846
***Article Received**: 29 April 2023 **Revised**: 22 June 2023 **Accepted**: 03 July 2023*

A Novel Approach for Triggering the Serverless Function in Serverless Environment

Shashank Srivastava[1] , Bineet Kumar Gupta[2], Dheeraj Tandon[3], Kartikesh Tiwari[4], Anshita Raj[4], Megha Agarwal[4]
[1]Author, DCSIS, IOT, Shri Ramswaroop Memorial University,Lucknow, Uttar Pradesh, India
[1]shivam.shashank@gmail.com
[2,4] DCSIS, IOT, Shri Ramswaroop Memorial University,Lucknow, Uttar Pradesh, India
[2]hod.dcsis@srmu.ac.in, [4]kartikesh.csis@srmu.ac.in, [4]anshitaraj.csis@srmu.ac.in, [4]meghaaggarwal.csis@srmu.ac.in
[3]Department of Computer Science and Engineering, SRM Institute of Science & Technology, Delhi - NCR Campus, Modinagar –Ghaziabad, Uttar Pradesh, India
[3]dheerajtandon9@gmail.com

Abstract— Serverless computing has gained significant popularity in recent years due to its scalability, cost efficiency, and simplified development process. In a serverless environment, functions are the basic units of computation that are executed on-demand, without the need for provisioning and managing servers. However, efficiently triggering serverless functions remains a challenge, as traditional methodologies often suffer from latency, Time limit and scalability issues and the efficient execution and management of serverless functions heavily rely on effective triggering mechanisms. This research paper explores various design considerations and proposes a novel approach for designing efficient triggering mechanisms in serverless environments. By leveraging our proposed methodology, developers can efficiently trigger serverless functions in a variety of scenarios, including event-driven architectures, data processing pipelines, and web application backend.

Keywords- Serverless Computing, AWS, AWS Lambda, Cloud Computing, Serverless Functions.

I. INTRODUCTION

Serverless computing is a cloud computing model where the cloud provider (such as AWS Lambda, Azure Functions, or Google Cloud Functions) manages the infrastructure and automatically provisions and scales the resources needed to run applications. In serverless computing, developers focus solely on writing and deploying code without having to manage servers or infrastructure.

A. *Key Characteristics of Serverless Computing:*

Event-Driven Architecture: Serverless applications are designed around events or triggers. Functions are invoked in response to events such as HTTP requests, database changes; file uploads, or scheduled events. This event-driven architecture allows for a highly decoupled and scalable system.
Automatic Scaling: Serverless platforms automatically handle the scaling of resources based on the incoming workload. Functions are automatically scaled up or down based on the demand, ensuring optimal resource utilization and high availability. This scalability is achieved without the need for manual configuration or capacity planning.
Pay-Per-Use Pricing: Serverless computing follows a pay-per-use pricing model, where users are billed only for the actual usage of resources. Billing is typically based on the number of function invocations and the duration of each invocation. This pricing model offers cost efficiency as users pay only for the actual execution time of their functions, rather than paying for idle resources.
Stateless Execution: Serverless functions are stateless, meaning they don't maintain any persistent state between invocations. This statelessness allows functions to be easily scaled horizontally and ensures that each invocation is independent and isolated.
Event-Driven Scaling: Serverless platforms scale resources based on the incoming workload. As the number of events increases, the platform automatically provisions more compute resources to handle the workload. This event-driven scaling ensures that resources are efficiently utilized and can handle varying workloads without manual intervention.
Developer Productivity: Serverless computing abstracts away the infrastructure management, allowing developers to focus solely on writing application logic. It reduces the operational burden by eliminating the need to provision, manage, and scale servers, enabling faster development cycles and improved developer productivity.
Fault Tolerance and High Availability: Serverless platforms handle fault tolerance and high availability transparently. They automatically replicate functions across multiple availability zones, ensuring resilience and fault tolerance. In case of a failure or resource unavailability, the platform seamlessly redirects the events to available resources.

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 7
DOI: https://doi.org/10.17762/ijritcc.v11i7.7846
***Article Received**: 29 April 2023 **Revised**: 22 June 2023 **Accepted**: 03 July 2023*

II. SERVERLESS COMPUTING OVERVIEW:

Serverless computing offers several benefits and advantages, but it also comes with its own set of challenges. The benefits and challenges of serverless computing are as follows:

A. Benefits of Serverless Computing:

Reduced Operational Overhead: With serverless computing, developers are relieved of the burden of managing infrastructure, including servers, operating systems, and networking. The cloud provider takes care of these operational aspects, allowing developers to focus solely on writing code and delivering business value.

Scalability and Elasticity: Serverless platforms automatically scale resources based on the incoming workload. They handle the provisioning and scaling of resources, ensuring that applications can handle sudden spikes in traffic and scale down during periods of low activity. This enables applications to achieve high scalability and responsiveness without manual intervention.

Cost Efficiency: Serverless computing follows a pay-per-use pricing model. Users are billed only for the actual execution time of their functions, rather than paying for idle resources. This results in cost savings as resources are efficiently utilized, and users pay only for the actual usage of compute resources.

Increased Developer Productivity: By abstracting away infrastructure management, serverless computing allows developers to focus on writing business logic and delivering features. It reduces the time and effort required for provisioning, managing, and scaling servers, enabling faster development cycles and improved developer productivity.

Event-Driven Architecture: Serverless applications are built around an event-driven architecture. They can easily integrate with various event sources, such as HTTP requests, database triggers, or message queues. This event-driven nature enables the development of loosely coupled, modular applications that can react to events in real-time.

B. Challenges of Serverless Computing Environment:

Cold Start Latency: Serverless functions may experience a cold start when invoked for the first time or after a period of inactivity. This can result in increased latency as the cloud provider provisions resources to handle the request. Cold starts can impact real-time and low-latency applications, requiring careful optimization and management of functions to minimize their occurrence **[11]**.

Vendor Lock-in: Serverless platforms have varying degrees of compatibility and interoperability between different cloud providers. Moving serverless functions from one provider to another may require significant modifications and rearchitecting. This can lead to vendor lock-in, limiting flexibility and making it challenging to switch providers.

Function Execution Limits: Serverless platforms impose limits on function **[3][6]** execution time, memory usage, and maximum request payload size. These limits can impact certain workloads that require longer execution times or deal with large data sets. Developers need to carefully design their functions and consider these limitations to ensure compatibility with serverless platforms.

Debugging and Testing: Debugging and testing serverless functions can be challenging due to the distributed and event-driven nature of serverless architectures. Traditional debugging techniques may not work effectively, requiring the use of specialized debugging tools and techniques for identifying and fixing issues in serverless applications.

State Management: Serverless functions are stateless by nature, meaning they don't maintain persistent state between invocations. Managing and persisting state across function invocations can introduce complexity, especially for applications that require context or have long-running workflows. Additional measures, such as external storage or caching, need to be implemented to handle stateful scenarios.

III. ARCHITECTURAL COMPONENTS OF SERVERLESS COMPUTING

As shown in Figure 1 Serverless computing relies on a set of architectural components to enable its functionality and provide a seamless experience for developers. These components work together to handle function invocations, manage resources, and ensure scalability and responsiveness. The key architectural components of serverless computing:

Function-as-a-Service (FaaS):

FaaS**[18][30]** is the core component of serverless computing. It allows developers to write and deploy functions that are executed in response to events or triggers.Functions are self-contained units of code that perform specific tasks or operations. They are stateless and designed to be event-driven.FaaS platforms, such as AWS Lambda, Azure Functions, or Google Cloud Functions, handle the deployment, scaling, and execution of functions.

Event Sources:

Event sources **[7][31][32]** trigger the execution of serverless functions. They generate events that serve as the input for functions to process.Common event sources include HTTP requests, database changes, file uploads, message queues, scheduled events, and IoT device messages.Event sources can be external systems or services that emit events, or they can be internal triggers defined within the serverless platform.

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 7
DOI: https://doi.org/10.17762/ijritcc.v11i7.7846
***Article Received**: 29 April 2023 **Revised**: 22 June 2023 **Accepted**: 03 July 2023*

Function Triggers:

Function triggers **[10]** are responsible for connecting the event sources to the serverless functions.Triggers define the conditions or rules that determine when a function should be invoked in response to an event.Triggers can be configured to handle specific events, filter events based on criteria, or route events to different functions or workflows.

Compute Infrastructure:

Serverless platforms manage the underlying compute infrastructure required to execute functions. The platforms automatically provision and allocate resources based on the incoming workload to ensure scalability and availability. The compute infrastructure can dynamically scale up or down to handle varying levels of demand without manual intervention.

Resource Orchestration:

Resource orchestration components manage the allocation and coordination of resources needed to execute functions. They handle the provisioning and lifecycle management of compute resources, network resources, and other dependencies required by functions. Orchestration components ensure that the necessary resources are available to handle function invocations and coordinate their execution.

Monitoring and Logging:

Monitoring and logging components provide visibility into the execution and performance of serverless functions.They collect and analyze metrics, logs, and traces to help developers monitor the behavior, identify bottlenecks, and optimize the performance of their functions.Monitoring and logging tools enable real-time monitoring, debugging, and troubleshooting of serverless applications.

Authentication and Authorization:

Serverless platforms incorporate authentication and authorization mechanisms to secure function invocations and access to resources.They provide authentication methods, such as API keys, OAuth, or IAM roles, to ensure that only authorized users or systems can trigger functions and access protected resources.Authorization mechanisms control the permissions and access levels granted to different entities interacting with the serverless application.

These architectural components[33][34] work together to provide the foundation for serverless computing. They enable the seamless execution of functions, handle event-driven triggers, manage resources, ensure scalability, and provide monitoring and security capabilities. By leveraging these components, developers can focus on writing business logic without the need to manage infrastructure or worry about scalability and availability.

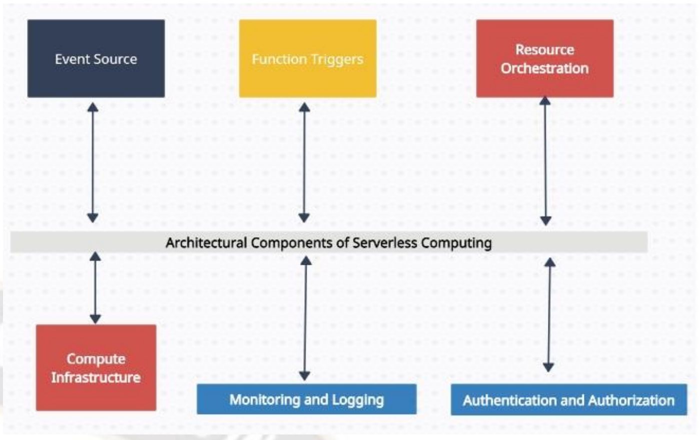

Fig. 1: Architectural Components of Serverless Computing

IV. RELATED APPROACHES

FaaS is based on the event-driven programming approach, which is inspired by the well-known Active Database Systems **[8].** Numerous event-driven abstractions, including as triggers, Event Condition Action (ECA), and composite event detection, have been implemented into the FaaS framework." In the past, event-based triggering was widely used to provide responsive coordination of remote systems **[9,10].** Furthermore, event-based mechanisms and triggers have been widely used in the development of workflow and orchestration systems **[11-14].** The ECA paradigm, which includes triggers and rules, is well-suited for describing transitions in finite state machines that reflect workflows. In research paper **[15],** for instance, they propose employing synchronous aggregation triggers to coordinate massively parallel data processing operations.

The research paper **[14]** highlights a captivating related work that shows how composite subscriptions in content-based publish/subscribe systems are used to provide decentralized Event-based Workflow Management. Through content-based subscriptions in a Composite Subscription Language, their PADRES technologies enable parallelization, alternation, sequence, and repetition compositions.

The intersections of the Complex Event Processing (CEP) and Business Process Management (BPM) communities have recently been examined in a pertinent study **[16].** This survey summarizes recent efforts in this field as well as current problems with merging both models. Our article specifically addresses their issue of "Executing business processes via CEP rules," and our main contribution is our serverless reactive and flexible architecture.

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 7
DOI: https://doi.org/10.17762/ijritcc.v11i7.7846
***Article Received**: 29 April 2023 **Revised**: 22 June 2023 **Accepted**: 03 July 2023*

The "Serverless trilemma" proposed by IBM **[17]** is a pertinent related work applicable to serverless settings that aims to provide reactive orchestration of serverless functions. The authors provide a method for sequential compositions on top of Apache OpenWhisk and support reactive run-time support for function orchestration.

The CNCF community has recently concentrated its efforts on creating a standardized specification for Serverless Workflows **[18].** In their approach, workflows are defined explicitly using a YAML file that includes state transitions for data management and control flow logic, descriptions for CloudEvents for consumption, and instructions for event-driven execution of serverless services. In order to ensure portability and prevent vendor lock-in, an abstract specification that may be understood by several systems is intended.

Various serverless computing orchestration systems have been proposed in numerous studies, including **[19–24].** Many of them, however, rely on centralized server-based resources that cannot scale down to zero, including machine virtualization or dedicated resources. The orchestrator component consequently remains active throughout the whole workflow execution, leading to wasteful resource use for workflows that take a long time to finish because the orchestrator frequently sits idle while waiting for lengthy tasks to complete. Additionally, the designs and fault tolerance of some of these systems are complicated by the utilization of functions calling functions patterns. None of these solutions currently available offer flexible trigger abstractions to generate different types of orchestrators.

Workflow orchestration is compared across triggers and Durable Functions by using workflow as code in **[25].** They contend that although using triggers for workflow orchestration is technically feasible, it is not ideal due to drawbacks such as the requirement to create different queues or directories for every step, triggers' inability to wait for the completion of numerous prior steps, and their unsuitability for proper error handling. In contrast, we will demonstrate in this post that by using a Rich Trigger framework, we may get around these difficulties. Extended trigger logic allows us to provide rules for event screening to prevent the creation of additional queues and event aggregation to carry out a multiple join. We can also ensure fault tolerance through the use of event replay and checkpointing.

Today, major cloud providers such as IBM Composer, Amazon Step Functions, Azure Durable Functions, and Google Cloud Workflows provide cloud orchestration and function composition capabilities. These services are suitable for various types of workloads because each one of them has particular features and limitations.

In two earlier articles **[4,5],** public FaaS orchestration services for managing massively parallel workloads were evaluated. When executing map jobs, it was discovered that IBM Composer provided the best speed and had the lowest overheads, but competing services like ASF or ADF had high overheads. Furthermore, we will show how ASFE performs well for concurrent workloads in this research.

Despite the wide range of cloud orchestration services already in existence, none of them provide an open and extensible trigger-based API that enables the creation of unique workflow engines.

Although there are many cloud orchestration services accessible none of them provide an extensible trigger-based API that enables the development of unique workflow engines. In this study, we demonstrate how Trigger flow may be used to implement models that are already in existence, such as ASF or Airflow DAGs. Trigger flow, which makes use of Kubernetes standard features, is not merely another scheduler but a reactive meta-tool to build reactive orchestrators.

The importance of cloud event routing and Knative Eventing as a unifying infrastructure for diverse cloud services and applications has grown in the context of event-based architectures. The CNCF Cloud Events standard is essential in this area and is utilized by major cloud providers such as Amazon, Azure, Google, and IBM to provide event routing services.

For developers utilizing Kubernetes, the Knative project was developed to give a serverless-like experience. It offers high-level abstractions for scalable functions (Knative Serving) and event processing (Knative Eventing). Utilizing sophisticated abstractions not found in Knative Eventing, such as dynamic triggers, trigger interception, custom filters, termination events, and a shared context, Trigger flow seeks to use Knative Eventing to enable extensible trigger-based orchestration of serverless workflows. Future event routing systems might use these innovative services to streamline task formulation, streaming, and orchestration.

V. TRIGGERING MECHANISMS IN SERVERLESS ENVIRONMENTS:

In serverless environments, triggering mechanisms play a crucial role in invoking functions based on events or triggers **[1][2].** These mechanisms determine when and how functions are executed in response to specific events or conditions.

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 7
DOI: https://doi.org/10.17762/ijritcc.v11i7.7846
***Article Received**: 29 April 2023 **Revised**: 22 June 2023 **Accepted**: 03 July 2023*

Some common triggering mechanisms used in serverless environments:

Event-Driven Invocations:

Event-driven**[7][8][35]** invocations are the most fundamental triggering mechanism in serverless computing. Functions are triggered when specific events occur, such as an HTTP request, changes in a database, file uploads, or messages in a message queue.The event sources generate events, which are then forwarded to the serverless platform, which in turn invokes the relevant functions.Event-driven invocations enable real-time processing and allow functions to react to changes in the system or external events.

HTTP Triggers:

HTTP triggers enable functions to be invoked via HTTP requests.Functions can be exposed as HTTP endpoints, allowing external systems or users to invoke the functions by sending HTTP requests.HTTP triggers are commonly used for building RESTful APIs or handling webhooks.

Timer-Based Triggers:

Timer-based triggers schedule functions to execute at predefined intervals or specific times.Functions can be configured to run periodically, such as every minute, hourly, daily, or on specific dates.Timer-based triggers are useful for tasks that require periodic data processing, batch jobs, or scheduled maintenance tasks.

Database Triggers:

Database triggers invoke functions when changes occur in a connected database.Functions can be triggered when there are insertions, updates, deletions, or specific changes in database records.Database triggers enable functions to react to changes in data and perform actions such as data validation, data synchronization, or generating notifications.

Message Queue Triggers:

Message queue triggers invoke functions when new messages are added to a message queue.Functions can be connected to message queues such as Amazon Simple Queue Service (SQS) or Azure Service Bus, and they are triggered when new messages are available.Message queue triggers facilitate decoupling and asynchronous processing, allowing functions to handle messages at their own pace.

Streaming Triggers:

Streaming triggers enable functions to process data streams in real-time.Functions can be connected to streaming platforms such as Amazon Kinesis or Apache Kafka, where they receive data records as they are generated.Streaming triggers are suitable for scenarios that require continuous processing of high-volume data streams, such as real-time analytics or event-driven architectures.These triggering mechanisms provide the flexibility to design serverless applications that respond to various events and conditions. By selecting the appropriate triggering mechanisms and configuring the event sources, developers can build reactive and event-driven systems in serverless environments.

VI. CONSIDERATIONS IN PROPOSED DESIGNING APPROACH

In our proposed designing approach for designing effective triggering mechanisms in AWS Lambda for handling more than 15 minutes processing requests, we have consider following considerations which are as follows:

Asynchronous Processing: To handle long-running requests, design your framework to support asynchronous processing. Break down the processing into smaller, manageable tasks that can be executed asynchronously. Implement mechanisms for tracking the progress and status of these tasks.
Job Queues: Utilize job queues to manage and prioritize the processing requests. When a request comes in, enqueue it into a job queue. This allows you to handle requests in a scalable and efficient manner, ensuring that they are processed in the order they were received.
Distributed Processing: Distribute the processing tasks across multiple instances of AWS Lambda functions to achieve parallel processing and improved performance. Partition the workload and assign different segments to individual function instances for concurrent execution.
Request Chunking: If the request payload is large, consider implementing request chunking. Split the request into smaller chunks that can be processed independently. This approach allows you to handle large requests efficiently and avoid any limitations imposed by AWS Lambda, such as maximum payload size.
State Management: Implement a mechanism for managing the state of long-running requests. Store the intermediate state and progress of the request in a persistent storage system, such as a database or an external caching service. This allows you to resume processing if a function invocation is interrupted or times out.
Timeouts and Retries: Set appropriate timeouts for the AWS Lambda functions to handle long-running requests. Implement retry mechanisms to handle failures or timeouts, ensuring that processing continues from the last checkpoint in case of failures. Consider using AWS Step Functions for orchestration and handling retries.
Monitoring and Alerting: Incorporate monitoring and alerting mechanisms to track the progress and performance of long-running requests. Monitor the execution time, resource

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 7
DOI: https://doi.org/10.17762/ijritcc.v11i7.7846
***Article Received**: 29 April 2023 **Revised**: 22 June 2023 **Accepted**: 03 July 2023*

utilization, and any errors or exceptions that occur during processing. Set up alerts to notify administrators or stakeholders in case of any issues.

Error handling: Design a robust error handling mechanism to handle failures during processing. Implement a rollback mechanism to undo any changes made in case of processing errors. Use Error Handling and Rollback compensating transactions or idempotent operations to ensure data consistency and integrity.

Cost Optimization: Consider the cost implications of long-running requests. Design your framework to optimize costs by utilizing AWS Lambda features like provisioned concurrency, which can help reduce cold starts and improve performance. Implement mechanisms to pause or optimize resources during idle periods to minimize costs.

Performance Optimization: Continuously optimize the performance of your framework by monitoring and analyzing the processing times, identifying bottlenecks, and optimizing resource utilization. Consider leveraging AWS Lambda features like provisioned concurrency, memory allocation tuning, and function-specific optimization**[9].**

By considering these design considerations, we have developed a methodology for effective triggering mechanisms in AWS Lambda that can handle long-running processing requests efficiently, ensure fault tolerance, optimize costs, and provide scalability and reliability.

VII. PROPOSED METHODOLOGY

To handle longer requests in AWS Lambda, we have implemented an asynchronous processing framework. AWS Lambda has a maximum execution time limit, which varies depending on the region and the type of Lambda function you're using. By implementing an asynchronous framework, you can overcome this time limit and process longer requests effectively. The methodology that we have followed:

Receive the initial request: When a request comes to your Lambda function, the first step is to receive and validate the request. Make sure the request contains all the necessary information for processing.

Store the request: Instead of processing the request immediately, store it in a durable storage service such as Amazon S3, Amazon DynamoDB, or an external database like Amazon RDS or Aurora. This step ensures that the request is not lost in case of a Lambda function timeout or failure.

Return a response: Acknowledge the receipt of the request by returning an appropriate response to the client. This response can include a unique identifier or a token that the client can use to check the processing status later.

Process the request asynchronously: Trigger another Lambda function or an AWS service like Amazon Simple Queue Service (SQS) or Amazon Simple Notification Service (SNS) to process the request asynchronously. Pass the unique identifier or token from the previous step so that the processing function can retrieve the request.

Implement the processing function: Develop a separate Lambda function or worker that reads the request from the storage service and performs the necessary processing. Depending on the workload, you can process the request entirely within a single function invocation or break it down into smaller tasks for parallel processing.

Update the processing status: As the processing function progresses, update the status of the request in the storage service. You can use a database table, a document in a NoSQL database, or an S3 object to store the status and progress of the request.

Notify completion: Once the processing is complete, notify the client using mechanisms like SNS or by updating a status field in the request storage. The client can then retrieve the processed results using the unique identifier or token.

By implementing this methodology, we can handle longer requests in AWS Lambda by breaking down the processing into asynchronous steps, ensuring reliability, and allowing for efficient utilization of resources.

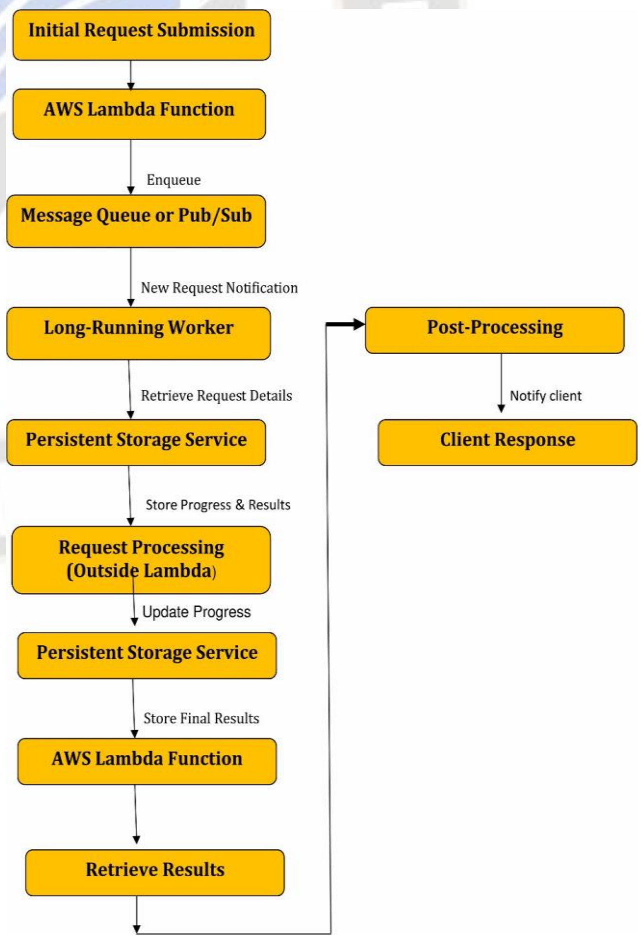

Fig. 2: Proposed Process Flow Diagram

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 7
DOI: https://doi.org/10.17762/ijritcc.v11i7.7846
***Article Received**: 29 April 2023 **Revised**: 22 June 2023 **Accepted**: 03 July 2023*

In the Figure 2 Process Flow diagram:

The initial request is submitted to an AWS Lambda function.
The Lambda function enqueues the request details into a message queue or pub/sub system.
A long-running worker process outside of AWS Lambda listens to the queue or topic and retrieves the request details from the persistent storage service.
The worker process performs the request processing outside of the time constraints of AWS Lambda and updates the progress and results in the persistent storage service.
The Lambda function periodically checks the storage service for completion and retrieves the final results for post-processing.
Finally, the Lambda function notifies the client about the completion of the request, and the client receives the processed results.

This process flow diagram demonstrates the separation of processing between AWS Lambda and a long-running worker process, facilitating the handling of requests that require more than 15 minutes of execution time.

VIII.COMPARISON BETWEEN EXISTING AWS SERVER & OUR PROPOSED METHODOLOGY

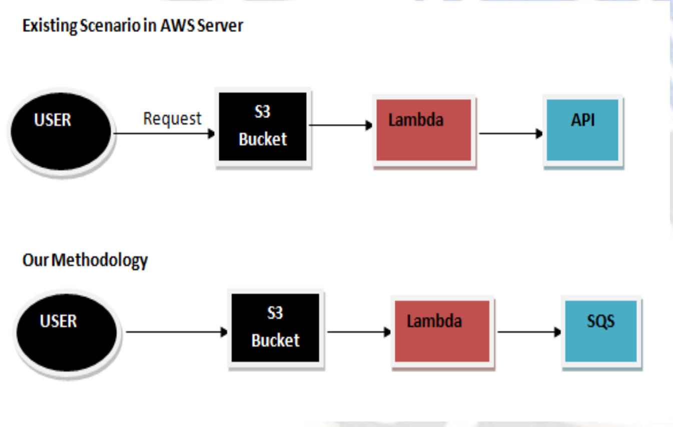

Fig 3: Comparison between Existing AWS Server & Proposed Methodology

In the Fig 3

S3 Bucket

An S3 bucket is a storage resource provided by Amazon Web Services (AWS). It is a scalable and secure object storage service designed to store and retrieve large amounts of data. S3 stands for Simple Storage Service.

SQS

Amazon Simple Queue Service (SQS) is a fully managed message queuing service provided by Amazon Web Services (AWS). It offers a reliable and scalable platform for decoupling and asynchronously processing distributed systems.

AWS Lambda

AWS Lambda is a serverless computing service provided by Amazon Web Services (AWS). It allows developers to run their code without provisioning or managing servers. Lambda executes code in response to events and automatically scales the resources to match the incoming workload.

Timeout Measurement in Existing Scenario in AWS Server

Total Request Processing Time = Request Execution Time+ Uploading Time on Vendor API

In current scenario of AWS Server total execution includes Request Execution Time & time to upload the response on Vendor or user API.

Timeout Measurement in our proposed Methodology

Total Request Processing Time = Request Execution Time & Store in SQS

In our proposed methodology we are saving the time of uploading the response on vendor or user API by saving the response in SQS. In this scenario the stored response will be available to user as per need by giving the unique token number. With this unique token number any user can identify its individual response.

Using Amazon SQS: Our approach utilizes Amazon SQS, a fully managed message queue service, to temporarily store responses.

Time-saving: Storing the response in SQS allows us to avoid the need to upload responses directly to the vendor or user API. This can save time as it offloads the direct processing overhead from the main application server.

Unique Token: Each user's response is associated with a unique token number, enabling easy identification and retrieval of individual responses.

Current AWS Server Working:

Direct API Responses: In the traditional approach, the application server would directly handle responses from vendors or users through APIs. When a user submits a request, the server processes it and responds directly to the user or vendor.

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 7
DOI: https://doi.org/10.17762/ijritcc.v11i7.7846
***Article Received**: 29 April 2023 **Revised**: 22 June 2023 **Accepted**: 03 July 2023*

Real-time Response: With direct API responses, users receive responses in real-time without the need for additional retrieval mechanisms like tokens.

Comparison:

Scalability: Our proposed methodology using SQS can offer better scalability because it decouples the processing of responses from the main application server. As the number of requests increases, SQS can handle the message queue efficiently, allowing the application server to focus on processing new requests.

Resilience: Amazon SQS provides high availability and fault tolerance, ensuring that messages (responses) are not lost even if a server or component fails. This resilience can lead to a more robust system compared to direct API processing.

Asynchronous Processing: Using SQS allows for asynchronous processing, where the application server can send responses to the queue and proceed with other tasks without waiting for a response from the user or vendor API. This can improve overall system responsiveness

IX. EXPERIMENTAL SPECIFICATION:

To perform the experiment on Server less Environment, we have chosen the following specifications, which have been shown in Table1.

S. No.	Requirement	Specifications
1	Programming Language	Node.js
2	Serverless Environment	AWS
3	Serverless Function	AWS Lambda

Table 1: Experimental Specification

X. ALGORITHM

```
async function validateResult(requestId) {
const maxRetries = 10;
let retries = 0;
while (retries &lt; maxRetries) {
// Query the status of the request from the storage service
const status = await queryRequestStatus(requestId);
if (status === &#100;completed&#100;) {
// Request is complete, retrieve the results
const results = await retrieveResults(requestId);
// Perform result validation or additional processing here
// Notify completion to the client
await notifyCompletionToClient(requestId, results);
break; // Exit the loop
} else if (status === &#100;pending&#100;) {
// Request is still in progress, wait before checking again
await wait(5000); // Wait for 5 seconds before the next iteration
} else {
// Handle other status scenarios or errors
break; // Exit the loop
}
retries++;
}
// Handle cases where the request is taking too long or encounters errors
// perform appropriate error handling or cleanup tasks
```

- **Implement helper functions:**

QueryRequestStatus(requested): This function queries the status of the request from the storage service, such as a database or an API endpoint.

RetrieveResults (requestId): This function retrieves the processed results from the storage service based on the request identifier.

Notify CompletionToClient (requestId, results): This function notifies the client about the completion of the request and provides the processed results.

wait(ms): This function creates a delay for the specified number of milliseconds before the next iteration.

- **Invoke the validate Result function:** Trigger the **validateResult** function from our result validation Lambda function or an external service to start the result validation program. Passed the unique identifier or token associated with the request.

By using this methodology, we have implemented a result validation program in Node.js that continuously monitors the status of a request, retrieves the results once available, and performs validation or additional processing as required.

XI. RESULT VALIDATION

Result validation program on node.js framework to handle more than 15 minutes request in AWS lambda server To implement a result validation program in Node.js to handle requests that exceed the 15-minute time limit in AWS Lambda, we have used the following methodology:

- **Initial request submission**: Follow the steps mentioned earlier to store the initial request and return a response to the client with a unique identifier or token.
- **Trigger the result validation program:** After storing the request, trigger a separate Lambda function or an external service to initiate the result validation program. This program will continuously monitor the status of the request and validate the result once it &#100;s available.
- **Result validation loop:** Implemented a loop that periodically checks the status of the request. We have used a combination of setTimeout and async/await to achieve this.

XII. CONCLUSION

In summary, the novel methodology addresses the limitation of AWS Lambda's 15-minute execution time limit by introducing a separate long-running worker process and leveraging persistent storage and asynchronous processing. It provides scalability, flexibility, resilience, and additional post-processing capabilities. On the other hand, the existing framework of AWS Lambda offers simplicity, serverless benefits, and rapid development but is constrained by the 15-minute time limit. The choice between the two frameworks depends on the specific requirements of the processing tasks and the desired trade-offs between development simplicity and long-running processing capabilities.

Acknowledgement

We would like to express our sincere gratitude to our supervisor, Dr. Bineet Kumar Gupta, Associate Professor, DCSIS, IOT, SRMU Lucknow for his valuable guidance and support throughout the research process. His expertise and insights were invaluable in shaping our research and helping us to overcome challenges. We also want to thank our SRMU University,Lucknow for their helpful feedback and support.

References:

[1] Bhamare, S., & Patil, V. (2020). Function Placement and Triggering Techniques in Serverless Computing: A Survey. International Journal of Computer Applications,pp. 975-980.

[2] Mao, H., & Wang, J. (2019). Fine-Grained Function Triggering Mechanism in Serverless Computing. IEEE Transactions on Cloud Computing, 7(2), pp. 446-459.

[3] Zhang, H., Huang, H., & Zhou, H. (2021). Intelligent Function Placement in Serverless Computing. IEEE Transactions on Services Computing, 14(4), pp. 656-669.

[4] P.G. López, M. Sánchez-Artigas, G. París, D.B. Pons, Á.R. Ollobarren, D.A. Pinto, Comparison of faas orchestration systems, in: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), IEEE, 2018, pp. 148–153.

[5] D. Barcelona-Pons, P. García-López, A. Ruiz, A. Gómez-Gómez, G. París, M. Sánchez-Artigas, FaaS orchestration of parallel workloads, in: Proceedings of the 5th International Workshop on Serverless Computing, in: WOSC '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 25–30, http://dx.doi.org/10.1145/3366623.3368137

[6] Chen, Y., & Chen, L. (2020). Survey and Taxonomy on Function Placement Strategies in Serverless Computing. IEEE Access, 8, 178530-178545.

[7] Shin, D., Yoo, S., & Woo, J. (2021). An Event-Driven Function Placement Algorithm for Optimizing Function Invocation Latency in Serverless Computing. Sensors, 21(1), 278.

[8] N.W. Paton, O. Díaz, Active database systems, ACM Comput. Surv. 31 (1) (1999) 63–103.

[9] C. Mitchell, R. Power, J. Li, Oolong: asynchronous distributed applications made easy, in: Proceedings of the Asia-Pacific Workshop on Systems, ACM, 2012, p. 11.

[10] S. Han, S. Ratnasamy, Large-scale computation not at the cost of expressiveness, in: Presented as Part of the 14th Workshop on Hot Topics in Operating Systems, 2013.

[11] A. Geppert, D. Tombros, Event-based distributed workflow execution with EVE, in: Middleware'98, Springer, 1998, pp. 427–442.

[12] Bent AL-Huda Sahib Ghetran, Enas Abdul Hafedh Mohammed. (2023). Bayes Estimation of Parameters of the Kibble-Bivariate Gamma Distribution Under A Precautionary Loss Function for Fuzzy Data Using Simulation. International Journal of Intelligent Systems and Applications in Engineering, 11(2s), 373–380. Retrieved from https://ijisae.org/index.php/IJISAE/article/view/2733

[13] W. Chen, J. Wei, G. Wu, X. Qiao, Developing a concurrent service orchestration engine based on event-driven architecture, in: OTM Confederated International Conferences'' on the Move to Meaningful Internet Systems'', Springer, 2008, pp. 675–690.

[14] W. Binder, I. Constantinescu, B. Faltings, Decentralized orchestration of composite web services, in: 2006 IEEE International Conference on Web Services (ICWS'06), IEEE, 2006, pp. 869–876.

[15] G. Li, H.-A. Jacobsen, Composite subscriptions in content-based publish/subscribe systems, in: ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing, Springer, 2005, pp. 249–269.

[16] D. Dai, Y. Chen, D. Kimpe, R. Ross, Trigger-based incremental data processing with unified sync and async model, IEEE Trans. Cloud Comput. (2018).

[17] P. Soffer, A. Hinze, A. Koschmider, H. Ziekow, C. Di Ciccio, B. Koldehofe, O. Kopp, A. Jacobsen, J. Sürmeli, W. Song, From event streams to process models and back: Challenges and opportunities, Inf. Syst. 81 (2019) 181–200.

[18] I. Baldini, P. Cheng, S.J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, O. Tardieu, The serverless trilemma: Function composition for serverless computing, in: Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2017, 2017, pp. 89–103.

[19] Schad, J., Dittrich, J., & Quiané-Ruiz, J. A. (2018). Towards Function-as-a-Service: Perspectives on Serverless Computing. ACM Queue, 16(2), 70-111.
[20] B. Carver, J. Zhang, A. Wang, Y. Cheng, In search of a fast and efficient serverless DAG engine, 2019, arxiv preprint arXiv:1910.05896.
[21] S. Joyner, M. MacCoss, C. Delimitrou, H. Weatherspoon, Ripple: A practical declarative programming framework for serverless compute, 2020, arxiv preprint arXiv:2001.00222.
[22] Prof. Parvaneh Basaligheh. (2020). Mining Of Deep Web Interfaces Using Multi Stage Web Crawler. International Journal of New Practices in Management and Engineering, 9(04), 11 - 16. Retrieved from http://ijnpme.org/index.php/IJNPME/article/view/94
[23] M. Malawski, A. Gajek, A. Zima, B. Balis, K. Figiela, Serverless execution of scientific workflows: experiments with hyperflow, aws lambda and google cloud functions, Future Generation Comput. Syst. (ISSN: 0167- 739X) 110 (2020) 502–514, http://dx.doi.org/10.1016/j.future.2017.10.029, https://www.sciencedirect.com/science/article/pii/S0167739X1730047X.
[24] A. Jangda, D. Pinckney, Y. Brun, A. Guha, Formal foundations of serverless computing, Proc. ACM Program. Lang. 3 (OOPSLA) (2019) 1–26.
[25] E. Van Eyk, J. Grohmann, S. Eismann, A. Bauer, L. Versluis, L. Toader, N. Schmitt, N. Herbst, C. Abad, A. Iosup, The SPEC-RG reference architecture for FaaS: From microservices and containers to serverless platforms, IEEE Internet Comput. (2019).
[26] Mark White, Thomas Wood, Carlos Rodríguez, Pekka Koskinen, Jónsson Ólafur. Exploring Natural Language Processing in Educational Applications. Kuwait Journal of Machine Learning, 2(1). Retrieved from http://kuwaitjournals.com/index.php/kjml/article/view/168
[27] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, K. Winstein, From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers, in: 2019 USENIX Annual Technical Conference (USENIX ATC 19), USENIX Association, Renton, WA, 2019, pp. 475–488, URL https://www.usenix.org/conference/atc19/presentation/ fouladi.
[28] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, C.S. Meiklejohn, Serverless workflows with durable functions and netherite, 2021, arXiv: 2103.00033
[29] Fernández, P., Tordsson, J., & Elmroth, E. (2019). Event-Driven Function Placement in Serverless Computing. IEEE International Conference on Cloud Engineering (IC2E), 32-37.
[30] Lu, Q., & Zhang, Z. (2020). Serverless Function Placement with Data Locality Optimization in Edge Computing Environment. Future Generation Computer Systems, 108, 136-147.
[31] Wang, W., Zhang, S., & Liu, J. (2021). A Trigger-Based Serverless Function Placement Algorithm for Edge Computing. Journal of Parallel and Distributed Computing, 154, 139-148.
[32] Castro, P., & Cheng, L. T. (2019). Serverless computing: Present and future trends. Journal of Systems and Software, 157, 110381.
[33] Shahrad, M., Yarom, Y., & Falkner, N. (2019). Function Placement in Serverless Computing: From User Requirements to Serverless Frameworks. IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 329-336.
[34] Bineet Kumar Gupta, et. al. “Integrated hesitant fuzzy-based decision-making framework for evaluating sustainable and renewable energy” in International Journal of Data Science and Analytics, ISSN 2364-4168, July 2023, Vlume 7(1), pp-1-12 https://doi.org/10.1007/s41060-023-00426-4,
[35] Kwame Boateng, Machine Learning-based Object Detection and Recognition in Autonomous Systems , Machine Learning Applications Conference Proceedings, Vol 3 2023.
[36] Bineet Kumar Gupta and Satya Bhushan“Containerization and its Architectures: A Study ”Advances in Distributed Computing and Artificial Intelligence Journal Regular Issue, Vol. 11 N. 4 (2022), pp-395-409, eISSN: 2255-2863, DOI: https://doi.org/10.14201/adcaij.28351
[37] Praveen Kumar Singh, Neeraj Kumar & Bineet Kumar Gupta "Smart Card ID: An Evolving and Viable Technology" International Journal of Advanced Computer Science and Applications (IJACSA), ISSN: 2158-107X, Volume 9 (3), pp. 114-124, April 2018. [34] N. Kumar & B. K. Gupta, E-Health Approach to Stipulate The Diabetic Patient Care and Management”, Value Health in The Journal of international Society for Pharmaeconomics and Outcomes Research, Volume 19, Issue 3, Page A211, May 2016, https://doi.org/10.1016/j.jval.2016.03.1281
[38] N. Kumar, B.K. Gupta, V. Sharma, V. Dixit, and S.K. Singh, “E-Health: Stipulation of mobile phone technology in adolescent Diabetic Patient Care” Paediatric Diabetes, Jon Wily & Sons A/A, Volume 14(18), October 2013, ISSN P: 1399-543X, O: 1399-5448, p-90, 90. (DOI:10.1111/pedi.12075).