Serverless Microservices Architecture on AWS

Lakshmikiran Nandula[*], Swathi Chitra Padmanabhan[**]

[*] Senior Lead Software Engineer, Capital One Services, LLC
[**] Lead Software Engineer, Capital One Services, LLC

DOI: 10.29322/IJSRP.13.10.2023.p14205
https://dx.doi.org/10.29322/IJSRP.13.10.2023.p14205

Paper Received Date: 14th August 2023
Paper Acceptance Date: 23rd September 2023
Paper Publication Date: 6th October 2023

Abstract- Architectural Patterns are used to solve common problems with a proven methodology. Serverless Implementation is nothing but leveraging cloud-based resources that are not managed by the teams developing the components. Rather the management of Servers is taken by the Cloud Provider. In this paper, we will explore some Microservices Architectural Patterns and how they can be implemented using Serverless Services within AWS. By no means, this is an exhaustive list of solutions or patterns, but should set a stepping stone for building Serverless Microservices on AWS.

Index Terms- Microservices, Serverless, Architectural Patterns, Amazon Web Services.

I. INTRODUCTION

Microservices can be defined as breaking a large monolithic application into smaller units that can be built independently with loose coupling, the services are usually broken down based on the capabilities that they can provide. Each Microservice is built to scale independently and can use different technologies and infrastructure components. This provides flexibility to the overall business architecture and individual components can be used independently of each other. In the Microservices architecture, one of the greatest advantages is fault tolerance and resiliency, since the functionality is divided into smaller components, the blast radius is reduced to a specific service and the rest of the platform is not impacted.

Microservices can be implemented using different architectural patterns that address the application-specific needs with the functional and non-functional requirements. Applications are usually built based on the business functional requirements, a selected programming language, and technology, infrastructure to deploy the code, and tools to monitor the application and maintain them. Since this is a common methodology for developing an application, multiple application development architecture patterns have evolved. Let's take a look at different Architectural Patterns in which an application can be built and define an application for which we need to lay out an Architecture.

Here are some of the important architectural patterns for building Microservices:

1. Service Decomposition Patterns: Helps define the responsibilities of different services and separates them logically.
    a. Single Responsibility Principle (SRP): Microservices are small cohesive units and autonomous services and each one defines a specific business capability that it solves. For Eg. In a large e-commerce application, Customer Management, Inventory Management, Last mile Delivery Management, can be different microservices.
    b. Domain Driven Design: In Domain Driven Design, services are organized into specific domains, with very specific business contexts. For Eg. Within a Bank, Platforms can be separated by different Business Domains, like Credit Card based Services, Savings Account based services, and Stock Trading based services.
    c. Bounded Contexts: Bounded contexts define a set of guardrails within which a microservice should operate. Bounded contexts help narrow down focus areas within a Domain, this may help identify additional microservices needed within the Domain.

2. Communication Patterns: Helps identify the different ways in which these microservices can communicate.
    a. API-Based Interactions: Microservices can interact with each other via Application Programming Interfaces, these are also known as HTTPS/REST API Calls. This is a communication that is widely used in real-time communication between different services.

   b. Event-Driven Interactions: In non-real-time communications when microservices need to communicate with each other in an Asynchronous manner, we can leverage the Event Driven Patterns like Publish-Subscribe, Event Sourcing, and CQRS (Command Query Responsibility Segregation.)
   c. Message Brokers: In the Publish-Subscribe model, message brokers such as Apache Kafka or RabbitMQ facilitate communication between services.

3. Data Management Patterns: Each Microservice needs the data handling mechanism either as a Storage, Transaction or Passthrough medium.
   a. Database Per Service: This allows each microservice to have its Database, the data model is tightly coupled within the bounded context of the domain. This allows the microservice to scale independently and isolates any issues within the DB scaling within the domain.
   b. Database Replication: To facilitate fault tolerance and resiliency requirements of the application, the data in the application should be replicated. This also helps in designing workloads that are ready heavy or any analytical needs of the Application.
   c. Polyglot persistence: This means any database technology or tool can be selected for different microservices. This provides flexibility to each service to choose the database query approach they need to choose.

4. API Gateway Patterns:
   a. API Gateways allow microservice outside of it's Bounded Context to securely connect and provide tools like Authentication, and Load Balancing before sending a request to the concerned microservice.

5. Observability:
   a. Logging: Each microservice can log its logging data independently for monitoring and doing production support. Different logging Data from each service can be combined to drive a holistic monitoring strategy for the entire Application.
   b. Distributed Tracing: Allows monitoring of each request across different microservices, Distributed Tracing is enabled usually to identify any performance bottlenecks in the architecture.
   c. Health Checks and Circuit Breakers: Health checks periodically monitor the application uptime, along with the Circuit Breaker pattern, these usually help to limit any traffic to the service if it's down.

6. Deployment and Scaling Patterns:
   a. Containerization: Deployment methodology for the microservices, bundle the source code and required packages as part of a Container, and deploy them in the Container Infrastructure.
   b. Orchestration of Services: Deployment of services using technologies like Kubernetes to manage the deployment, scaling, and orchestration of containers.
   c. Serverless: Methodology that leverages servers that are completely managed by the cloud providers.

7. Resiliency Patterns:
   a. Bulkhead Pattern: This prevents the complete system downtime by isolating the failures in one part of the system from affecting others.
   b. Retry & Timeout: Retry and timeouts need to be set in the application to recover from systematic failure and guarantee fulfillment of the request.
   c. Circuit Breaker: This pattern prevents the system from being hit when a system is experiencing failures.

These patterns will help in designing and developing a Scalable, Resilient, and Maintainable application. In the upcoming section, let's dig into an application that can be used to apply the architectural patterns learned so far.

II. Application of Architectural Patterns

In this exercise, let's build an e-commerce application and evaluate the different capabilities it has and how to build such an application using the Architectural patterns above. First, let's break down the application and identify the functional requirements of the application, then apply these patterns while designing this application in the upcoming sections, we can look at how to build this system completely using AWS's Serverless Architecture.

In a typical e-commerce application, the following capabilities exist let's define them below:

1. Product Catalog Management:
   a. Product Categories
   b. Product Information such as Description, Prices, and Specifications.

   c. Product Characteristics such as Sizes and colors.
2. Search and Navigation:
   a. Search Bar with the ability to search by any keyword and the ability to apply filters.
   b. Provide a faceted search with the ability to refine the search by various attributes.
   c. Provide sitemap or breadcrumb navigation for users to track their location within the site.
3. Shopping Cart and Checkout
   a. Provide a Shopping Cart for users to add/remove products.
   b. Provide the ability to navigate between the shopping cart and the rest of the website.
   c. Provide Checkout ability as a regular and guest user.
   d. Offer multiple avenues for users to make the payment.
   e. Provide the ability to apply coupons and promo codes.
   f. Provide the ability to calculate taxes and shipping costs.
   g. Provide the ability to show the estimated delivery date.
4. User Authentication and Profiles:
   a. Provide the ability for users to perform user registration and login capabilities.
   b. Provide the ability for users to create profiles, and save their Address Information, Payment Methods, and Order History.
   c. Provide an ability to recover passwords and provide account safety features.
5. Order Management & Tracking:
   a. Provide order confirmation details via Email and SMS to customers.
   b. Provide the ability for Order Tracking
   c. Provide updates via Email, and SMS for Order Lifecycle.
   d. Provide the users the ability to Cancel or Return the orders.
6. Wishlist & Favorites:
   a. Provide features such as a Wishlist or Favorites.
   b. Provide the ability to share the Wishlist with others.
7. Reviews and Ratings:
   a. Provide the ability to users to leave a review and a rating for the product and service.
   b. Product Reviews and Ratings should be visible on the main product Page.
8. Recommendations and Personalization:
   a. Personalized Product Recommendations based on the user's historical behavior and preferences.
   b. Provide related product suggestions.
9. Inventory Management:
   a. Track Product Availability and Stock Levels.
   b. Showing out-of-stock message when product is unavailable.
   c. Provide an ability to Pre-order and backorder.
10. Customer Support:
   a. Provide multiple ways for customers to reach out to the Platform with any queries.
   b. Provide Live Chat, Email Support, and Self-help sections like FAQs
11. Mobile Responsiveness:
   a. Provide support for Mobile Accessibility and in general accessibility to users.
12. Localization and Internationalization:
   a. Support for Multiple Languages and Currencies if the Platform is going to be a Global Platform.
   b. Prices, taxes, discounts - All need calculations in local currencies.
13. Social Media Integration:
   a. Social Media Integration is needed to promote products on these platforms.
   b. Ability to perform user registration via Social Media Integration.
14. Analytics and Reporting:
   a. Provides insights on the overall Business Strategy, user behaviors, and Conversion Rates.
   b. Reporting to analyze trends and make informed decisions.
15. Content Management:
   a. Static Content Management, Banners, and Landing Pages.
   b. Product Guides, Tutorials, and FAQs.

The capabilities listed above may not be a comprehensive list of capabilities today's e-commerce platforms have, but this should get us started. The actual set of capabilities that any platform can have varies by the business needs and the products that the platform wants to support.

Below is a Diagrammatic representation of this application with different capabilities as a high-level Domain Driven Architecture.

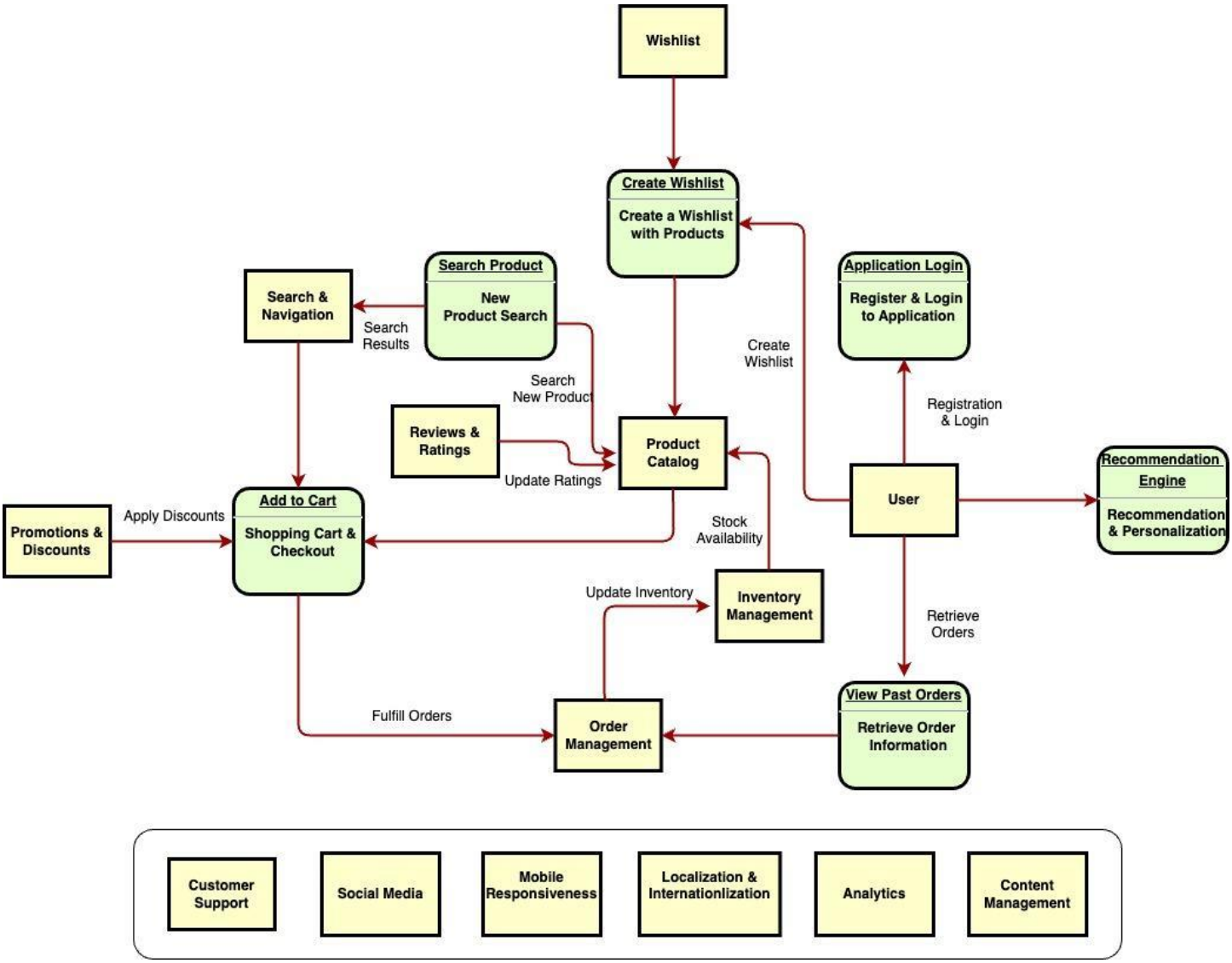

Fig 1

Figure 1 above shows the different interactions within the system, not every interaction within the components is required to be real-time or synchronous, there can be processes that can run asynchronously as well. Let's break down the overall platform into a few smaller interactions and see how we can use the serverless patterns to solve these.

1. API/RESTful Interactions:
    a. Login and Registration
    b. Add Products to Cart
2. Event Driven Interactions:
    a. Analytics
    b. Fulfill Orders

III. Implementation of Serverless Architecture

API/Restful Interactions:

In the case of API/Restful Interactions, two capabilities were outlined, Login & Registration and the other is Add Products to the Cart. Login and registration capabilities can be logically depicted with the help of the diagram below. A User on his laptop is connecting to the ECommerce Website and performing Registration or Login Activity. This view is a very simplified version of the very high-level architecture. Looking at the architecture below, all the interactions seem to be Synchronous and User would get a response back in real time whether their registration is successful or not.

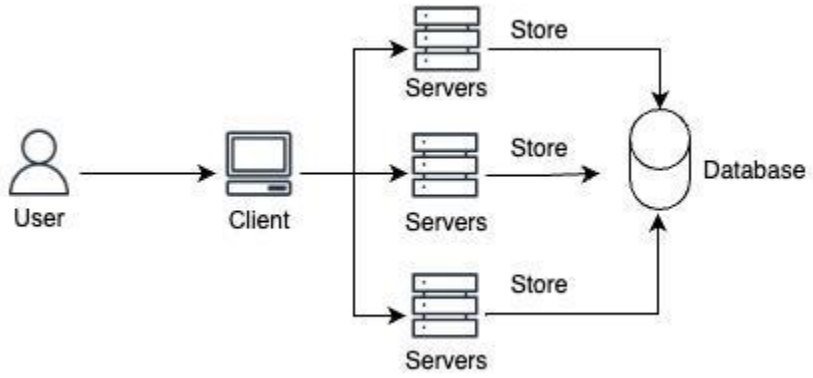

Fig 2

Let's dive deeper and build the same in AWS Architecture and talk about a few Serverless Services that we can leverage from AWS to build

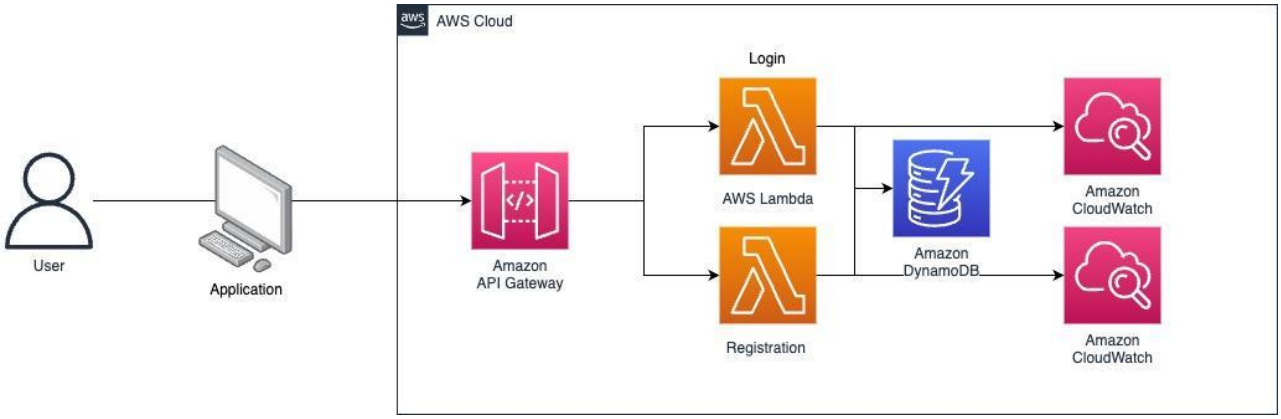

Fig 3

For a Login and registration Interaction, you can refer to the above diagram where services like Amazon API Gateway, AWS Lambda, Amazon DynamoDB, and Amazon Cloud Watch are leveraged. This is kept at a very simple level, in an actual production system you will add a lot of other configs such as VPC (Virtual Private Cloud), Security Groups, IAM (Identity Access Management), and Route 53.

Let's review the Services we used and learn more about them.

API Gateway - As the name suggests, an API Gateway acts as a front-end service that manages the interaction between client applications and backend services. It serves as a centralized entry point for APIs, allowing developers to define and configure how

requests from client applications should be routed to different backed services, including AWS Lambda functions, Amazon EC2 instances, or other HTTP endpoints.

Key features and functions of Amazon API Gateway:

1. API Creation: Developers can define and create RESTful APIs or WebSocket APIs using Amazon API Gateway, specifying endpoints, methods, request/response mappings, and data transformations.
2. Security and Authentication: API Gateway provides various security mechanisms, including authentication and authorization options such as AWS Identity and Access Management (IAM), Amazon Cognito, and custom authorizers. It also supports OAuth and API keys for access control.
3. Traffic Management: Developers can set up API Gateway to handle traffic routing, request/response transformation, and caching, optimizing API performance and scalability.
4. Monitoring and Logging: API Gateway offers comprehensive monitoring and logging capabilities, allowing developers to track API usage, errors, and performance metrics. It integrates with AWS CloudWatch for monitoring and AWS CloudTrail for logging.
5. Throttling and Rate Limiting: You can configure the API Gateway to control traffic and prevent abuse by applying throttling and rate-limiting policies.
6. Cross-Origin Resource Sharing (CORS): API Gateway supports CORS, making it easier to develop APIs that can be consumed by web applications hosted on different domains.
7. Integration with AWS Services: API Gateway seamlessly integrates with various AWS services, enabling developers to build serverless applications using AWS Lambda, connect to AWS services like S3 and DynamoDB, or integrate with AWS Step Functions for workflow automation.
8. Developer Portal: AWS provides a Developer Portal feature that allows organizations to create customized documentation and a developer-friendly interface for their APIs

AWS Lambda - AWS Lambda is a serverless computing service provided by Amazon Web Services (AWS). It allows developers to run code in response to events and triggers without the need to manage servers. AWS Lambda automatically scales the compute capacity to handle incoming traffic and executes code statelessly.

Key features and functionalities of AWS Lambda:

1. Event-Driven: AWS Lambda is designed to execute code in response to various types of events, such as HTTP requests via API Gateway, changes to data in Amazon S3, modifications in DynamoDB tables, or scheduled events triggered by Amazon CloudWatch.
2. Serverless: AWS takes care of managing the underlying infrastructure and developers don’t need to manage the server configuration, patching, or maintenance.
3. Pay-as-You-Go: You are charged based on the number of requests and the execution duration of your code. There are no charges when your code is not running.
4. Stateless Execution: Each Lambda function execution is stateless, meaning it doesn't retain information between invocations. You need to store data in a Database, as in this case, we stored the data in Amazon DynamoDB.
5. Multiple Language Support: Lambda supports multiple programming languages like Node.js, Python, Java, GoLang, C#, and custom runtimes using the AWS Lambda Runtime API.
6. Scalability: Lambda functions can automatically scale horizontally in response to increased workload. This allows your applications that leverage the full ecosystem of AWS Services.
7. Integration: Lambda can be easily integrated with other AWS services, allowing you to build serverless Applications that leverage the full ecosystem of AWS Services.
8. Security & Identity: You can configure AWS Identity and Access Management (IAM) roles for your Lambda functions, controlling what AWS resources they can access and securing your application.
9. Versioning and Aliases: Lambda functions can have multiple versions, and aliases can be created to route traffic to specific versions. This enables you to manage deployments and perform A/B testing seamlessly.
10. Monitoring & Logging: AWS Lambda integrates with AWS CloudWatch, allowing you to monitor function executions, set up alarms, and collect logs for debugging and troubleshooting.
11. Customizable Execution Environment: For certain runtimes, you can customize the execution environment by including additional libraries, dependencies, or configurations.

AWS DynamoDB: Amazon DynamoDB is a fully managed, NoSQL database service provided by Amazon Web Services (AWS). It is designed for high-performance and scalable applications that require fast and predictable performance, and it is particularly well-suited for use cases where flexible, schema-less data storage is necessary.

Key features and characteristics of DynamoDB:

1. NoSQL Database: DynamoDB is a NoSQL database, which means it is designed to handle unstructured or semi-structured data, making it versatile for a wide range of data storage needs.
2. Fully Managed: AWS takes care of the underlying infrastructure, including server provisioning, setup, configuration, scaling, and maintenance, allowing developers to focus on application development rather than database administration.
3. Scalable: DynamoDB is designed to scale horizontally by adding more read and write capacity units to handle increased traffic and data storage needs. This scalability is automatic and on-demand.
4. High Performance: It provides low-latency, single-digit millisecond response times for read and write operations, making it suitable for applications that require fast data access.
5. Built-in Replication: DynamoDB offers built-in data replication across multiple Availability Zones for high availability and fault tolerance. This ensures that data is always available even in the case of hardware failures or network issues.
6. Consistency Models: DynamoDB provides two consistency models: strong consistency and eventual consistency, allowing developers to choose the level of consistency that best suits their application's needs.
7. Security: It integrates with AWS Identity and Access Management (IAM) for fine-grained access control, and it supports encryption at rest and in transit to ensure data security.
8. Automatic Backups and Restore: DynamoDB automatically creates and retains backups of your data, and you can use these backups to restore your tables at any point in time.
9. Global Tables: Global Tables enable you to replicate your data across multiple AWS regions, allowing for global distribution and low-latency access for users around the world.
10. On-demand Pricing: DynamoDB offers a pay-as-you-go pricing model where you are billed only for the read-and-write capacity units and storage you consume, with no upfront costs or long-term commitments.
11. Event-Driven Integration: DynamoDB can be integrated with AWS Lambda and other AWS services, enabling you to build serverless applications that react to changes in your database.
12. Rich Query Capabilities: While DynamoDB is a NoSQL database, it supports rich query capabilities, including secondary indexes, allowing you to query your data in various ways.

Amazon CloudWatch is a monitoring and observability service provided by Amazon Web Services (AWS). It enables developers and system administrators to collect and track metrics, collect and monitor log files, and set alarms. CloudWatch provides insights into the operational health and performance of AWS resources and applications, helping users gain a better understanding of their AWS infrastructure.

Key features and functionalities of AWS CloudWatch:

1. Metrics and Monitoring: CloudWatch allows you to collect and store performance data and operational metrics from various AWS services and resources. These metrics provide insights into the health and performance of your applications and infrastructure.
2. Custom Metrics: You can create custom metrics to monitor specific application or resource-level performance. These metrics can be collected from your applications, on-premises resources, or other cloud environments.
3. Log Monitoring: CloudWatch Logs enables you to collect, monitor, and store log files generated by applications and AWS resources. You can set up alarms and create metric filters to gain insights from log data.
4. Alarms: You can create alarms based on CloudWatch metrics to notify you when specific thresholds are breached. Alarms can trigger actions, such as sending notifications or automatically scaling AWS resources.
5. Dashboards: CloudWatch Dashboards allow you to create customizable, visual representations of your metrics, making it easier to monitor your resources and applications in real time.
6. Events and Event Rules: CloudWatch Events enables you to respond to changes in your AWS environment and automate tasks. You can create event rules to trigger actions when specific events occur.
7. Retention and Storage: You can choose how long to retain your CloudWatch metrics and log data, and CloudWatch provides data archival options for long-term storage.
8. Integration: CloudWatch integrates seamlessly with other AWS services, such as AWS Lambda, EC2, RDS, and more. You can use it to monitor and manage the performance of these services.

9. Global Monitoring: CloudWatch allows you to monitor AWS resources and applications globally, across different AWS regions and accounts.
10. Cost Monitoring: You can use CloudWatch to track your AWS billing and usage data, helping you understand and manage your AWS costs.
11. Application Insights: CloudWatch Application Insights provides automated, end-to-end monitoring for applications using various AWS resources, making it easier to identify and troubleshoot issues.

Based on the definitions and the AWS Architecture above, in simple terms, we can say that when a Customer tries to Register or login from their Computer, the request is routed from API Gateway to AWS Lambda which is a compute function and contains the Business logic of the Application. The Lambda connects to DynamoDB and stores the Registration information or performs a query to the database to validate that login information, before letting the customers log in. Both Registration and Login are synchronous capabilities to happen in real time while the customer is waiting for the next page to load. Cloud Watch is leveraged here to do Application Monitoring, the logs in Cloud Watch can be shipped to Splunk.

Event-Driven Interactions:

Events are nothing but messages coming out of each system that have meaningful information about a certain interaction taking place within the overall system. The events can be used for two main reasons: One to move forward to the next step within the process, and second for Analytical or Monitoring purposes. From Fig 1, we can pick the Fulfill Orders step as a task that can be asynchronously performed, and at the same time, we can also pick the Analytics Horizontal Task to see how to leverage AWS Serverless Components to build these functionalities.

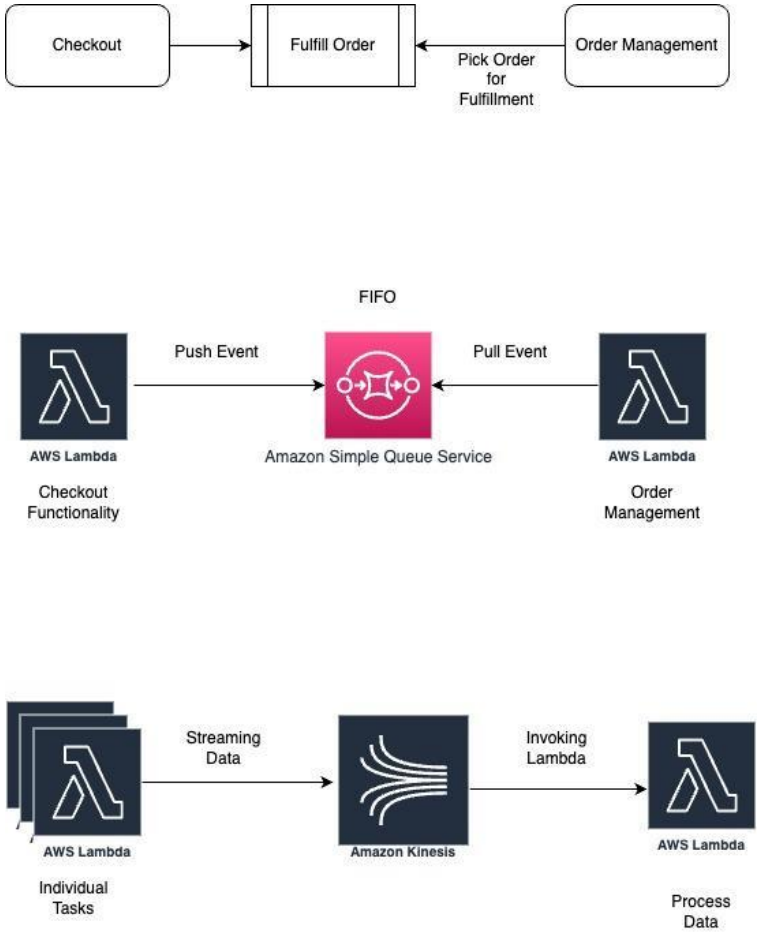

Fig 4

In the above figure, I laid out a very simple architecture depicting the 2 most common use cases of Event-Driven Architecture. In the first AWS Diagram, we can see how a Lambda is leveraged to contain the function knowledge of Checkout capability and a message is dropped in Amazon SQS (Simple Queue Service), which is a Serverless Queuing service. Amazon SQS can support Standard

Queues and FIFO (First in first out) Queues. The messages in the queue can be pulled by a Lambda like Order Management Lambda to process the message.

Functionally, Checkout functionality allows the users to check out the items they would like to purchase and provides the ability to process the payment, at the same time, the entire Order Management need not be taken care of right away and can be delegated to another service. In order to do that, Checkout Lambda drops a message in Amazon SQS and lets Order Management Service pick from the queue and process the order.

In the final diagram in Fig 4, each individual task from Fig 1, Streaming events to Amazon Kinesis for Data Analytics. The visual here is pretty basic again. Complete Analytics can be defined as Log Ingestion, Processing, and Data lookup. In this case, the Ingestion is happening within Amazon Kinesis Data Streams, and the Lambda on the right consumes this data from the Kinesis Data Streams to process those events and write them either to Amazon S3 (Simple Storage System) or Send them to Amazon Cloud Watch to draw insights.

For any business, it's important to draw meaningful insights from their data in order to grow the business and improve the customer experience. Data Analytics through above mentioned way is one such technique to achieve that.

Let's review the Services we used in the above scenario and learn more about them.

Amazon SQS: Amazon Simple Queue Service (Amazon SQS) is a fully managed message queuing service provided by Amazon Web Services (AWS). It is designed to facilitate the decoupling of the components of a cloud application by allowing them to communicate asynchronously.

Key features and functionalities associated with Amazon SQS:

1. Queue: SQS provides a scalable and reliable queue system where messages are stored until they are processed by a consumer (typically an application or service). Queues act as intermediaries between message producers and consumers.
2. Message: A message in SQS is a unit of data that can be up to 256 KB in size. Messages are typically used to convey information between different parts of a distributed system.
3. Producers: Producers are applications or components that send messages to an SQS queue. Messages can be sent to a queue programmatically using AWS SDKs or through various AWS services like AWS Lambda, Amazon S3, etc.
4. Consumers: Consumers are applications or components that retrieve and process messages from an SQS queue. Consumers can be instances of EC2, AWS Lambda functions, or any other component capable of making API calls to SQS.
5. Visibility Timeout: When a consumer retrieves a message from an SQS queue, the message becomes invisible to other consumers for a specified duration called the visibility timeout. This prevents multiple consumers from processing the same message simultaneously.
6. Long Polling: SQS supports long polling, which allows consumers to wait for new messages to arrive in the queue before making a request to retrieve messages. This reduces the number of empty responses and provides more efficient message retrieval.
7. Message Retention: SQS retains messages in a queue for a configurable retention period, which can be from 1 minute to 14 days. Once the retention period expires, messages are automatically deleted.
8. Dead Letter Queue (DLQ): You can configure a DLQ for an SQS queue to store messages that cannot be successfully processed after a certain number of retries. This helps in isolating and troubleshooting message processing errors.
9. FIFO Queues: SQS offers FIFO (First-In-First-Out) queues that guarantee the order in which messages are processed and ensure that each message is processed exactly once.
10. Message Attributes: Messages can include custom metadata in the form of message attributes. These attributes are key-value pairs that provide additional information about the message.
11. Access Control: You can control who can send messages to and receive messages from an SQS queue using AWS Identity and Access Management (IAM) policies.
12. Scaling: SQS is designed to be highly scalable, and it can automatically scale to handle a large number of messages and consumers.

Amazon Kinesis: Amazon Kinesis is a suite of managed services offered by Amazon Web Services (AWS) designed to help process and analyze real-time streaming data at scale. Kinesis provides various tools and capabilities for collecting, processing, and analyzing streaming data, making it a powerful platform for building real-time applications and extracting insights from large volumes of data.

1. Amazon Kinesis Data Streams (KDS) allows you to collect and process real-time data streams. It enables you to ingest data from various sources, such as IoT devices, logs, clickstreams, and more, and then process and analyze that data in real-time.
2. Data is partitioned into shards, and multiple consumers can process the data in parallel. Each shard can handle a certain amount of data throughput.
3. Typical use cases include real-time analytics, data transformation, monitoring, and alerting.

IV. CONCLUSION

In conclusion, we can see how some basic architectural patterns can be implemented using Amazon Web Services. Serverless Technologies are being leveraged in modern-day architectures to be cost-effective and well-managed. The usage of AWS Services is beyond the 2 basic use cases indicated in this paper. In order to take complete benefit of all the Serverless technologies in AWS, go through these services in detail and evaluate how they would fit your use case.

REFERENCES

[1] Microservices on AWS (2023, September 18), htttps://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/microservices-on-aws.html
[2] The Tweleve-Factor App (2023, September 18), https://12factor.net/
[3] Microservices Pattern (2023, September 18), https://microservices.io/patterns/microservices.html
[4] Patterns: Serverless Land (2023, September 18), https://serverlessland.com/patterns

AUTHORS

First Author – Lakshmikiran Nandula, M.S Computer Science, Georgia Institute of Technology and Lakshmikiran.Nandula@gmail.com
Second Author – Swathi Chitra Padmanabhan, B.Tech Mechanical Engineering and Swathichitra.Padmanabhan@gmail.com.