**Architect your microservices in Serverless with right design pattern**

Diagram

Description automatically generated

Serverless has become an increasingly popular paradigm among organizations looking to modernize their applications as it allows them to increase agility while reducing their operational overhead and costs. But the highly distributed nature of serverless architectures requires developers to rethink their approach to application design and development.

AWS-based serverless applications hinge on AWS Lambda functions, which are stateless and ephemeral by design. Serverless is a natural choice for microservice-based architectures because Lambda functions are designed to run small chunks of code in response to events emitted by other services. Lambda also integrates with a range of managed services that can be used to implement common patterns in distributed systems, such as message queues (Amazon Simple Queue Service), APIs (Amazon API Gateway), and event streams (Amazon Kinesis). This helps minimize the typical pain points of building microservices.

*Select appropriate design patterns for workloads*

1. ***Strangler pattern*** allows developers to gradually replace components of their monolith with microservices (which can be implemented with one or more Lambda functions), rather than completely shutting down and replacing their monolith in one go.

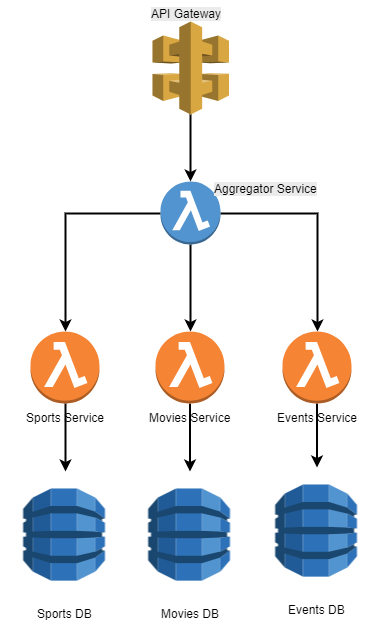
Diagram

Description automatically generated

Strangler pattern

This pattern uses a strangler facade, such as API Gateway, to accept all incoming requests to the legacy system. The facade then routes them to either the legacy application or the new serverless application. Because clients only interact with the facade, they have no knowledge of affected or unaffected services in the backend. Once the entire legacy system has been refactored, and all traffic is routed to the new application, the former can be safely deprecated.

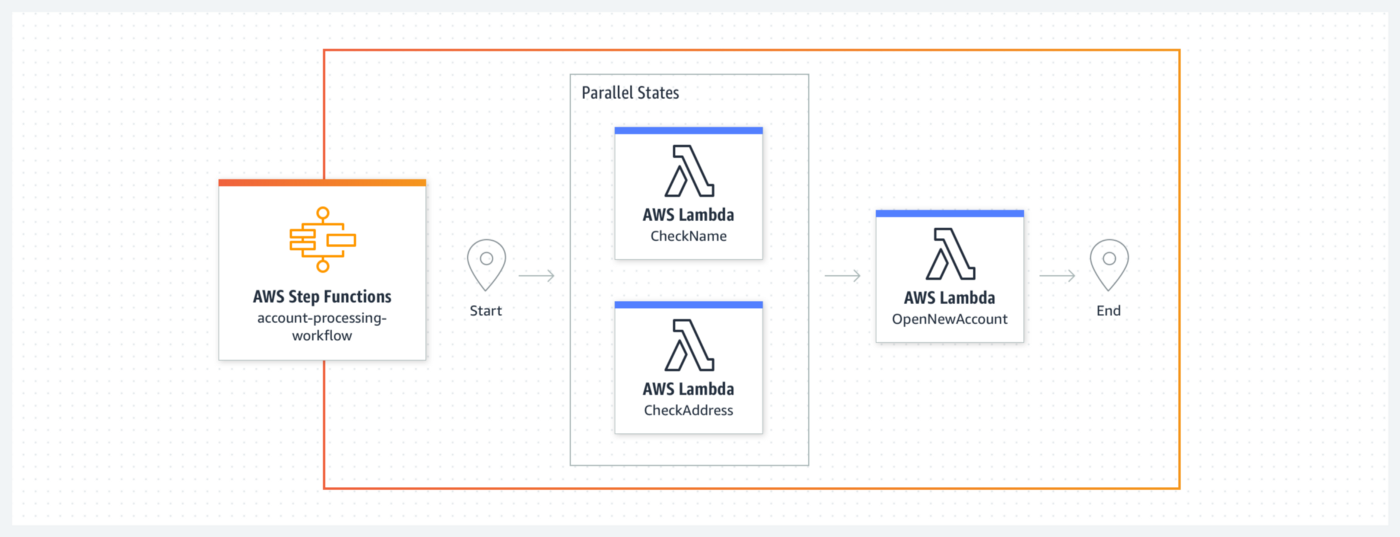
***2. Aggregator pattern*** — In microservice-based architectures, clients often need to make calls to multiple backend services to perform an operation. Because these calls occur over the network, chatty communication between clients and microservices can increase application latency, particularly in situations where bandwidth is limited.



Aggregator pattern

The Aggregator pattern reduces the number of calls clients need to make by using a single Lambda function to accept all client requests. The Lambda function then forwards the requests to the appropriate microservices and third-party APIs, aggregates their results, and returns a single response to the client.

***3. State Machine pattern***— As you build out your application, your business workflows can become highly complex. AWS Step Functions is used to orchestrate complex workflows that involve multiple microservices. Step Functions includes built-in state management, branching, error handling, and retry capabilities, which eliminates the need to write boilerplate code.



State machine pattern

***4. Saga pattern*** — In a microservice-based application, each microservice typically has its own database, which contains data that is closely related to the data in other microservices’ databases. The Saga pattern ensures data consistency by coordinating a sequence of local transactions in interconnected microservices. Once a microservice performs its local transaction, it triggers the next service in the chain to perform its transaction. If a transaction fails along the way, a series of compensating transactions is kicked off to roll back the changes made in prior transactions.

A picture containing diagram

Description automatically generated

Saga pattern

The Saga pattern can be implemented through choreography or orchestration. In the choreography model, each service publishes an event that triggers the next service to run. With orchestration, a central coordinator manages the entire chain of transactions.

***5. Circuit Breaker pattern*** —In distributed systems, where multiple services are involved in fulfilling a request, it is crucial to think about how service failures are handled. Some issues, such as network latency, are intermittent and resolve on their own, so a retry call from an upstream service is likely to succeed. More severe issues or outages, however, may require active intervention and can take an indeterminate amount of time to resolve. Continuous retries in these situations can consume critical resources and starve other services that depend on the same resource pool, which may result in a catastrophic cascading failure.

Diagram

Description automatically generated

Circuit breaker

The Circuit Breaker pattern allows you to build fault tolerance into your system by using an key-value store to keep track of request failures and circuit breaker status, along with a Lambda function to decide whether or not to allow subsequent calls to the impacted service based on the failure count

***It’s so easy to quickly deploy serverless resources. Because of this, we should follow best practices to design, protect our resources and applications.***

Disclaimer: All views and opinions are my own. It is not a direct or indirect view of my current or past employers.