Serverless Application Lens

AWS Well-Architected Framework

December 2019

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided “as is” without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

Contents

Abstract

This document describes the **Serverless Applications Lens** for the AWS Well-Architected Framework. The document covers common serverless applications scenarios and identifies key elements to ensure that your workloads are architected according to best practices.

Introduction

The AWS Well-Architected Framework helps you understand the pros and cons of decisions you make while building systems on AWS.[1] By using the Framework, you will learn architectural best practices for designing and operating reliable, secure, efficient, and cost-effective systems in the cloud. It provides a way for you to consistently measure your architectures against best practices and identify areas for improvement. We believe that having well-architected systems greatly increases the likelihood of business success.

In this “Lens” we focus on how to design, deploy, and architect your **serverless application workloads** in the AWS Cloud. For brevity, we have only covered details from the Well-Architected Framework that are specific to serverless workloads. You should still consider best practices and questions that have not been included in this document when designing your architecture. We recommend that you read the AWS Well-Architected Framework whitepaper.[2]

This document is intended for those in technology roles, such as chief technology officers (CTOs), architects, developers, and operations team members. After reading this document, you will understand AWS best practices and strategies to use when designing architectures for serverless applications.

Definitions

The AWS Well-Architected Framework is based on five pillars: operational excellence, security, reliability, performance efficiency, and cost optimization. For serverless workloads, AWS provides multiple core components (serverless and non-serverless) that allow you to design robust architectures for your serverless applications. In this section, we will present an overview of the services that will be used throughout this document. There are seven areas you should consider when building a serverless workload:

- Compute layer
- Data layer
- Messaging and streaming layer
- User management and identity layer
- Edge layer

- Systems monitoring and deployment
- Deployment approaches

Compute Layer

The compute layer of your workload manages requests from external systems, controlling access and ensuring requests are appropriately authorized. It contains the runtime environment that your business logic will be deployed and executed by.

AWS Lambda lets you run stateless serverless applications on a managed platform that supports microservices architectures, deployment, and management of execution at the function layer.

With **Amazon API Gateway,** you can run a fully managed REST API that integrates with Lambda to execute your business logic and includes traffic management, authorization and access control, monitoring, and API versioning.

AWS Step Functions orchestrates serverless workflows including coordination, state, and function chaining as well as combining long-running executions not supported within Lambda execution limits by breaking into multiple steps or by calling workers running on Amazon Elastic Compute Cloud (Amazon EC2) instances or on-premises.

Data Layer

The data layer of your workload manages persistent storage from within a system. It provides a secure mechanism to store the states that your business logic will need. It provides a mechanism to trigger events in response to data changes.

Amazon DynamoDB helps you build serverless applications by providing a managed NoSQL database for persistent storage. Combined with **DynamoDB Streams,** you can respond in near real time to changes in your DynamoDB table by invoking Lambda functions. **DynamoDB Accelerator** (DAX) adds a highly available in-memory cache for DynamoDB that delivers up to 10x performance improvement from milliseconds to microseconds.

With **Amazon Simple Storage Service** (Amazon S3)**,** you can build serverless web applications and websites by providing a highly available key-value store, from which static assets can be served via a Content Delivery Network (CDN), such as **Amazon CloudFront**.

Amazon Elasticsearch Service (Amazon ES) makes it easy to deploy, secure, operate, and scale Elasticsearch for log analytics, full-text search, application monitoring, and more. Amazon ES is a fully managed service that provides both a search engine and analytics tools.

AWS AppSync is a managed GraphQL service with real-time and offline capabilities, as well as enterprise grade security controls that make developing applications simple. AWS AppSync provides a data-driven API and consistent programming language for applications and devices to connect to services such as DynamoDB, Amazon ES, and Amazon S3.

Messaging and Streaming Layer

The messaging layer of your workload manages communications between components. The streaming layer manages real-time analysis and processing of streaming data.

Amazon Simple Notification Service (Amazon SNS) provides a fully managed messaging service for pub/sub patterns using asynchronous event notifications and mobile push notifications for microservices, distributed systems, and serverless applications.

Amazon Kinesis makes it easy to collect, process, and analyze real-time streaming data. With **Amazon Kinesis Data Analytics,** you can run standard SQL or build entire streaming applications using SQL.

Amazon Kinesis Data Firehose captures, transforms, and loads streaming data into Kinesis Data Analytics, Amazon S3, Amazon Redshift, and Amazon ES, enabling near real-time analytics with existing business intelligence tools.

User Management and Identity Layer

The user management and identity layer of your workload provides identity, authentication, and authorization for both external and internal customers of your workload’s interfaces.

With **Amazon Cognito**, you can easily add user sign-up, sign-in, and data synchronization to serverless applications. **Amazon Cognito** user pools provide built-in sign-in screens and federation with Facebook, Google, Amazon, and Security Assertion Markup Language (SAML). **Amazon Cognito Federated Identities** lets you securely provide scoped access to AWS resources that are part of your serverless architecture.

Edge Layer

The edge layer of your workload manages the presentation layer and connectivity to external customers. It provides an efficient delivery method to external customers residing in distinct geographical locations.

Amazon CloudFront provides a CDN that securely delivers web application content and data with low latency and high transfer speeds.

Systems Monitoring and Deployment

The system monitoring layer of your workload manages system visibility through metrics and creates contextual awareness of how it operates and behaves over time. The deployment layer defines how your workload changes are promoted through a release management process.

With **Amazon CloudWatch**, you can access system metrics on all the AWS services you use, consolidate system and application level logs, and create business key performance indicators (KPIs) as custom metrics for your specific needs. It provides dashboards and alerts that can trigger automated actions on the platform.

AWS X-Ray lets you analyze and debug serverless applications by providing distributed tracing and service maps to easily identify performance bottlenecks by visualizing a request end-to-end.

AWS Serverless Application Model (AWS SAM) is an extension of AWS CloudFormation that is used to package, test, and deploy serverless applications. The AWS SAM CLI can also enable faster debugging cycles when developing Lambda functions locally.

Deployment Approaches

A best practice for deployments in a microservice architecture is to ensure that a change does not break the service contract of the consumer. If the API owner makes a change that breaks the service contract and the consumer is not prepared for it, failures can occur.

Being aware of which consumers are using your APIs is the first step to ensure that deployments are safe. Collecting metadata on consumers and their usage allows you to make data driven decisions about the impact of changes. API Keys are an effective way to capture metadata about the API consumer/clients and often used as a form of contact if a breaking change is made to an API.

Some customers who want to take a risk-adverse approach to breaking changes may choose to clone the API and route customers to a different subdomain (for example, v2.my-service.com) to ensure that existing consumers aren't impacted. While this approach enables new deployments with a new service contract, the tradeoff is that the overhead of maintaining dual APIs (and subsequent backend infrastructure) requires additional overhead.

The table shows the different approaches to deployment:

Deployment	Consumer Impact	Rollback	Event Model Factors	Deployment Speed
All-at-once	All at once	Redeploy older version	Any event model at low concurrency rate	Immediate
Blue/Green	All at once with some level of production environment testing beforehand	Revert traffic to previous environment	Better for async and sync event models at medium concurrency workloads	Minutes to hours of validation and then immediate to customers
Canary/Linear	1–10% typical initial traffic shift, then phased increases or all at once	Revert 100% of traffic to previous deployment	Better for high concurrency workloads	Minutes to hours

All-at-once Deployments

All-at-once deployments involve making changes on top of the existing configuration. An advantage to this style of deployment is that backend changes to data stores, such as a relational database, require a much smaller level of effort to reconcile transactions during the change cycle. While this type of deployment style is low-effort and can be made with little impact in low-concurrency models, it adds risk when it comes to rollback and usually causes downtime. An example scenario to use this deployment model is for development environments where the user impact is minimal.

Blue/Green Deployments

Another traffic shifting pattern is enabling blue/green deployments. This near zero-downtime release enables traffic to shift to the new live environment (green) while still keeping the old production environment (blue) warm in case a rollback is necessary. Since API Gateway allows you to define what percentage of traffic is shifted to a particular environment; this style of deployment can be an effective technique. Since blue/green deployments are designed to reduce downtime, many customers adopt this pattern for production changes.

Serverless architectures that follow the best practice of statelessness and idempotency are amenable to this deployment style because there is no affinity to the underlying infrastructure. You should bias these deployments toward smaller incremental changes so that you can easily roll back to a working environment if necessary.

You need the right indicators in place to know if a rollback is required. As a best practice, we recommend customers using CloudWatch high-resolution metrics, which can monitor in 1-second intervals, and quickly capture downward trends. Used with CloudWatch alarms, you can enable an expedited rollback to occur. CloudWatch metrics can be captured on API Gateway, Step Functions, Lambda (including custom metrics), and DynamoDB.

Canary Deployments

Canary deployments are an ever-increasing way for you to leverage the new release of a software in a controlled environment and enabling rapid deployment cycles. Canary deployments involve deploying a small number of requests to the new change to analyze impact to a small number of your users. Since you no longer need to worry about provisioning and scaling the underlying infrastructure of the new deployment, the AWS Cloud has helped facilitate this adoption.

With Canary deployments in API Gateway, you can deploy a change to your backend endpoint (for example, Lambda) while still maintaining the same API Gateway HTTP endpoint for consumers. In addition, you can also control what percentage of traffic is routed to new deployment and for a controlled traffic cutover. A practical scenario for a canary deployment might be a new website. You can monitor the click-through rates on a small number of end users before shifting all traffic to the new deployment.

Lambda Version Control

Like all software, maintaining versioning enables the quick visibility of previously functioning code as well as the ability to revert back to a previous version if a new deployment is unsuccessful. Lambda allows you to publish one or more immutable versions for individual Lambda functions; such that previous versions cannot be changed. Each Lambda function version has a unique Amazon Resource Name (ARN) and new version changes are auditable as they are recorded in CloudTrail. As a best practice in production, customers should enable versioning to best leverage a reliable architecture.

To simplify deployment operations and reduce the risk of error, Lambda Aliases enable different variations of your Lambda function in your development workflow, such as development, beta, and production. An example of this is when an API Gateway integration with Lambda points to the ARN of a production alias. The production alias will point to a Lambda version. The value of this technique is that it enables a safe deployment when promoting a new version to the live environment because the Lambda Alias within the caller configuration remains static thus less changes to make.

General Design Principles

The Well-Architected Framework identifies a set of general design principles to facilitate good design in the cloud for serverless applications:

- **Speedy, simple, singular**: Functions are concise, short, single purpose and their environment may live up to their request lifecycle. Transactions are efficiently cost aware and thus faster executions are preferred.
- **Think concurrent requests, not total requests**: Serverless applications take advantage of the concurrency model, and tradeoffs at the design level are evaluated based on concurrency.
- **Share nothing**: Function runtime environment and underlying infrastructure are short-lived, therefore local resources such as temporary storage is not guaranteed. State can be manipulated within a state machine execution lifecycle, and persistent storage is preferred for highly durable requirements.
- **Assume no hardware affinity**: Underlying infrastructure may change. Leverage code or dependencies that are hardware-agnostic as CPU flags, for example, may not be available consistently**.**

- **Orchestrate your application with state machines, not functions**: Chaining Lambda executions within the code to orchestrate the workflow of your application results in a monolithic and tightly coupled application. Instead, use a state machine to orchestrate transactions and communication flows.
- **Use events to trigger transactions**: Events such as writing a new Amazon S3 object or an update to a database allow for transaction execution in response to business functionalities. This asynchronous event behavior is often consumer agnostic and drives just-in-time processing to ensure lean service design.
- **Design for failures and duplicates**: Operations triggered from requests/events must be idempotent as failures can occur and a given request/event can be delivered more than once. Include appropriate retries for downstream calls.

Scenarios

In this section, we cover the five key scenarios that are common in many serverless applications and how they influence the design and architecture of your serverless application workloads on AWS. We will present the assumptions we made for each of these scenarios, the common drivers for the design, and a reference architecture of how these scenarios should be implemented.

RESTful Microservices

When building a microservice, you’re thinking about how a business context can be delivered as a re-usable service for your consumers. The specific implementation will be tailored to individual use cases, but there are several common themes across microservices to ensure that your implementation is secure, resilient, and constructed to give the best experience for your customers.

Building serverless microservices on AWS enables you to not only take advantage of the serverless capabilities themselves, but also to use other AWS services and features, as well as the ecosystem of AWS and AWS Partner Network (APN) tools. Serverless technologies are built on top of fault-tolerant infrastructure, enabling you to build reliable services for your mission-critical workloads. The ecosystem of tooling enables you to streamline the build, automate tasks, orchestrate dependencies, and monitor and govern your microservices. Lastly, AWS serverless tools are pay-as-you-go, enabling you to grow the service with your business and keep your costs down during entry phases and non-peak times.

Characteristics:

- You want a secure, easy-to-operate framework that is simple to replicate and has high levels of resiliency and availability.
- You want to log utilization and access patterns to continually improve your backend to support customer usage.
- You are seeking to leverage managed services as much as possible for your platforms, which reduces the heavy lifting associated with managing common platforms including security and scalability.

Reference Architecture

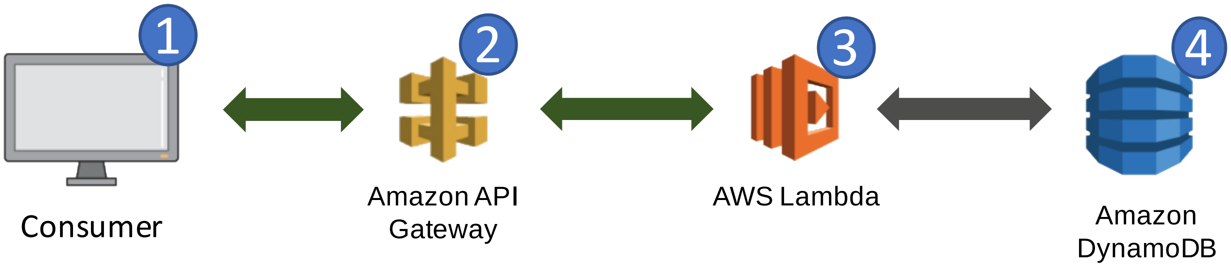


Figure 1: Reference architecture for RESTful microservices

1. **Customers** leverage your microservices by making HTTP API calls. Ideally, your consumers should have a tightly bound service contract to your API to achieve consistent expectations of service levels and change control.
2. **Amazon API Gateway** hosts RESTful HTTP requests and responses to customers. In this scenario, API Gateway provides built-in authorization, throttling, security, fault tolerance, request/response mapping, and performance optimizations.
3. **AWS Lambda** contains the business logic to process incoming API calls and leverage DynamoDB as a persistent storage.
4. **Amazon DynamoDB** persistently stores microservices data and scales based on demand. Since microservices are often designed to do one thing well, a schemaless NoSQL data store is regularly incorporated.

Configuration notes:

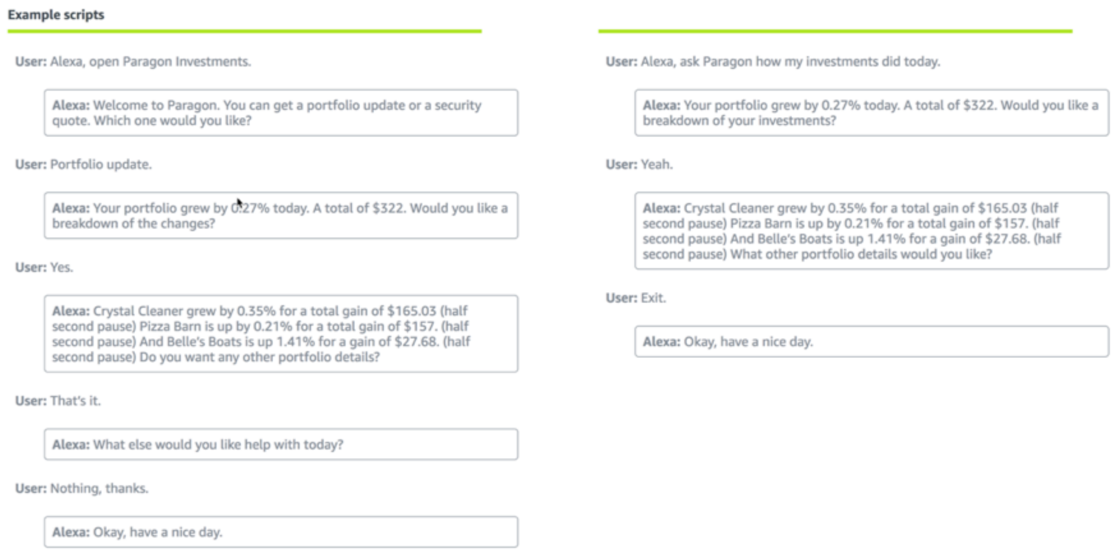
- Leverage API Gateway logging to understand visibility of microservices consumer access behaviors. This information is visible in Amazon CloudWatch Logs and can be quickly viewed through Log Pivots, analyzed in CloudWatch Logs Insights or fed into other searchable engines such as Amazon ES or Amazon S3 (with Amazon Athena). The information delivered gives key visibility, such as:
  - Understanding common customer locations, which may change geographically based on the proximity of your backend
  - Understanding how customer input requests may have an impact on how you partition your database
  - Understanding the semantics of abnormal behavior, which can be a security flag
  - Understanding errors, latency, and cache hits/misses to optimize configuration

This model provides a framework that is easy to deploy and maintain and a secure environment that will scale as your needs grow.

Alexa Skills

The Alexa Skills Kit gives developers the ability to extend Alexa's capabilities by building natural and engaging voice and visual experiences. Successful skills are habit-forming, where users routinely come back because it offers something unique, it provides value in new, novel, and frictionless ways.

The biggest cause of frustration from users is when the skill doesn’t act how they expect it to and it might take multiple interactions before accomplishing what they need. It’s essential to start by designing a voice interaction model and working backwards from that, since some users may say too little, too much, or possibly something you aren’t expecting. The voice design process involves creating, scripting, and planning for expected as well as unexpected utterances.



Example scripts

User: Alexa, open Paragon Investments.

Alexa: Welcome to Paragon. You can get a portfolio update or a security quote. Which one would you like?

User: Portfolio update.

Alexa: Your portfolio grew by 0.27% today. A total of $322. Would you like a breakdown of the changes?

User: Yes.

Alexa: Crystal Cleaner grew by 0.35% for a total gain of $165.03 (half second pause) Pizza Barn is up by 0.21% for a total gain of $157. (half second pause) And Belle's Boats is up 1.41% for a gain of $27.68. (half second pause) Do you want any other portfolio details?

User: That's it.

Alexa: What else would you like help with today?

User: Nothing, thanks.

Alexa: Okay, have a nice day.

User: Alexa, ask Paragon how my investments did today.

Alexa: Your portfolio grew by 0.27% today. A total of $322. Would you like a breakdown of your investments?

User: Yeah.

Alexa: Crystal Cleaner grew by 0.35% for a total gain of $165.03 (half second pause) Pizza Barn is up by 0.21% for a total gain of $157. (half second pause) And Belle's Boats is up 1.41% for a gain of $27.68. (half second pause) What other portfolio details would you like?

User: Exit.

Alexa: Okay, have a nice day.

Figure 2: Alexa Skill example design script

With a basic script in mind, you can use the following techniques before start building a skill:

- **Outline the shortest route to completion**
  - The shortest route to completion is generally when the user gives all information and slots at once, an account is already linked if relevant, and other prerequisites are satisfied in a single invocation of the skill.
- **Outline alternate paths and decision trees**
  - Often, what the user says doesn't include all information necessary to complete the request. In the flow, identify alternate pathways and user decisions.
- **Outline behind-the-scenes decisions the system logic will have to make**
  - Identify behind-the-scenes system decisions, for example with new or returning users. A background system check might change the flow a user follows.
- **Outline how the skill will help** the **user**
  - Include clear directions in the help for what users can do with the skill. Based on the complexity of the skill, the help might provide one simple response or many responses.
- **Outline the account linking process, if present**

   - Determine the information that is required for account linking. You also need to identify how the skill will respond when account linking hasn't been completed.

Characteristics:

- You want to create a complete serverless architecture without managing any instances or servers.
- You want your content to be decoupled from your skill as much as possible.
- You are looking to provide engaging voice experiences exposed as an API to optimize development across wide-ranging Alexa devices, Regions, and languages.
- You want elasticity that scales up and down to meet the demands of users and handles unexpected usage patterns.

Reference Architecture

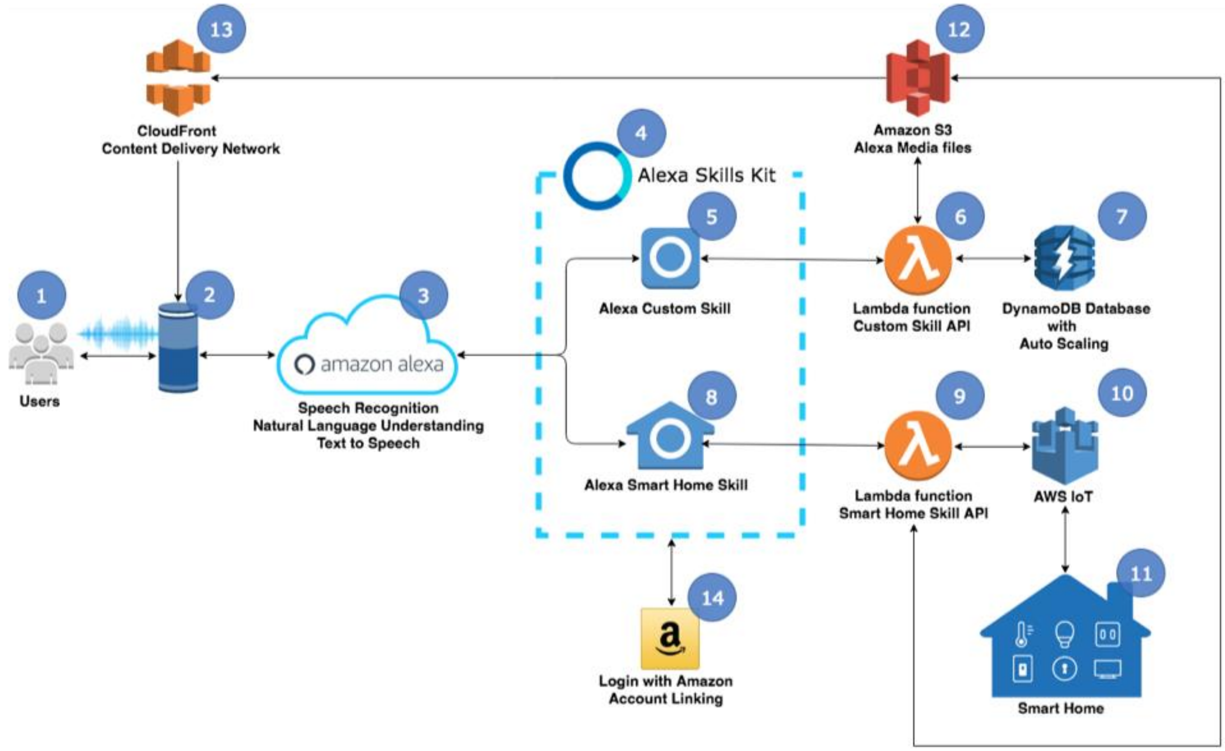


Figure 3: Reference architecture for an Alexa Skill

1. **Alexa users** interact with Alexa skills by speaking to Alexa-enabled devices using voice as the primary method of interaction.

2. **Alexa-enabled devices** listen for a wake word and activate as soon as one is recognized. Supported wake words are Alexa, Computer, and Echo.
3. The **Alexa Service** performs common Speech Language Understanding (SLU) processing on behalf of your Alexa Skill, including Automated Speech Recognition (ASR), Natural Language Understanding (NLU), and Text to Speech (TTS) conversion.
4. **Alexa Skills Kit (ASK)** is a collection of self-service APIs, tools, documentation, and code examples that make it fast and easy for you to add skills to Alexa. ASK is a trusted AWS Lambda trigger, allowing for seamless integration.
5. **Alexa Custom Skill** gives you control over the user experience, allowing you to build a custom interaction model. It is the most flexible type of skill, but also the most complex.
6. A **Lambda function** using the Alexa Skills Kit, allowing you to seamlessly build skills avoiding unneeded complexity. Using it you can process different types of requests sent from the Alexa Service and build speech responses.
7. A **DynamoDB Database** can provide a NoSQL data store that can elastically scale with the usage of your sill. It is commonly used by skills to for persisting user state and sessions.
8. **Alexa Smart Home Skill** allows you to control devices such as lights, thermostats, smart TVs, etc. using the Smart Home API. Smart Home skills are simpler to build that custom skills as the don’t give you control over the interaction model.
9. A **Lambda function** is used to respond to device discovery and control requests from the Alexa Service. Developers use it to control a wide-ranging number of devices including entertainment devices, cameras, lighting, thermostats, locks, and many more.
10. **AWS Internet of Things (IoT)** allows developers to securely connect their devices to AWS and control interaction between their Alexa skill and their devices.
11. An Alexa-enabled **Smart Home** can have an unlimited number of IoT connected devices receiving and responding and to directives from an Alexa Skill.
12. **Amazon S3** stores your skills static assets including images, content, and media. Its contents are securely served using CloudFront.

13. **Amazon CloudFront Content Delivery Network** (CDN) provides a CDN that serves content faster to geographically distributed mobile users and includes security mechanisms to static assets in Amazon S3.
14. **Account Linking** is needed when your skill must authenticate with another system. This action associates the Alexa user with a specific user in the other system.

Configuration notes:

- Validate Smart Home request and response payloads by validating against the JSON schema for all possible Alexa Smart Home messages sent by a skill to Alexa.
- Ensure that your Lambda function timeout is less than eight seconds and can handle requests within that timeframe. (The Alexa Service timeout is 8 seconds.)
- Follow best practices[7] when creating your DynamoDB tables. Use on-demand tables when you are not certain how much read/write capacity you need. Otherwise, choose provisioned capacity with automatic scaling enabled. For Skills that are heavy on ready, DynamoDB Accelerator (DAX) can greatly improve response times.
- Account linking can provide user information that may be stored in an external system. Use that information to provide contextual and personalized experience for your user. Alexa has guidelines on Account Linking to provide frictionless experiences.
- Use the skill beta testing tool to collect early feedback on skill development and for skills versioning to reduce impact on skills that are already live.
- Use ASK CLI to automate skill development and deployment.

Mobile Backend

Users increasingly expect their mobile applications to have a fast, consistent, and feature-rich user experience. At the same time, mobile user patterns are dynamic with unpredictable peak usage and often have a global footprint.

The growing demand from mobile users means that applications need a rich set of mobile services that work together seamlessly without sacrificing control and flexibility of the backend infrastructure. Certain capabilities across mobile applications, are expected by default:

- Ability to query, mutate, and subscribe to database changes
- Offline persistence of data and bandwidth optimizations when connected
- Search, filtering, and discovery of data in applications
- Analytics of user behavior
- Targeted messaging through multiple channels (Push Notifications, SMS, Email)
- Rich content such as images and videos
- Data synchronization across multiple devices and multiple users
- Fine-Grained authorization controls for viewing and manipulating data

Building a serverless mobile backend on AWS enables you to provide these capabilities while automatically managing scalability, elasticity, and availability in an efficient and cost effective way.

Characteristics:

- You want to control application data behavior from the client and explicitly select what data you want from the API
- You want your business logic to be decoupled from your mobile application as much as possible.
- You are looking to provide business functionalities as an API to optimize development across multiple platforms.
- You are seeking to leverage managed services to reduce undifferentiated heavy lifting of maintaining mobile backend infrastructure while providing high levels of scalability and availability.
- You want to optimize your mobile backend costs based upon actual user demand versus paying for idle resources

Reference Architecture

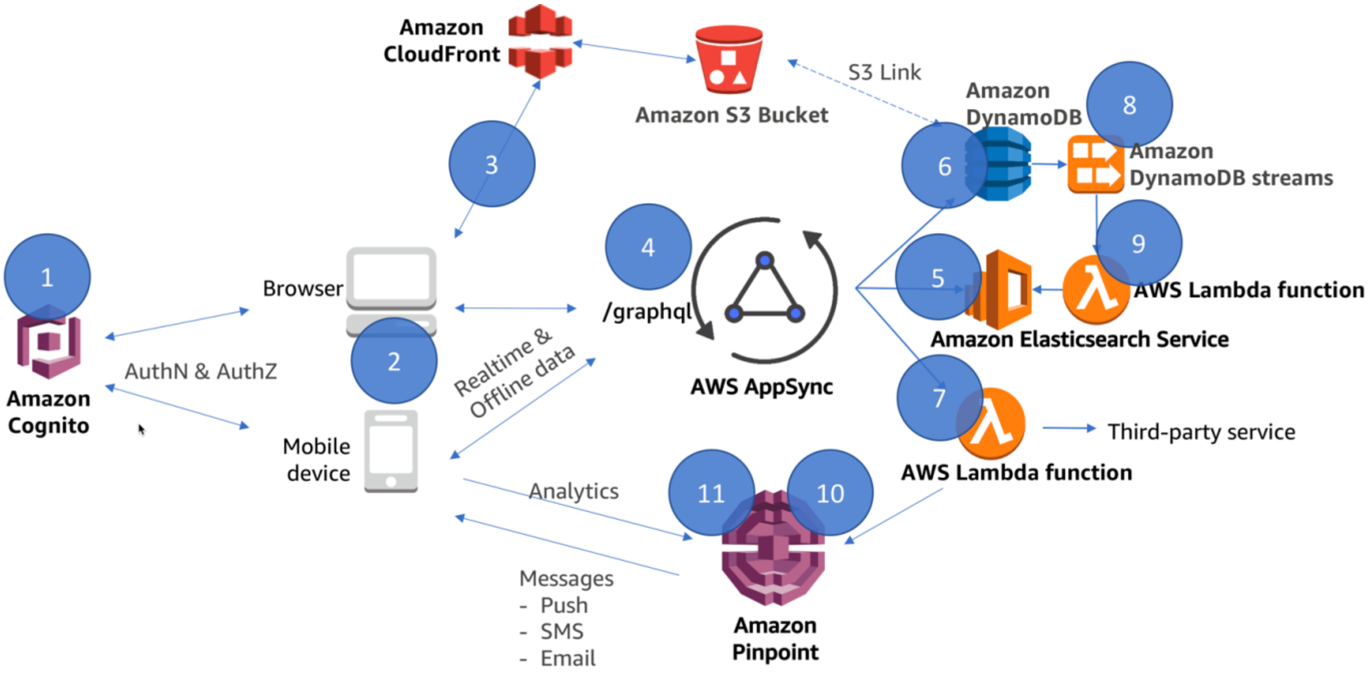


Figure 2: Reference architecture for a mobile backend

1. **Amazon Cognito** is used for user management and as an identity provider for your mobile application. Additionally, it allows mobile users to leverage existing social identities such as Facebook, Twitter, Google+, and Amazon to sign in.
2. **Mobile users** interact with the mobile application backend by performing GraphQL operations against AWS AppSync and AWS service APIs (for example, Amazon S3 and Amazon Cognito).
3. **Amazon S3** stores mobile application static assets including certain mobile user data such as profile images. Its contents are securely served via CloudFront.
4. **AWS AppSync** hosts GraphQL HTTP requests and responses to mobile users. In this scenario, data from AWS AppSync is real-time when devices are connected, and data is available offline as well. Data sources for this scenario are **Amazon DynamoDB**, **Amazon Elasticsearch Service**, or **AWS Lambda** functions
5. **Amazon Elasticsearch Service** acts as a main search engine for your mobile application as well as analytics.
6. **DynamoDB** provides persistent storage for your mobile application, including mechanisms to expire unwanted data from inactive mobile users through a Time to Live (TTL) feature.

7. A **Lambda** function handles interaction with other third-party services, or calling other AWS services for custom flows, which can be part of the GraphQL response to clients.
8. **DynamoDB Streams** captures item-level changes and enables a Lambda function to update additional data sources.
9. A **Lambda** function manages streaming data between DynamoDB and Amazon ES, allowing customers to combine data sources logical GraphQL types and operations.
10. **Amazon Pinpoint** captures analytics from clients, including user sessions and custom metrics for application insights.
11. **Amazon Pinpoint** delivers messages to all users/devices or a targeted subset based on analytics that have been gathered. Messages can be customized and sent using push notifications, email, or SMS channels.

Configuration notes:

- Performance test[3] your Lambda functions with different memory and timeout settings to ensure that you're using the most appropriate resources for the job.
- Follow best practices[4] when creating your DynamoDB tables and consider having AWS AppSync automatically provision them from a GraphQL schema, which will use a well-distributed hash key and create indexes for your operations. Make certain to calculate your read/write capacity and table partitioning to ensure reasonable response times.
- Use the AWS AppSync server-side data caching to optimize your application experience, as all subsequent query requests to your API will be returned from the cache, which means data sources won't be contacted directly unless the TTL expires.
- Follow best practices[5] when managing Amazon ES Domains. Additionally, Amazon ES provides an extensive guide[6] on designing concerning sharding and access patterns that also apply here.
- Use the fine-grained access controls of AWS AppSync, configured in resolvers, to filter GraphQL requests down to the per-user or group level if necessary. This can be applied to AWS Identity and Access Management (IAM) or Amazon Cognito User Pools authorization with AWS AppSync.

- Use AWS Amplify and Amplify CLI to compose and integrate your application with multiple AWS services. Amplify Console also takes care of deploying and managing stacks.

For low-latency requirements where near-to-none business logic is required, Amazon Cognito Federated Identity can provide scoped credentials so that your mobile application can talk directly to an AWS service, for example, when uploading a user's profile picture, retrieve metadata files from Amazon S3 scoped to a user, etc.

Stream Processing

Ingesting and processing real-time streaming data requires scalability and low latency to support a variety of applications such as activity tracking, transaction order processing, click-stream analysis, data cleansing, metrics generation, log filtering, indexing, social media analysis, and IoT device data telemetry and metering. These applications are often spiky and process thousands of events per second.

Using AWS Lambda and Amazon Kinesis, you can build a serverless stream process that automatically scales without provisioning or managing servers. Data processed by AWS Lambda can be stored in DynamoDB and analyzed later.

Characteristics:

- You want to create a complete serverless architecture without managing any instance or server for processing streaming data.
- You want to use the Amazon Kinesis Producer Library (KPL) to take care of data ingestion from a data producer-perspective.

Reference Architecture

Here we are presenting a scenario for common stream processing, which is a reference architecture for analyzing social media data.

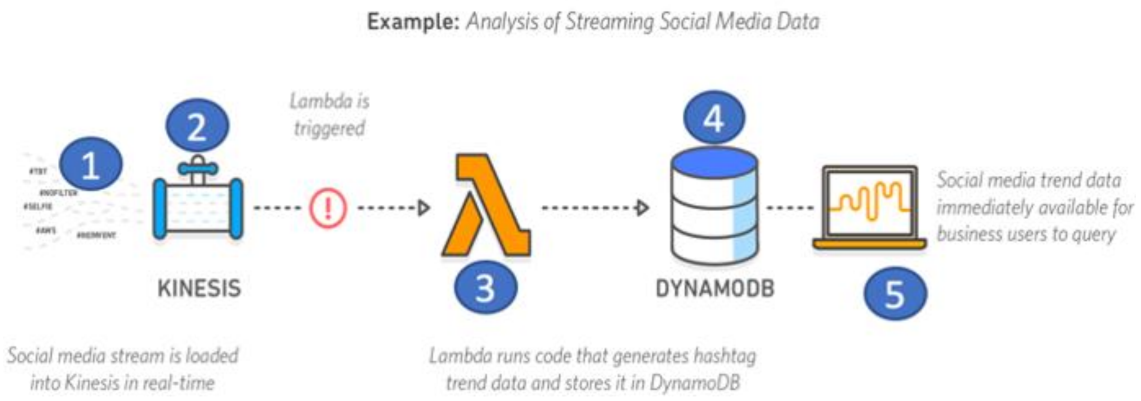

Example: Analysis of Streaming Social Media Data
#TBT
#NOFILTER
#SELFIE
#AWS
#REINVENT
1
2
KINESIS
Lambda is triggered
3
4
DYNAMODB
5
Social media trend data immediately available for business users to query
Social media stream is loaded into Kinesis in real-time
Lambda runs code that generates hashtag trend data and stores it in DynamoDB

Figure 3: Reference architecture for stream processing

1. **Data producers** use the Amazon Kinesis Producer Library (KPL) to send social media streaming data to a Kinesis stream. Amazon Kinesis Agent and custom data producers that leverage the Kinesis API can also be used.
2. An **Amazon Kinesis stream** collects, processes, and analyzes real-time streaming data produced by data producers. Data ingested into the stream can be processed by a consumer, which, in this case, is Lambda
3. **AWS Lambda** acts as a consumer of the stream that receives an array of the ingested data as a single event/invocation. Further processing is carried out by the Lambda function. The transformed data is then stored in a persistent storage, which, in this case, is DynamoDB.
4. **Amazon DynamoDB** provides a fast and flexible NoSQL database service including triggers that can integrate with AWS Lambda to make such data available elsewhere.
5. **Business users** leverage a reporting interface on top of DynamoDB to gather insights out of social media trend data

Configuration notes:

- Follow best practices[7] when re-sharding Kinesis streams to accommodate a higher ingestion rate. Concurrency for stream processing is dictated by the number of shards and by the parallelization factor. Therefore, adjust it according to your throughput requirements.
- Consider reviewing the Streaming Data Solutions whitepaper[8] for batch processing, analytics on streams, and other useful patterns.

- When not using KPL, make certain to take into account partial failures for non-atomic operations, such as PutRecords, since the Kinesis API returns both successfully and unsuccessfully processed records[9] upon ingestion time.
- Duplicated records[10] may occur, and you must leverage both retries and idempotency within your application for both consumers and producers.
- Consider using Kinesis Data Firehose over Lambda when ingested data needs to be continuously loaded into Amazon S3, Amazon Redshift, or Amazon ES.
- Consider using Kinesis Data Analytics over Lambda when standard SQL could be used to query streaming data, and load only its results into Amazon S3, Amazon Redshift, Amazon ES, or Kinesis Streams.
- Follow best practices for AWS Lambda stream-based invocation[11] since that covers the effects on batch size, concurrency per shard, and monitoring stream processing in more detail.
- Use Lambda maximum retry attempts, maximum record age, bisect batch on function error, and on-failure destination error controls to build more resilient stream processing applications.

Web Application

Web applications typically have demanding requirements to ensure a consistent, secure, and reliable user experience. To ensure high availability, global availability, and the ability to scale to thousands or potentially millions of users, you often had to reserve substantial excess capacity to handle web requests at their highest anticipated demand. This often required managing fleets of servers and additional infrastructure components which, in turn, led to significant capital expenditures and long lead times for capacity provisioning.

Using serverless computing on AWS, you can deploy your entire web application stack without performing the undifferentiated heavy lifting of managing servers, guessing at provisioning capacity, or paying for idle resources. Additionally, you do not have to compromise on security, reliability, or performance.

Characteristics:

- You want a scalable web application that can go global in minutes with a high level of resiliency and availability.
- You want a consistent user experience with adequate response times.

- You are seeking to leverage managed services as much as possible for your platforms to limit the heavy lifting associated with managing common platforms.
- You want to optimize your costs based on actual user demand versus paying for idle resources.
- You want to create a framework that is easy to set up and operate, and that you can extend with limited impact later.

Reference Architecture

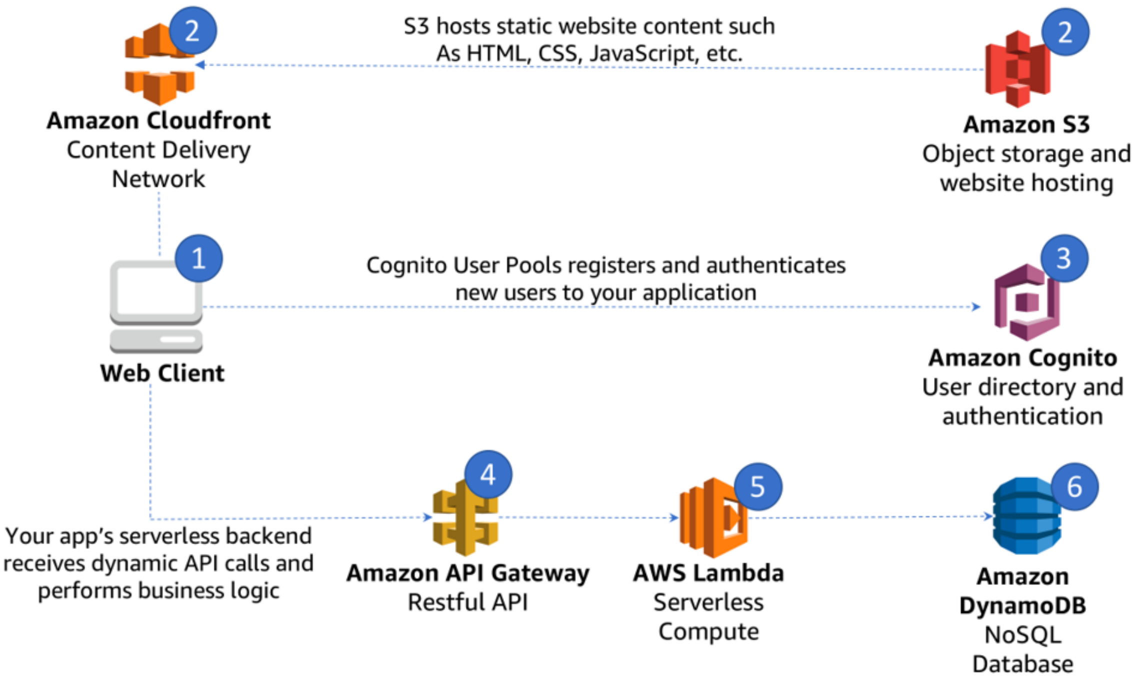

2
S3 hosts static website content such As HTML, CSS, JavaScript, etc.
2
Amazon Cloudfront
Content Delivery Network
Amazon S3
Object storage and website hosting
1
Cognito User Pools registers and authenticates new users to your application
3
Web Client
Amazon Cognito
User directory and authentication
4
5
6
Your app's serverless backend receives dynamic API calls and performs business logic
Amazon API Gateway
Restful API
AWS Lambda
Serverless Compute
Amazon DynamoDB
NoSQL Database

Figure 4: Reference architecture for a web application

1. **Consumers** of this web application might be geographically concentrated or distributed worldwide. Leveraging Amazon CloudFront not only provides a better performance experience for these consumers through caching and optimal origin routing, but also limits redundant calls to your backend.
2. **Amazon S3** hosts web application static assets and is securely served through CloudFront.
3. An **Amazon Cognito user pool** provides user management and identity provider features for your web application.

4. In many scenarios, as static content from Amazon S3 is downloaded by the consumer, dynamic content needs to be sent to or received by your application. For example, when a user submits data through a form, **Amazon API Gateway** serves as the secure endpoint to make these calls and return responses displayed through your web application.
5. An **AWS Lambda** function provides create, read, update, and delete (CRUD) operations on top of DynamoDB for your web application.
6. **Amazon DynamoDB** can provide the backend NoSQL data store to elastically scale with the traffic of your web application.

Configuration Notes:

- Follow best practices for deploying your serverless web application frontend on AWS. More information can be found in the operational excellence pillar.
- For single-page web applications, use AWS Amplify Console to manage atomic deployments, cache expiration, custom domain, and user interface (UI) testing.
- Refer to the security pillar for recommendations on authentication and authorization.
- Refer to the RESTful Microservices scenario for recommendations on web application backend.
- For web applications that offer personalized services, you can leverage API Gateway usage plans[12] as well as Amazon Cognito user pools to scope what different sets of users have access to. For example, a premium user can have higher throughput for API calls, access to additional APIs, additional storage, etc.
- Refer to the Mobile Backend scenario if your application uses search capabilities that are not covered in this scenario.

The Pillars of the Well-Architected Framework

This section describes each of the pillars, and includes definitions, best practices, questions, considerations, and key AWS services that are relevant when architecting solutions for serverless applications.

For brevity, we have only selected the questions from the Well-Architected Framework that are specific to serverless workloads. Questions that have not been included in this document should still be considered when designing your architecture. We recommend that you read the AWS Well-Architected Framework whitepaper.

Operational Excellence Pillar

The **operational excellence** pillar includes the ability to run and monitor systems to deliver business value and to continually improve supporting processes and procedures.

Definition

There are three best practice areas for operational excellence in the cloud:

- Prepare
- Operate
- Evolve

In addition to what is covered by the Well-Architected Framework concerning processes, runbooks, and game days, there are specific areas you should look into to drive operational excellence within serverless applications.

Best Practices

Prepare

There are no operational practices unique to serverless applications that belong to this subsection.

Operate

OPS 1: How do you understand the health of your Serverless application?

Metrics and Alerts

It’s important to understand Amazon CloudWatch Metrics and Dimensions for every AWS service you intend to use so that you can put a plan in a place to assess its behavior and add custom metrics where you see fit.

Amazon CloudWatch provides automated cross service and per service dashboards to help you understand key metrics for the AWS services that you use. For custom metrics, use Amazon CloudWatch Embedded Metric Format to log a batch of metrics that will be processed asynchronously by CloudWatch without impacting the performance of your Serverless application.

The following guidelines can be used whether you are creating a dashboard or looking to formulate a plan for new and existing applications when it comes to metrics:

- **Business Metrics**
  - Business KPIs that will measure your application performance against business goals and are important to know when something is critically affecting your overall business, revenue wise or not.
  - **Examples**: Orders placed, debit/credit card operations, flights purchased, etc.
- **Customer Experience Metrics**
  - Customer experience data dictates not only the overall effectiveness of its UI/UX but also whether changes or anomalies are affecting customer experience in a particular section of your application. Often times, these are measured in percentiles to prevent outliers when trying to understand the impact over time and how it’s spread across your customer base.
  - **Examples**: Perceived latency, time it takes to add an item to a basket or to check out, page load times, etc.
- **System Metrics**
  - Vendor and application metrics are important to underpin root causes from the previous sections. They also tell you if your systems are healthy, at risk, or already your customers.
  - **Examples**: Percentage of HTTP errors/success, memory utilization, function duration/error/throttling, queue length, stream records length, integration latency, etc.
- **Operational Metrics**
  - Operational metrics are equally important to understand sustainability and maintenance of a given system and crucial to pinpoint how stability progressed/degraded over time.

  - **Examples**: Number of tickets (successful and unsuccessful resolutions, etc.), number of times people on-call were paged, availability, CI/CD pipeline stats (successful/failed deployments, feedback time, cycle and lead time, etc.)

CloudWatch Alarms should be configured at both individual and aggregated levels. An individual-level example is alarming on the *Duration* metric from Lambda or *IntegrationLatency* from API Gateway when invoked through API, since different parts of the application likely have different profiles. In this instance, you can quickly identify a bad deployment that makes a function execute for much longer than usual.

Aggregate-level examples include alarming but is not limited to the following metrics:

- **AWS Lambda**: *Duration*, *Errors*, *Throttling*, and *ConcurrentExecutions*. For stream-based invocations, alert on *IteratorAge*. For Asynchronous invocations, alert on *DeadLetterErrors*.
- **Amazon API Gateway**: *IntegrationLatency, Latency, 5XXError*
- **Application Load Balancer**: *HTTPCode_ELB_5XX_Count, RejectedConnectionCount, HTTPCode_Target_5XX_Count, UnHealthyHostCount, LambdaInternalError, LambdaUserError*
- **AWS AppSync**: 5XX and *Latency*
- **Amazon SQS:** *ApproximateAgeOfOldestMessage*
- **Amazon Kinesis Data Streams:** *ReadProvisionedThroughputExceeded, WriteProvisionedThroughputExceeded, GetRecords.IteratorAgeMilliseconds, PutRecord.Success, PutRecords.Success* (if using Kinesis Producer Library) and *GetRecords.Success*
- **Amazon SNS**: *NumberOfNotificationsFailed, NumberOfNotificationsFilteredOut-InvalidAttributes*
- **Amazon SES:** *Rejects, Bounces, Complaints, Rendering Failures*
- **AWS Step Functions:** *ExecutionThrottled, ExecutionsFailed, ExecutionsTimedOut*
- **Amazon EventBridge:** *FailedInvocations, ThrottledRules*
- **Amazon S3:** *5xxErrors, TotalRequestLatency*
- **Amazon DynamoDB:** *ReadThrottleEvents, WriteThrottleEvents, SystemErrors, ThrottledRequests, UserErrors*

Centralized and structured logging

Standardize your application logging to emit operational information about transactions, correlation identifiers, request identifiers across components, and business outcomes. Use this information to answer arbitrary questions about the state of your workload.

Below is an example of a structured logging using JSON as the output:

```
{
   "timestamp":"2019-11-26 18:17:33,774",
   "level":"INFO",
   "location":"cancel.cancel_booking:45",
   "service":"booking",
   "lambda_function_name":"test",
   "lambda_function_memory_size":"128",
   "lambda_function_arn":"arn:aws:lambda:eu-west-1:
12345678910:function:test",
   "lambda_request_id":"52fdfc07-2182-154f-163f-5f0f9a621d72",
   "cold_start": "true",
   "message":{
      "operation":"update_item",
      "details:":{
         "Attributes":{
            "status":"CANCELLED"
         },
         "ResponseMetadata":{

"RequestId":"G7S3SCFDEMEINPG6AOC6CL5IDNVV4KQNSO5AEMVJF66Q9ASUAAJG",
            "HTTPStatusCode":200,
            "HTTPHeaders":{
               "server":"Server",
               "date":"Thu, 26 Nov 2019 18:17:33 GMT",
               "content-type":"application/x-amz-json-1.0",
               "content-length":"43",
               "connection":"keep-alive",
               "x-amzn-
requestid":"G7S3SCFDEMEINPG6AOC6CL5IDNVV4KQNSO5AEMVJF66Q9ASUAAJG",
               "x-amz-crc32":"1848747586"
            },
            "RetryAttempts":0
         }
      }
```

```
        }
}
```

Centralized logging helps you search and analyze your serverless application logs. Structured logging makes it easier to derive queries to answer arbitrary questions about the health of your application. As your system grows and more logging is ingested, consider using appropriate logging levels and a sampling mechanism to log a small percentage of logs in DEBUG mode.

Distributed Tracing

Similar to non-serverless applications, anomalies can occur at larger scale in distributed systems. Due to the nature of serverless architectures, it's fundamental to have distributed tracing.

Making changes to your serverless application entails many of the same principles of deployment, change, and release management used in traditional workloads. However, there are subtle changes in how you use existing tools to accomplish these principles.

Active tracing with AWS X-Ray should be enabled to provide distributed tracing capabilities as well as to enable visual service maps for faster troubleshooting. X-Ray helps you identify performance degradation and quickly understand anomalies, including latency distributions.

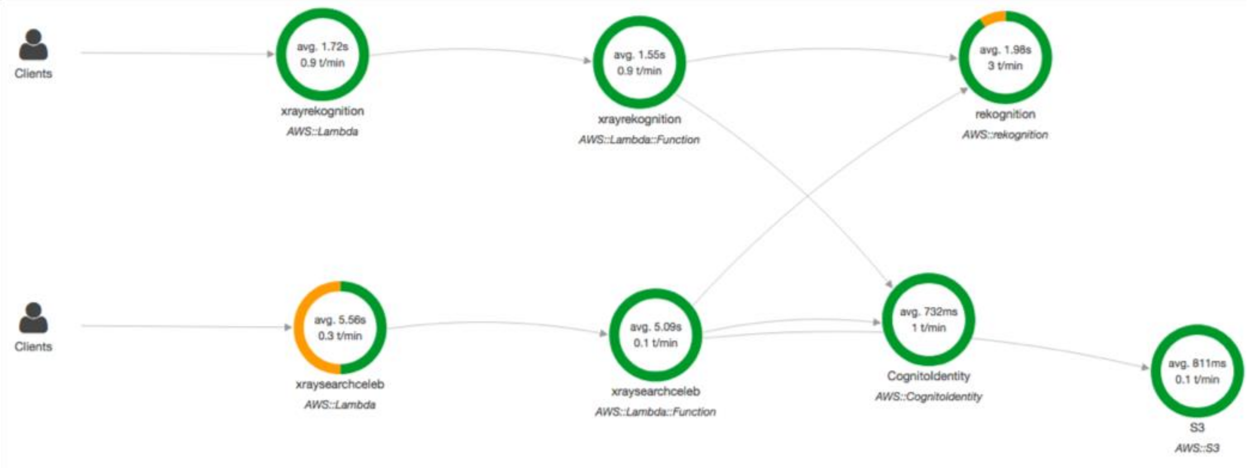


Figure 7: AWS X-Ray Service Map visualizing 2 services

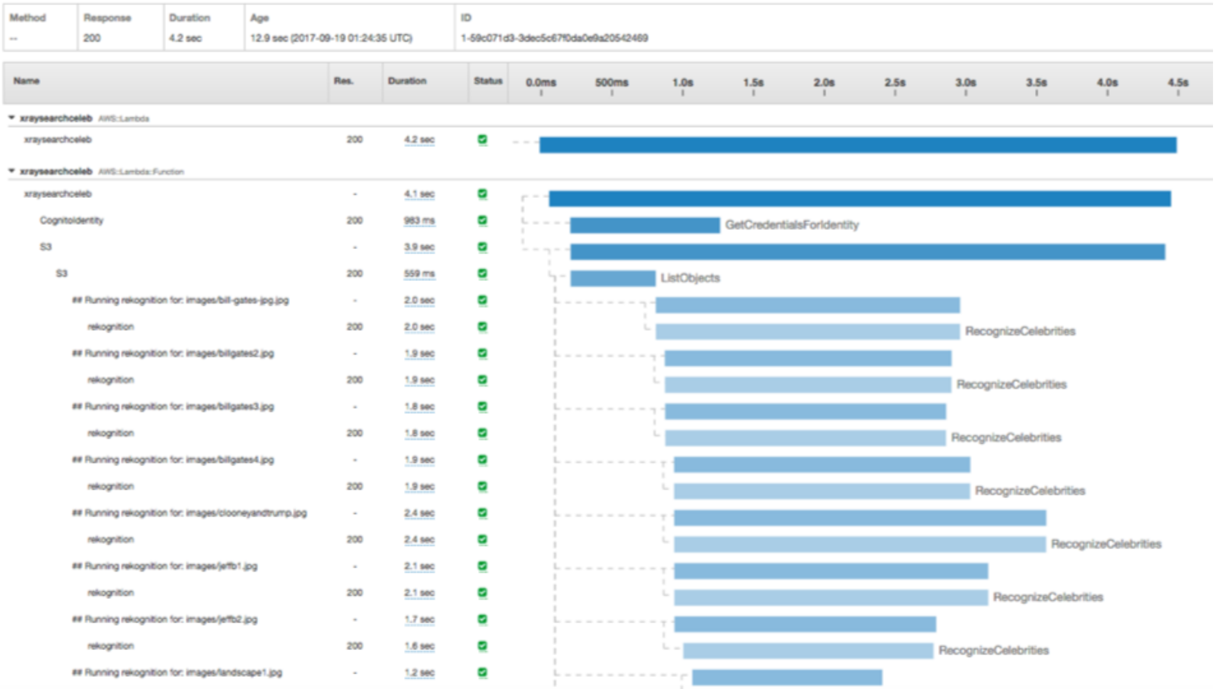
Service Maps are helpful to understand integration points that need attention and resiliency practices. For integration calls, retries, backoffs, and possibly circuit breakers are necessary to prevent faults from propagating to downstream services.

Another example is networking anomalies. You should not rely on default timeouts and retry settings. Instead, tune them to fail fast if a socket read/write timeout happens where the default can be seconds if not minutes in certain clients.

X-Ray also provides two powerful features that can improve the efficiency on identifying anomalies within applications: Annotations and Subsegments.

Subsegments are helpful to understand how application logic is constructed and what external dependencies it has to talk to. Annotations are key-value pairs with string, number, or Boolean values that are automatically indexed by AWS X-Ray.

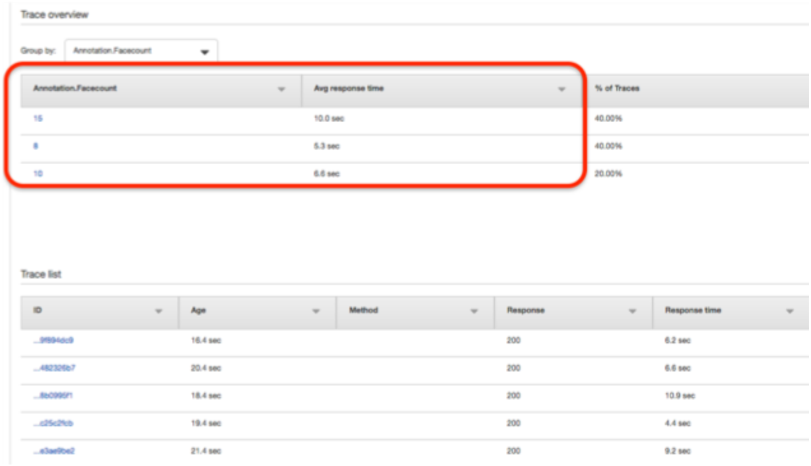
Combined, they can help you quickly identify performance statistics on specific operations and business transactions, for example, how long it takes to query a database, or how long it takes to process pictures with large crowds.



Method	Response	Duration	Age	ID
--	200	4.2 sec	12.9 sec (2017-09-19 01:24:35 UTC)	1-59c071d3-3dec5c67f0da0e9a20542469

Name	Res.	Duration	Status
▾ xraysearchceleb AWS::Lambda			
xraysearchceleb	200	4.2 sec	✓
▾ xraysearchceleb AWS::Lambda::Function			
xraysearchceleb	-	4.1 sec	✓
CognitoIdentity	200	983 ms	✓
S3	-	3.9 sec	✓
S3	200	559 ms	✓
## Running rekognition for: images/bill-gates-jpg.jpg	-	2.0 sec	✓
rekognition	200	2.0 sec	✓
## Running rekognition for: images/billgates2.jpg	-	1.9 sec	✓
rekognition	200	1.9 sec	✓
## Running rekognition for: images/billgates3.jpg	-	1.8 sec	✓
rekognition	200	1.8 sec	✓
## Running rekognition for: images/billgates4.jpg	-	1.9 sec	✓
rekognition	200	1.9 sec	✓
## Running rekognition for: images/clooneyandtrump.jpg	-	2.4 sec	✓
rekognition	200	2.4 sec	✓
## Running rekognition for: images/jeffb1.jpg	-	2.1 sec	✓
rekognition	200	2.1 sec	✓
## Running rekognition for: images/jeffb2.jpg	-	1.7 sec	✓
rekognition	200	1.6 sec	✓
## Running rekognition for: images/landscape1.jpg	-	1.2 sec	✓

Figure 8: AWS X-Ray Trace with subsegments beginning with ##



Trace overview

Group by: Annotation.Facecount

Annotation.Facecount	Avg response time	% of Traces
15	10.0 sec	40.00%
8	5.3 sec	40.00%
10	6.6 sec	20.00%

Trace list

ID	Age	Method	Response	Response time
...9f894dc9	16.4 sec		200	6.2 sec
...482326b7	20.4 sec		200	6.6 sec
...8b0995f1	18.4 sec		200	10.9 sec
...c25c2fcb	19.4 sec		200	4.4 sec
...e3ae9be2	21.4 sec		200	9.2 sec

Figure 9: AWS X-Ray Traces grouped by custom annotations

OPS 2: How do you approach application lifecycle management?

Prototyping

Use infrastructure as code to create temporary environments for new features that you want to prototype, and tear them down as you complete them. You can use dedicated accounts per team or per developer depending, on the size of the team and the level of automation within the organization.

Temporary environments allow for higher fidelity when working with managed services, and increase levels of control to help you gain confidence that your workload integrates and operates as intended.

For configuration management, use environment variables for infrequent changes, such as logging level and database connection strings. Use AWS System Manager Parameter Store for dynamic configuration, such as feature toggles, and store sensitive data using AWS Secrets Manager.

Testing

Testing is commonly done through unit, integration, and acceptance tests. Developing robust testing strategies allows you to emulate your serverless application under different loads and conditions.

Unit tests shouldn’t be different from non-serverless applications and, therefore, can run locally without any changes.

Integration tests shouldn’t mock services you can’t control, since they might change and provide unexpected results. These tests are better performed when using real services because they can provide the same environment a serverless application would use when processing requests in production.

Acceptance or end-to-end tests should be performed without any changes because the primary goal is to simulate the end users’ actions through the available external interface. Therefore, there is no unique recommendation to be aware of here.

In general, Lambda and third-party tools that are available in the AWS Marketplace can be used as a test harness in the context of performance testing. Here are some considerations during performance testing to be aware of:

- Metrics such as invoked max memory used and init duration are available in CloudWatch Logs. For more information, read the performance pillar section.
- If your Lambda function runs inside Amazon Virtual Private Cloud (VPC), pay attention to available IP address space inside your subnet.
- Creating modularized code as separate functions outside of the handler enables more unit-testable functions.
- Establishing externalized connection code (such as a connection pool to a relational database) referenced in the Lambda function’s static constructor/initialization code (that is, global scope, outside the handler) will ensure that external connection thresholds aren’t reached if the Lambda execution environment is reused.
- Use DynamoDB on-demand table unless your performance tests exceed current limits in your account.
- Take into account any other service limits that might be used within your serverless application under performance testing.

Deploying

Use infrastructure as code and version control to enable tracking of changes and releases. Isolate development and production stages in separate environments. This reduces errors caused by manual processes and helps increase levels of control to help you gain confidence that your workload operates as intended.

Use a serverless framework to model, prototype, build, package, and deploy serverless applications, such as AWS SAM or Serverless Framework. With infrastructure as code and a framework, you can parametrize your serverless application and its dependencies to ease deployment across isolated stages and across AWS accounts.

For example, a CI/CD pipeline Beta stage can create the following resources in a beta AWS account and equally for the respective stages you may want to have in different accounts too (Gamma, Dev, Prod): *OrderAPIBeta, OrderServiceBeta, OrderStateMachineBeta, OrderBucketBeta, OrderTableBeta*.

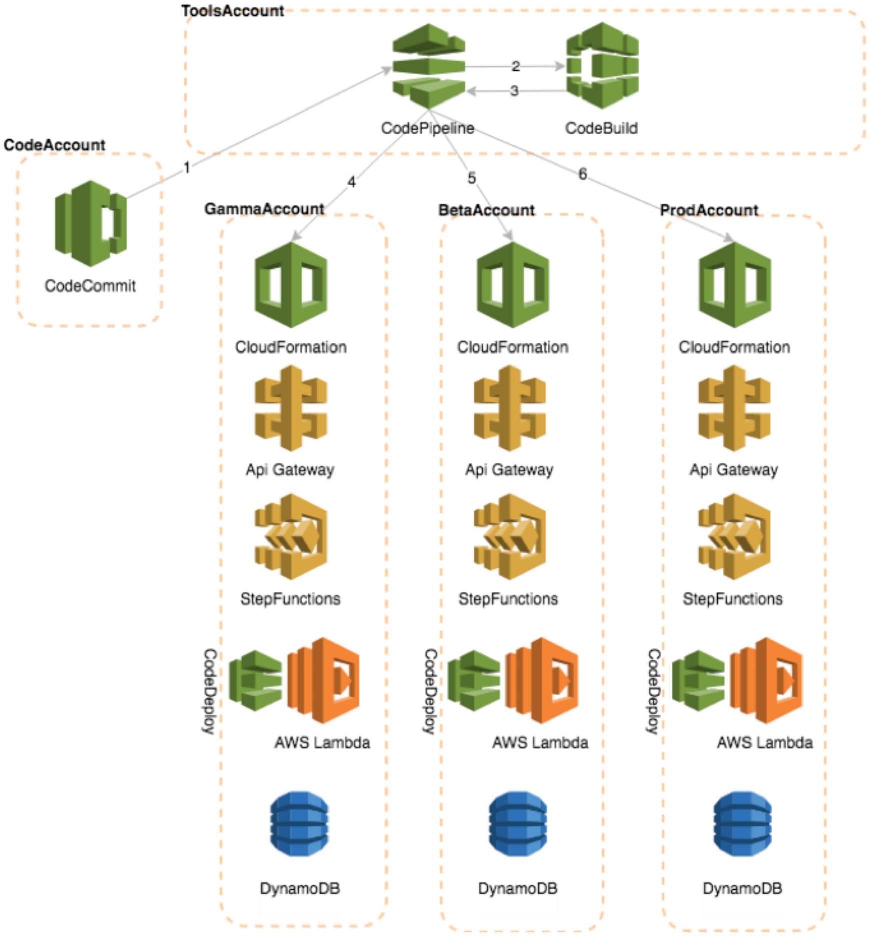


Figure 10: CI/CD Pipeline for multiple accounts

When deploying to production, favor safe deployments over all-at-once systems as new changes will gradually shift over time towards the end user in a canary or linear deployment. Use CodeDeploy hooks (*BeforeAllowTraffic*, *AfterAllowTraffic*) and alarms to gain more control over deployment validation, rollback, and any customization you may need for your application.

You can also combine the use of synthetic traffic, custom metrics, and alerts as part of a rollout deployment. These help you proactively detect errors with new changes that otherwise would have impacted your customer experience.

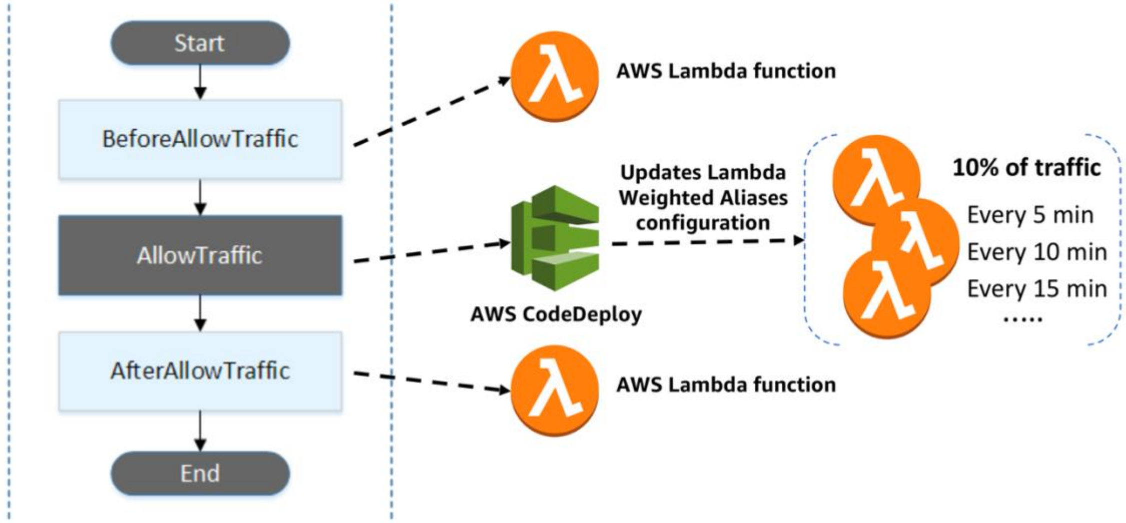


Figure 5: AWS CodeDeploy Lambda deployment and Hooks

Evolve

There are no operational practices unique to serverless applications that belong to this subsection.

Key AWS Services

Key AWS services for operational excellence include AWS Systems Manager Parameter Store, AWS SAM, CloudWatch, AWS CodePipeline, AWS X-Ray, Lambda, and API Gateway.

Resources

Refer to the following resources to learn more about our best practices for operational excellence.

Documentation & Blogs

- API Gateway stage variables[13]
- Lambda environment variables[14]
- AWS SAM CLI[15]

- X-Ray latency distribution[16]
- Troubleshooting Lambda-based applications with X-Ray[17]
- System Manager (SSM) Parameter Store[18]
- Continuous Deployment for Serverless applications blog post[19]
- SamFarm: CI/CD example[20]
- Serverless Application example using CI/CD
- Serverless Application example automating Alerts and Dashboard
- CloudWatch Embedded Metric Format library for Python
- CloudWatch Embedded Metric Format library for Node.js
- Example library to implement tracing, structured logging and custom metrics
- General AWS Limits
- Stackery: Multi-Account Best Practices

Whitepaper

- Practicing Continuous Integration/Continuous Delivery on AWS[21]

Third-Party Tools

- Serverless Developer Tools page including third-party frameworks/tools[22]
- Stelligent: CodePipeline Dashboard for operational metrics

Security Pillar

The **security** pillar includes the ability to protect information, systems, and assets while delivering business value through risk assessments and mitigation strategies.

Definition

There are five best practice areas for security in the cloud:

- Identity and access management
- Detective controls
- Infrastructure protection

- Data protection
- Incident response

Serverless addresses some of today's biggest security concerns as it removes infrastructure management tasks, such as operating system patching, updating binaries, etc. Although the attack surface is reduced compared to non-serverless architectures, the Open Web Application Security Project (OWASP) and application security best practices still apply.

The questions in this section are designed to help you address specific ways an attacker could try to gain access to or exploit misconfigured permissions, which could lead to abuse. The practices described in this section strongly influence the security of your entire cloud platform and so they should be validated carefully and also reviewed frequently.

The **incident response** category will not be described in this document because the practices from the AWS Well-Architected Framework still apply.

Best Practices

Identity and Access Management

SEC 1: How do you control access to your Serverless API?

APIs are often targeted by attackers because of the operations that they can perform and the valuable data they can obtain. There are various security best practices to defend against these attacks.

From an authentication/authorization perspective, there are currently four mechanisms to authorize an API call within API Gateway:

- AWS_IAM authorization
- Amazon Cognito user pools
- API Gateway Lambda authorizer
- Resource policies

Primarily, you want to understand if, and how, any of these mechanisms are implemented. For consumers who currently are located within your AWS environment or have the means to retrieve AWS Identity and Access Management (IAM) temporary credentials to access your environment, you can use AWS_IAM authorization and add least-privileged permissions to the respective IAM role to securely invoke your API.

The following diagram illustrates using AWS_IAM authorization in this context:

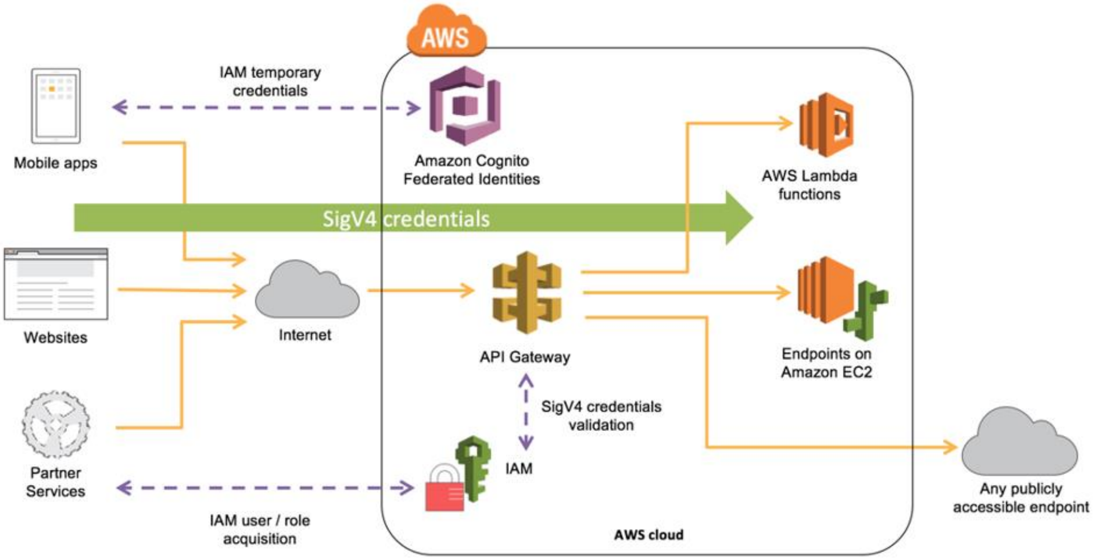

AWS
IAM temporary credentials
Mobile apps
Amazon Cognito Federated Identities
AWS Lambda functions
SigV4 credentials
Websites
Internet
API Gateway
Endpoints on Amazon EC2
SigV4 credentials validation
Partner Services
IAM
IAM user / role acquisition
AWS cloud
Any publicly accessible endpoint

Figure 10: AWS_IAM authorization

If you have an existing Identity Provider (IdP), you can use an API Gateway Lambda authorizer to invoke a Lambda function to authenticate/validate a given user against your IdP. You can use a Lambda authorizer for custom validation logic based on identity metadata.

A Lambda authorizer can send additional information derived from a bearer token or request context values to your backend service. For example, the authorizer can return a map containing user IDs, user names, and scope. By using Lambda authorizers, your backend does not need to map authorization tokens to user-centric data, allowing you to limit the exposure of such information to just the authorization function.

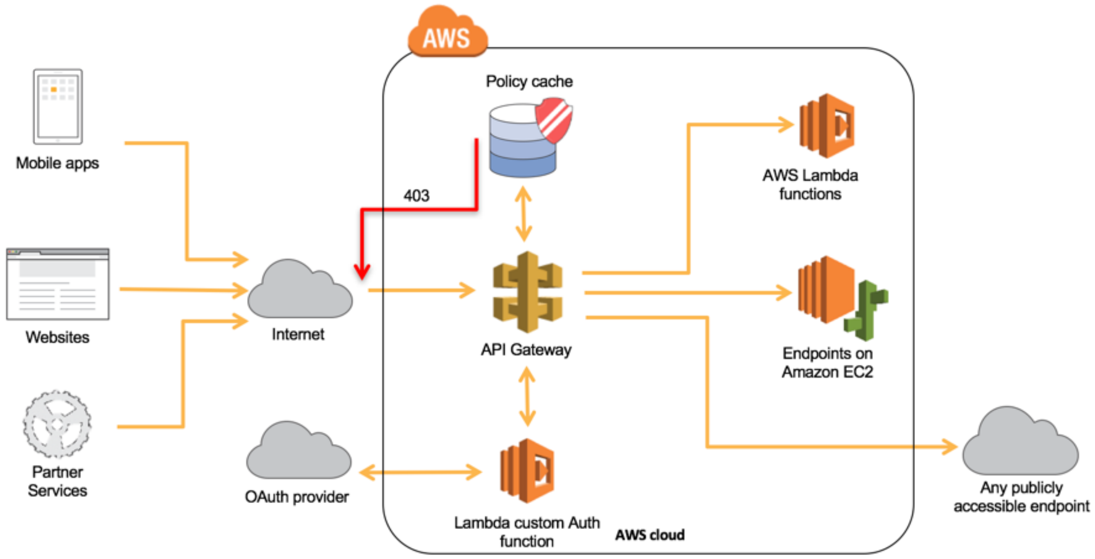


Figure 6: API Gateway Lambda authorizer

If you don't have an IdP, you can leverage Amazon Cognito user pools to either provide built-in user management or integrate with external identity providers, such as Facebook, Twitter, Google+, and Amazon.

This is commonly seen in the mobile backend scenario, where users authenticate by using existing accounts in social media platforms while being able to register/sign in with their email address/username. This approach also provides granular authorization through OAuth Scopes.

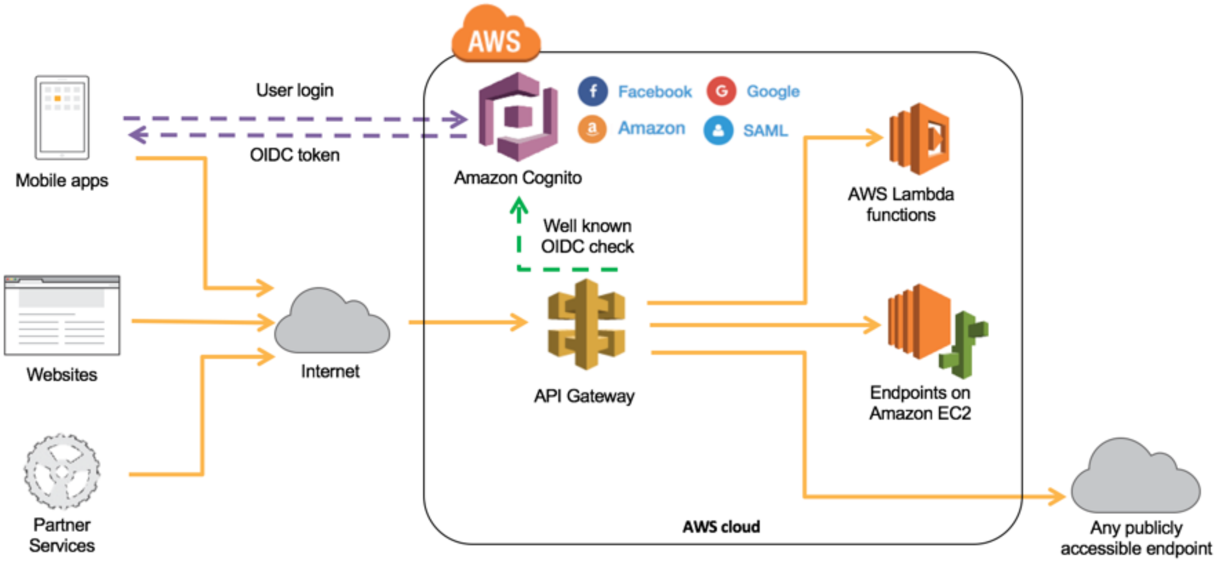


Figure 7: Amazon Cognito user pools

API Gateway API Keys is not a security mechanism and should not be used for authorization unless it's a public API. It should be used primarily to track a consumer's usage across your API and could be used in addition to the authorizers previously mentioned in this section.

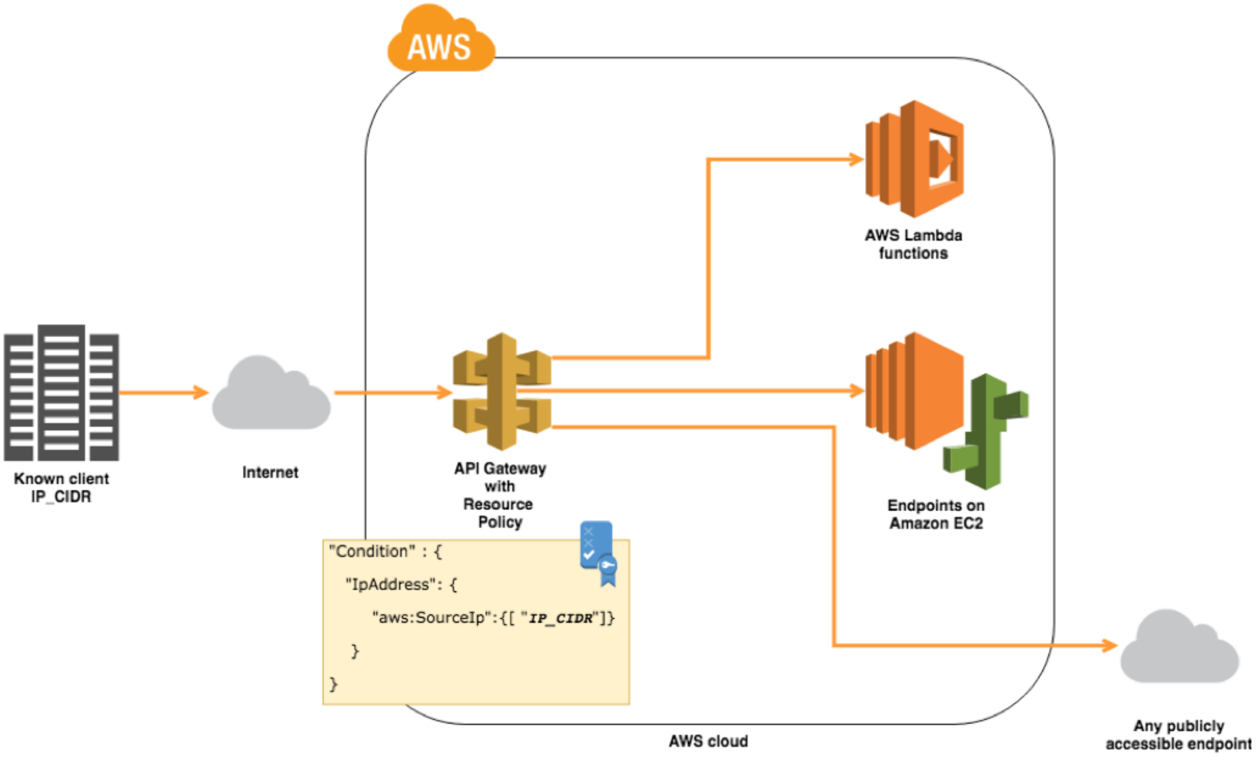
When using Lambda authorizers, we strictly advise against passing credentials or any sort of sensitive data via query string parameters or headers, otherwise you may open your system up to abuse.

Amazon API Gateway resource policies are JSON policy documents that can be attached to an API to control whether a specified AWS Principal can invoke the API.

This mechanism allows you to restrict API invocations by:

- Users from a specified AWS account, or any AWS IAM identity
- Specified source IP address ranges or CIDR blocks
- Specified virtual private clouds (VPCs) or VPC endpoints (in any account)

With resource policies, you can restrict common scenarios, such as only allowing requests coming from known clients with a specific IP range or from another AWS account. If you plan to restrict requests coming from private IP addresses, it's recommended to use API Gateway private endpoints instead.



```
"Condition" : {
  "IpAddress": {
     "aws:SourceIp":{[ "IP_CIDR"]}
  }
}
```

Figure 14: Amazon API Gateway Resource Policy based on IP CIDR

With private endpoints, API Gateway will restrict access to services and resources inside your VPC, or those connected via Direct Connect to your own data centers.

Combining both private endpoints and resource policies, an API can be limited to specific resource invocations within a specific private IP range. This combination is mostly used on internal microservices where they may be in the same or another account.

When it comes to large deployments and multiple AWS accounts, organizations can leverage cross-account Lambda authorizers in API Gateway to reduce maintenance and centralize security practices. For example, API Gateway has the ability to use Amazon Cognito User Pools in a separate account. Lambda authorizers can also be created and managed in a separate account and then re-used across multiple APIs managed by API Gateway. Both scenarios are common for deployments with multiple microservices that need to standardize authorization practices across APIs.

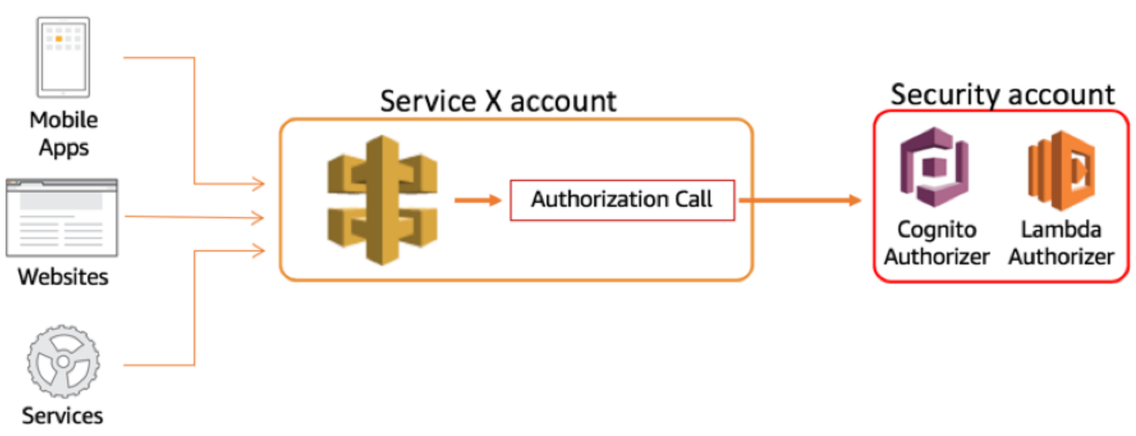


Figure 15: API Gateway Cross-Account Authorizers

SEC 2: How are you managing the security boundaries of your Serverless Application?

With Lambda functions, it’s recommended that you follow least-privileged access and only allow the access needed to perform a given operation. Attaching a role with more permissions than necessary can open up your systems for abuse.

With the security context, having smaller functions that perform scoped activities contribute to a more well-architected serverless application. Regarding IAM roles, sharing an IAM role within more than one Lambda function will likely violate least-privileged access.

Detective Controls

Log management is an important part of a well-architected design for reasons ranging from security/forensics to regulatory or legal requirements.

It is equally important that you track vulnerabilities in application dependencies because attackers can exploit known vulnerabilities found in dependencies regardless of which programming language is used.

For application dependency vulnerability scans, there are several commercial and open-source solutions, such as OWASP Dependency Check, that can integrate within your CI/CD pipeline. It’s important to include all your dependencies, including AWS SDKs, as part of your version control software repository.

Infrastructure Protection

For scenarios where your serverless application needs to interact with other components deployed in a virtual private cloud (VPC) or applications residing on-premises, it's important to ensure that networking boundaries are considered.

Lambda functions can be configured to access resources within a VPC. Control traffic at all layers as described in the AWS Well-Architected Framework. For workloads that require outbound traffic filtering due to compliance reasons, proxies can be used in the same manner that they are applied in non-serverless architectures.

Enforcing networking boundaries solely at the application code level and giving instructions as to what resources one could access is not recommended due to separation of concerns.

For service-to-service communication, favor dynamic authentication, such as temporary credentials with AWS IAM over static keys. API Gateway and AWS AppSync both support IAM Authorization that makes it ideal to protect communication to and from AWS services.

Data Protection

Consider enabling API Gateway Access Logs and selectively choose only what you need, since the logs might contain sensitive data, depending on your serverless application design. For this reason, we recommend that you encrypt any sensitive data traversing your serverless application.

API Gateway and AWS AppSync employ TLS across all communications, clients, and integrations. Although HTTP payloads are encrypted in-transit, request path and query strings that are part of a URL might not be. Therefore, sensitive data can be accidentally exposed via CloudWatch Logs if sent to standard output.

Additionally, malformed or intercepted input can be used as an attack vector—either to gain access to a system or cause a malfunction. Sensitive data should be protected at all times in all layers possible as discussed in detail in the AWS Well-Architected Framework. The recommendations in that whitepaper still apply here.

With regard to API Gateway, sensitive data should be either encrypted at the client-side before making its way as part of an HTTP request, or sent as a payload as part of an HTTP POST request. That also includes encrypting any headers that might contain sensitive data prior to making a given request.

Concerning Lambda functions or any integrations that API Gateway may be configured with, sensitive data should be encrypted before any processing or data manipulation. This will prevent data leakage if such data gets exposed in persistent storage or by standard output that is streamed and persisted by CloudWatch Logs.

In the scenarios described earlier in this document, Lambda functions would persist encrypted data in either DynamoDB, Amazon ES, or Amazon S3 along with encryption at rest. We strictly advise against sending, logging, and storing unencrypted sensitive data, either as part of HTTP request path/query strings or in standard output of a Lambda function.

Enabling logging in API Gateway where sensitive data is unencrypted is also discouraged. As mentioned in the Detective Controls subsection, you should consult your compliance team before enabling API Gateway logging in such cases.

SEC 3: How do you implement Application Security in your workload?

Review security awareness documents authored by AWS Security bulletins and industry threat intelligence as covered in the AWS Well-Architected Framework. OWASP guidelines for application security still apply.

Validate and sanitize inbound events, and perform a security code review as you normally would for non-serverless applications. For API Gateway, set up basic request validation as a first step to ensure that the request adheres to the configured JSON-Schema request model as well as any required parameters in the URI, query string, or headers. Application-specific deep validation should be implemented, whether that is as a separate Lambda function, library, framework, or service.

Store your secrets, such as database passwords or API keys, in a secrets manager that allows for rotation, secure and audited access. Secrets Manager allows fine-grained policies for secrets including auditing.

Key AWS Services

Key AWS services for security are Amazon Cognito, IAM, Lambda, CloudWatch Logs, AWS CloudTrail, AWS CodePipeline, Amazon S3, Amazon ES, DynamoDB, and Amazon Virtual Private Cloud (Amazon VPC).

Resources

Refer to the following resources to learn more about our best practices for security.

Documentation & Blogs

- IAM role for Lambda function with Amazon S3 example[23]
- API Gateway Request Validation[24]
- API Gateway Lambda Authorizers[25]
- Securing API Access with Amazon Cognito Federated Identities, Amazon Cognito User Pools, and Amazon API Gateway[26]
- Configuring VPC Access for AWS Lambda[27]
- Filtering VPC outbound traffic with Squid Proxies[28]
- Using AWS Secrets Manager with Lambda
- Auditing Secrets with AWS Secrets Manager
- OWASP Input validation cheat sheet
- AWS Serverless Security Workshop

Whitepapers

- OWASP Secure Coding Best Practices[29]
- AWS Security Best Practices[30]

Partner Solutions

- PureSec Serverless Security
- Twistlock Serverless Security[31]
- Protego Serverless Security
- Snyk – Commercial Vulnerability DB and Dependency Check[32]
- Using Hashicorp Vault with Lambda & API Gateway

Third-Party Tools

- OWASP Vulnerability Dependency Check[33]

Reliability Pillar

The **reliability** pillar includes the ability of a system to recover from infrastructure or service disruptions, dynamically acquire computing resources to meet demand, and mitigate disruptions such as misconfigurations or transient network issues.

Definition

There are three best practice areas for reliability in the cloud:

- Foundations
- Change management
- Failure management

To achieve reliability, a system must have a well-planned foundation and monitoring in place, with mechanisms for handling changes in demand, requirements, or potentially defending an unauthorized denial of service attack. The system should be designed to detect failure and, ideally, automatically heal itself.

Best Practices

Foundations

REL 1: How are you regulating inbound request rates?

Throttling

In a microservices architecture, API consumers may be in separate teams or even outside the organization. This creates a vulnerability due to unknown access patterns as well as the risk of consumer credentials being compromised. The service API can potentially be affected if the number of requests exceeds what the processing logic/backend can handle.

Additionally, events that trigger new transactions, such as an update in a database row or new objects being added to an S3 bucket as part of the API, will trigger additional executions throughout a serverless application.

Throttling should be enabled at the API level to enforce access patterns established by a service contract. Defining a request access pattern strategy is fundamental to establishing how a consumer should use a service, whether that is at the resource or global level.

Returning the appropriate HTTP status codes within your API (such as a 429 for throttling) helps consumers plan for throttled access by implementing back-off and retries accordingly.

For more granular throttling and metering usage, issuing API keys to consumers with usage plans in addition to global throttling enables API Gateway to enforce quota and access patterns in unexpected behavior. API keys also simplify the process for administrators to cut off access if an individual consumer is making suspicious requests.

A common way to capture API keys is through a developer portal. This provides you, as the service provider, with additional metadata associated with the consumers and requests. You may capture the application, contact information, and business area/purpose and store this data in a durable data store, such as DynamoDB. This gives you additional validation of your consumers and provides traceability of logging with identities, so that you can contact consumers for breaking change upgrades/issues.

As discussed in the security pillar, API keys are not a security mechanism to authorize requests, and, therefore, should only be used with one of the available authorization options available within API Gateway.

Concurrency controls are sometimes necessary to protect specific workloads against service failure as they may not scale as rapidly as Lambda. Concurrency controls enable you to control the allocation of how many concurrent invocations of a particular Lambda function are set at the individual Lambda function level.

Lambda invocations that exceed the concurrency set of an individual function will be throttled by the AWS Lambda Service and the result will vary depending on their event source – Synchronous invocations return HTTP 429 error, Asynchronous invocations will be queued and retried while Stream-based event sources will retry up to their record expiration time.

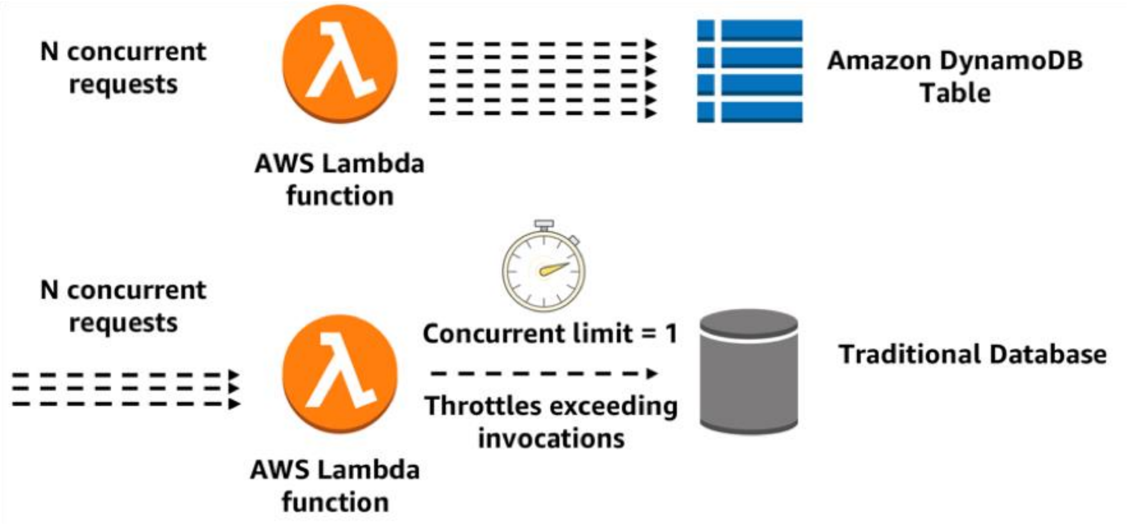


Figure 16: AWS Lambda concurrency controls

Controlling concurrency is particularly useful for the following scenarios:

- Sensitive backend or integrated systems that may have scaling limitations
- Database Connection Pool restrictions such as a relational database, which may impose concurrent limits
- Critical Path Services: Higher priority Lambda functions, such as authorization vs lower priority functions (for example, back-office) against limits in the same account
- Ability to disable Lambda function (concurrency = 0) in the event of anomalies.
- Limiting desired execution concurrency to protect against Distributed Denial of Service (DDoS) attacks

Concurrency controls for Lambda functions also limit its ability to scale beyond the concurrency set and draws from your account reserved concurrency pool. For asynchronous processing, use Kinesis Data Streams to effectively control concurrency with a single shard as opposed to Lambda function concurrency control. This gives you the flexibility to increase the number of shards or the parallelization factor to increase concurrency of your Lambda function.

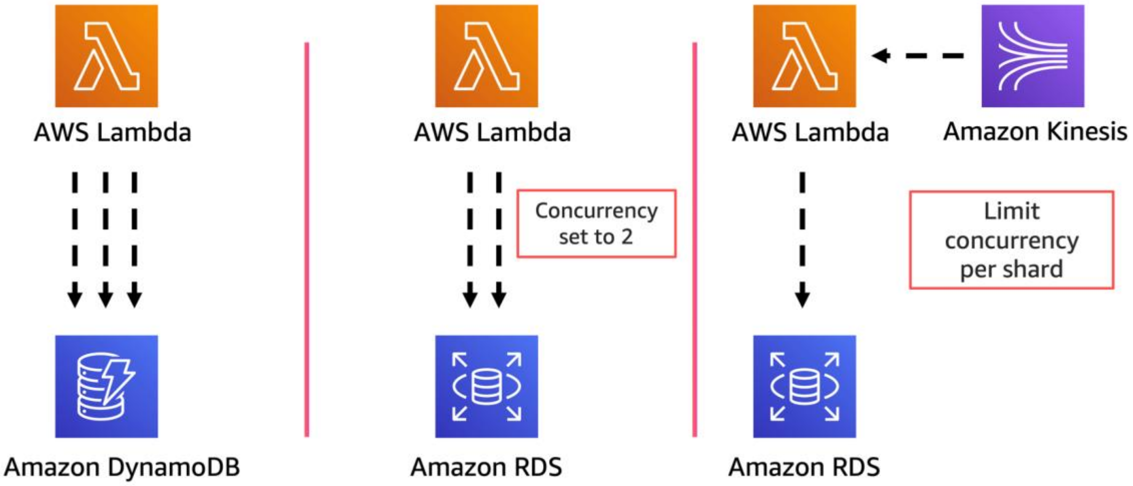


Figure 8: Concurrency controls for synchronous and asynchronous requests

REL 2: How are you building resiliency into your serverless application?

Asynchronous Calls and Events

Asynchronous calls reduce the latency on HTTP responses. Multiple synchronous calls, as well as long-running wait cycles, may result in timeouts and “locked” code that prevents retry logic.

Event-driven architectures enable streamlining asynchronous executions of code, thus limiting consumer wait cycles. These architectures are commonly implemented asynchronously using queues, streams, pub/sub, Webhooks, state machines, and event rule managers across multiple components that perform a business functionality.

User experience is decoupled with asynchronous calls. Instead of blocking the entire experience until the overall execution is completed, frontend systems receive a reference/job ID as part of their initial request and they subscribe for real-time changes, or in legacy systems use an additional API to poll its status. This decoupling allows the frontend to be more efficient by using event loops, parallel, or concurrency techniques while making such requests and lazily loading parts of the application when a response is partially or completely available.

The frontend becomes a key element in asynchronous calls as it becomes more robust with custom retries and caching. It can halt an in-flight request if no response has been received within an acceptable SLA, be it caused by an anomaly, transient condition, networking, or degraded environments.

Alternatively, when synchronous calls are necessary, it's recommended at a minimum to ensure that the total execution time doesn't exceed the API Gateway or AWS AppSync maximum timeout. Use an external service (for example, AWS Step Functions) to coordinate business transactions across multiple services, to control state and handle error handling that occurs along the request lifecycle.

Change Management

This is covered in the AWS Well-Architected Framework, and specific information on serverless can be found in the operational excellence pillar.

Failure Management

Certain parts of a serverless application are dictated by asynchronous calls to various components in an event-driven fashion, such as by pub/sub and other patterns. When asynchronous calls fail, they should be captured and retried whenever possible. Otherwise, data loss can occur, resulting in a degraded customer experience.

For Lambda functions, build retry logic into your Lambda queries to ensure that spiky workloads don't overwhelm your backend. Use structured logging as covered in the operational excellence pillar to log retries, including contextual information about errors as they can be captured as a custom metric. Use Lambda Destinations to send contextual information about errors, stack traces, and retries into dedicated Dead Letter Queues (DLQ), such as SNS topics and SQS queues. You also want to develop a plan to poll by a separate mechanism to re-drive these failed events back to their intended service.

AWS SDKs provide back-off and retry mechanisms by default when talking to other AWS services that are sufficient in most cases. However, review and tune them to suit your needs, especially HTTP keepalive, connection, and socket timeouts.

Whenever possible, use Step Functions to minimize the amount of custom try/catch, back-off, and retry logic within your serverless applications. For more information, see the cost optimization pillar section. Use Step Functions integration to save failed state executions and their state into a DLQ.

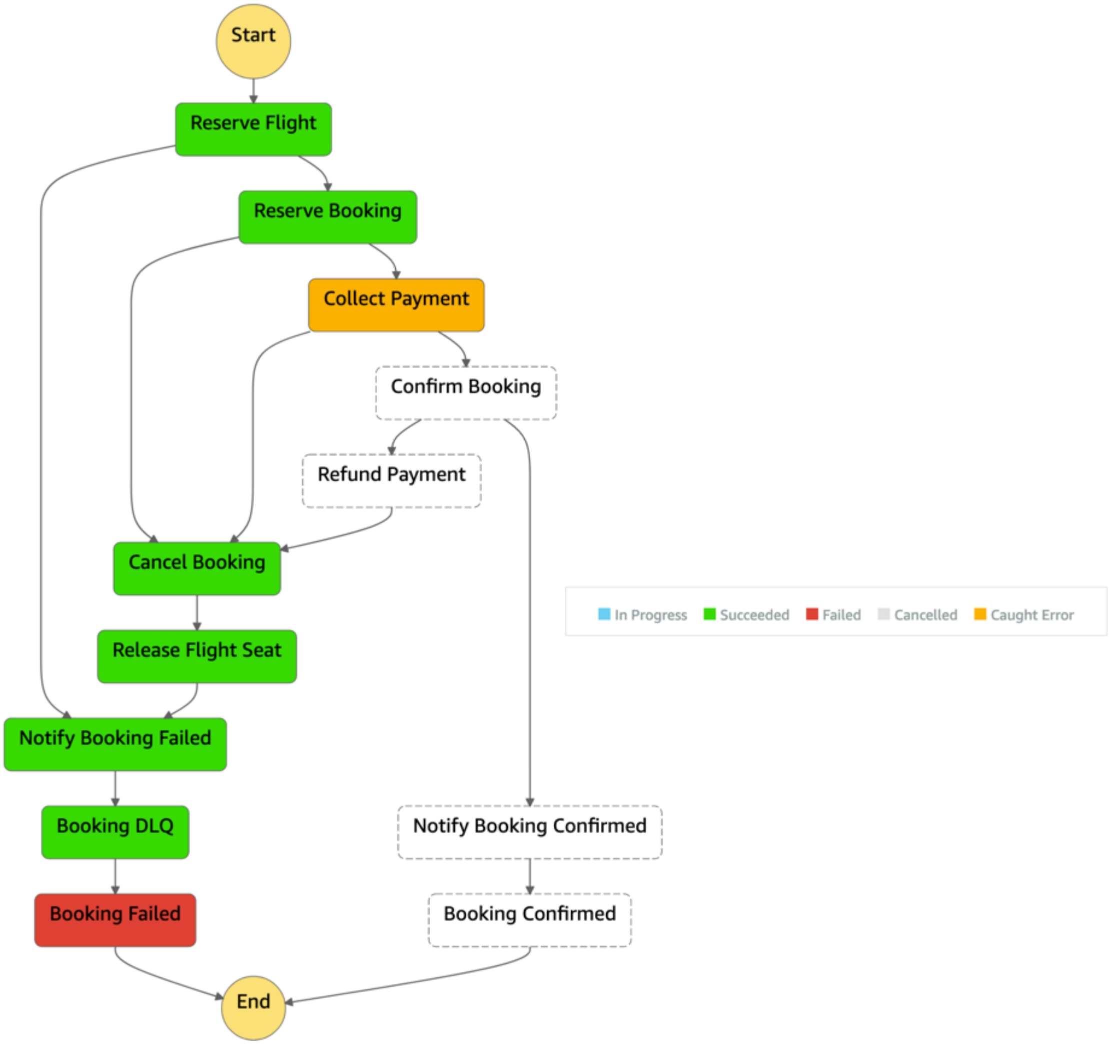


Figure 9: Step Functions state machine with DLQ step

Partial failures can occur in non-atomic operations, such as PutRecords (Kinesis) and BatchWriteItem (DynamoDB), since they return successful if at least one record has been ingested successfully. Always inspect the response when using such operations and programmatically deal with partial failures.

When consuming from Kinesis or DynamoDB streams, use Lambda error handling controls, such as maximum record age, maximum retry attempts, DLQ on failure, and Bisect batch on function error, to build additional resiliency into your application.

For synchronous parts that are transaction-based and depend on certain guarantees and requirements, rolling back failed transactions as described by the Saga pattern[34] also can be achieved by using Step Functions state machines, which will decouple and simplify the logic of your application.

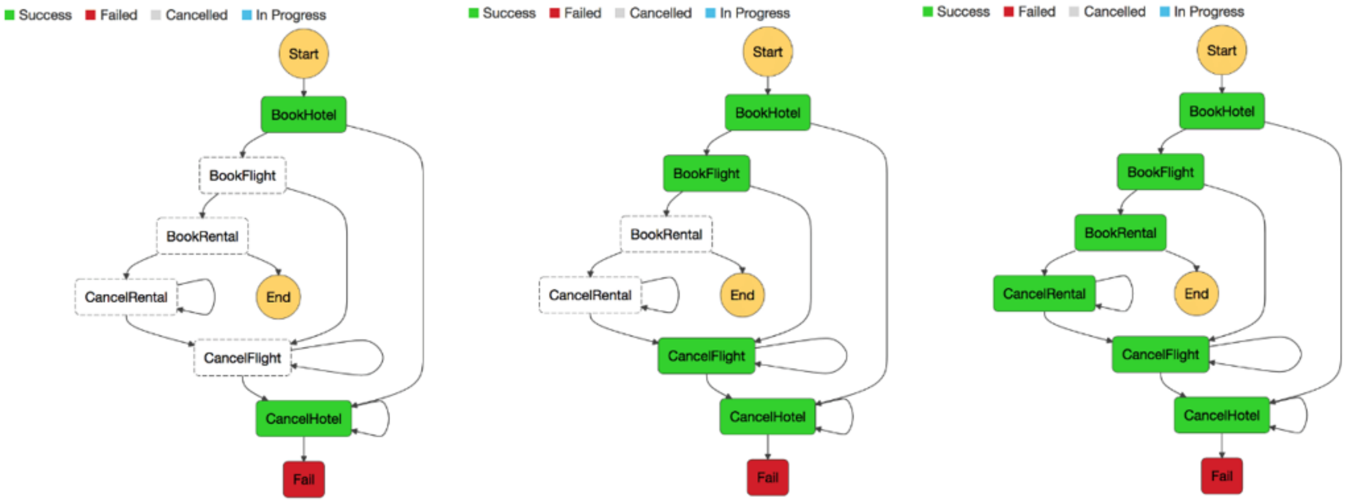


Figure 10: Saga pattern in Step Functions by Yan Cui

Limits

In addition to what is covered in the Well-Architected Framework, consider reviewing limits for burst and spiky use cases. For example, API Gateway and Lambda have different limits for steady and burst request rates. Use scaling layers and asynchronous patterns when possible, and perform load test to ensure that your current account limits can sustain your actual customer demand.

Key AWS Services

Key AWS services for reliability are AWS Marketplace, Trusted Advisor, CloudWatch Logs, CloudWatch, API Gateway, Lambda, X-Ray, Step Functions, Amazon SQS, and Amazon SNS.

Resources

Refer to the following resources to learn more about our best practices for reliability.

Documentation & Blogs

- Limits in Lambda[35]
- Limits in API Gateway[36]
- Limits in Kinesis Streams[37]

- Limits in DynamoDB[38]
- Limits in Step Functions[39]
- Error handling patterns[40]
- Serverless testing with Lambda[41]
- Monitoring Lambda Functions Logs[42]
- Versioning Lambda[43]
- Stages in API Gateway[44]
- API Retries in AWS[45]
- Step Functions error handling[46]
- X-Ray[47]
- Lambda DLQ[48]
- Error handling patterns with API Gateway and Lambda[49]
- Step Functions Wait state[50]
- Saga pattern[51]
- Applying Saga pattern via Step Functions[52]
- Serverless Application Repository App – DLQ Redriver
- Troubleshooting retry and timeout issues with AWS SDK
- Lambda resiliency controls for stream processing
- Lambda Destinations
- Serverless Application Repository App – Event Replay
- Serverless Application Repository App – Event Storage and Backup

Whitepapers

- Microservices on AWS[53]

Performance Efficiency Pillar

The **performance efficiency** pillar focuses on the efficient use of computing resources to meet requirements and the maintenance of that efficiency as demand changes and technologies evolve.

Definition

Performance efficiency in the cloud is composed of four areas:

- Selection
- Review
- Monitoring
- Tradeoffs

Take a data-driven approach to selecting a high-performance architecture. Gather data on all aspects of the architecture, from the high-level design to the selection and configuration of resource types. By reviewing your choices on a cyclical basis, you will ensure that you are taking advantage of the continually evolving AWS Cloud.

Monitoring will ensure that you are aware of any deviance from expected performance and can take action on it. Finally, you can make tradeoffs in your architecture to improve performance, such as using compression or caching, or by relaxing consistency requirements.

PER 1: How have you optimized the performance of your serverless application?

Selection

Run performance tests on your serverless application using steady and burst rates. Using the result, try tuning capacity units and load test after changes to help you select the best configuration:

- **Lambda:** Test different memory settings as CPU, network, and storage IOPS are allocated proportionally.
- **API Gateway:** Use Edge endpoints for geographically dispersed customers. Use Regional for regional customers and when using other AWS services within the same Region.

- **DynamoDB:** Use on-demand for unpredictable application traffic, otherwise provisioned mode for consistent traffic.
- **Kinesis:** Use enhanced-fan-out for dedicated input/output channel per consumer in multiple consumer scenarios. Use an extended batch window for low volume transactions with Lambda.

Configure VPC access to your Lambda functions only when necessary. Set up a NAT gateway if your VPC-enabled Lambda function needs access to the internet. As covered in the Well-Architected Framework, configure your NAT gateway across multiple Availability Zones for high availability and performance.

API Gateway Edge-optimized APIs provide a fully managed CloudFront distribution to optimize access for geographically dispersed consumers. API requests are routed to the nearest CloudFront Point of Presence (POP), which typically improves connection time.

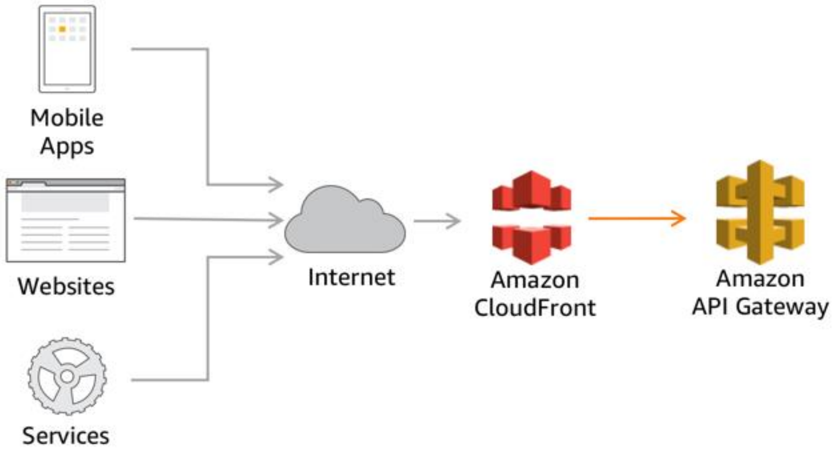


Figure 11: Edge-optimized API Gateway deployment

API Gateway Regional endpoint doesn’t provide a CloudFront distribution and enables HTTP2 by default, which helps reduce overall latency when requests originate from the same Region. Regional endpoints also allow you to associate your own Amazon CloudFront distribution or an existing CDN.

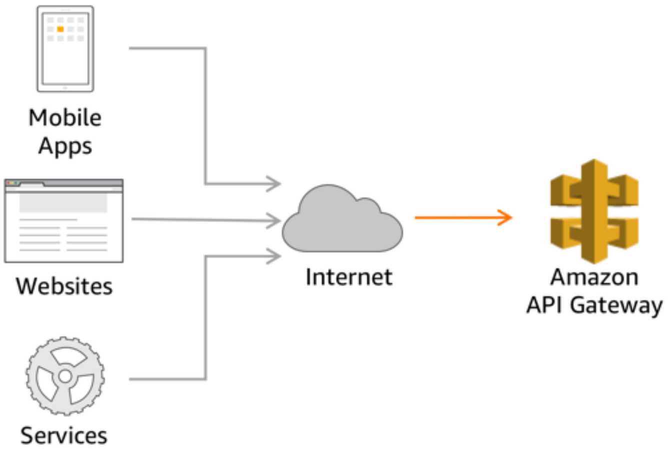


Figure 21: Regional Endpoint API Gateway deployment

This table can help you decide whether to deploy and Edge-optimized API or Regional API Endpoint:

	Edge-optimized API	Regional API Endpoint
API is accessed across Regions. Includes API Gateway-managed CloudFront distribution.	X	
API is accessed within same Region. Least request latency when API is accessed from same Region as API is deployed.		X
Ability to associate own CloudFront distribution.		X

This decision tree can help you decide when to deploy your Lambda function in a VPC.

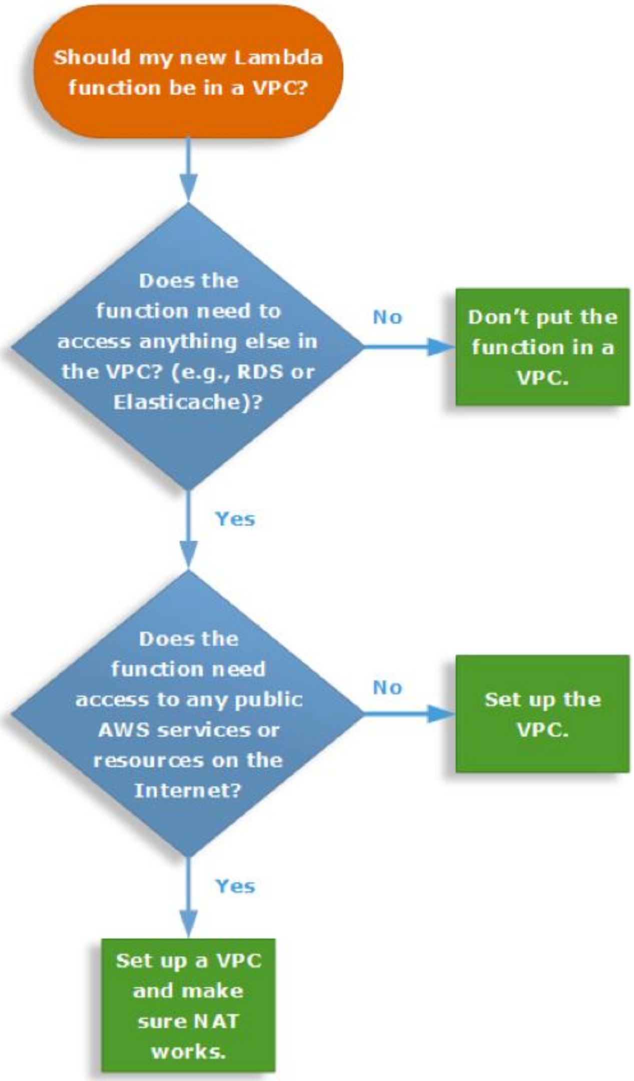

Should my new Lambda function be in a VPC?
Does the function need to access anything else in the VPC? (e.g., RDS or Elasticache)?
No
Don't put the function in a VPC.
Yes
Does the function need access to any public AWS services or resources on the Internet?
No
Set up the VPC.
Yes
Set up a VPC and make sure NAT works.

Figure 12: Decision tree for deploying a Lambda function in a VPC

Optimize

As a serverless architecture grows organically, there are certain mechanisms that are commonly used across a variety of workload profiles. Despite performance testing, design tradeoffs should be considered to increase your application's performance, always keeping your SLA and requirements in mind.

API Gateway and AWS AppSync caching can be enabled to improve performance for applicable operations. DAX can improve read responses significantly as well as Global and Local Secondary Indexes to prevent DynamoDB full table scan operations. These details and resources were described in the Mobile Backend scenario.

API Gateway content encoding allows API clients to request the payload to be compressed before being sent back in the response to an API request. This reduces the number of bytes that are sent from API Gateway to API clients and decreases the time it takes to transfer the data. You can enable content encoding in the API definition and you can also set the minimum response size that triggers compression. By default, APIs do not have content encoding support enabled.

Set your function timeout a few seconds higher than the average execution to account for any transient issues in downstream services used in the communication path. This also applies when working with Step Functions activities, tasks, and SQS message visibility.

Choosing a default memory setting and timeout in AWS Lambda may have an undesired effect in performance, cost, and operational procedures.

Setting the timeout much higher than the average execution may cause functions to execute for longer upon code malfunction, resulting in higher costs and possibly reaching concurrency limits depending on how such functions are invoked.

Setting a timeout that equals one successful function execution may trigger a serverless application to abruptly halt an execution should a transient networking issue or abnormality in downstream services occur.

Setting a timeout without performing load testing and, more importantly, without considering upstream services may result in errors whenever any part reaches its timeout first.

Follow best practices for working with Lambda functions[54] such as container reuse, minimizing deployment package size to its runtime necessities, and minimizing the complexity of your dependencies including frameworks that may not be optimized for fast startup. The latency 99th percentile (P99) should always be taken into account, as one may not impact application SLA agreed with other teams.

For Lambda functions in VPC, avoid DNS resolution of public host names of underlying resources in your VPC. For example, if your Lambda function accesses an Amazon RDS DB instance in your VPC, launch the instance with the no-publicly-accessible option.

After a Lambda function has executed, AWS Lambda maintains the execution context for some arbitrary time in anticipation of another Lambda function invocation. That allows you to use the global scope for one-off expensive operations, for example establishing a database connection or any initialization logic. In subsequent invocations, you can verify whether it’s still valid and reuse the existing connection.

Asynchronous Transactions

Because your customers expect more modern and interactive user interfaces, you can no longer sustain complex workflows using synchronous transactions. The more service interaction you need the more you end up chaining calls that may end up increasing the risk on service stability as well as response time.

Modern UI frameworks, such as Angular.js, VueJS, and React, asynchronous transactions, and cloud native workflows provide a sustainable approach to meet customers demand as well as helping you decouple components and focus on process and business domains instead.

These asynchronous transactions (or often times described as an event-driven architecture) kick off downstream subsequent choreographed events in the cloud instead of constraining clients to lock-and-wait (I/O blocking) for a response. Asynchronous workflows handle a variety of use cases including, but not limited to: data Ingestion, ETL operations, and order/request fulfillment.

In these use-cases, data is processed as it arrives and is retrieved as it changes. We outline best practices for two common asynchronous workflows where you can learn a few optimization patterns for integration and async processing.

Serverless Data Processing

In a serverless data processing workflow, data is ingested from clients into Kinesis (using the Kinesis agent, SDK, or API), and arrives in Amazon S3.

New objects kick off a Lambda function that is automatically executed. This function is commonly used to transform or partition data for further processing and possibly stored in other destinations such as DynamoDB, or another S3 bucket where data is in its final format.

As you may have different transformations for different data types, we recommend granularly splitting the transformations into different Lambda functions for optimal performance. With this approach, you have the flexibility to run data transformation in parallel and gain speed as well as cost.

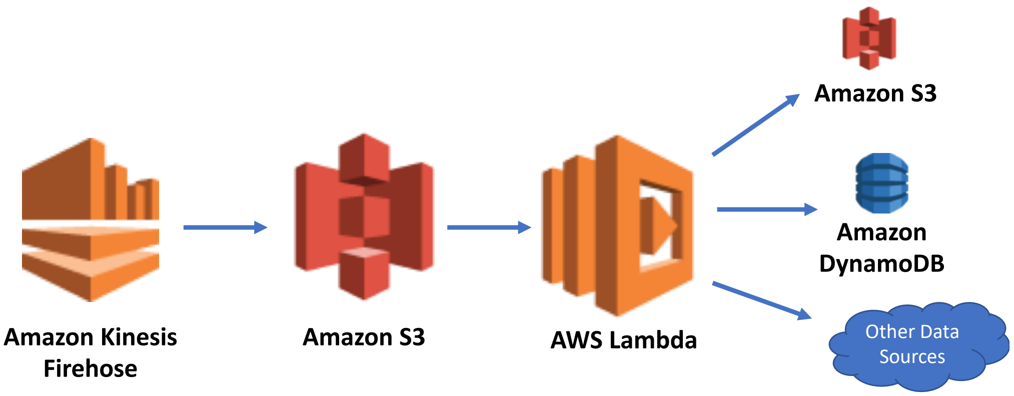


Figure 23: Asynchronous data ingestion

Kinesis Data Firehose offers native data transformations that can be used as an alternative to Lambda, where no additional logic is necessary for transforming records in Apache Log/System logs to CSV, JSON; JSON to Parquet or ORC.

Serverless Event Submission with Status Updates

Suppose you have an ecommerce site and a customer submits an order that kicks off an inventory deduction and shipment process; or an enterprise application that submits a large query that may take minutes to respond.

The processes required to complete this common transaction may require multiple service calls that may take a couple of minutes to complete. Within those calls, you want to safeguard against potential failures by adding retries and exponential backoffs, However, that can cause a suboptimal user experience for whoever is waiting for the transaction to complete.

For long and complex workflows similar to this, you can integrate API Gateway or AWS AppSync with Step Functions that upon new authorized requests will start this business workflow. Step Functions responds immediately with an execution ID to the caller (Mobile App, SDK, web service, etc.).

For legacy systems, you can use the execution ID to poll Step Functions for the business workflow status via another REST API. With WebSockets whether you're using REST or GraphQL, you can receive business workflow status in real-time by providing updates in every step of the workflow.

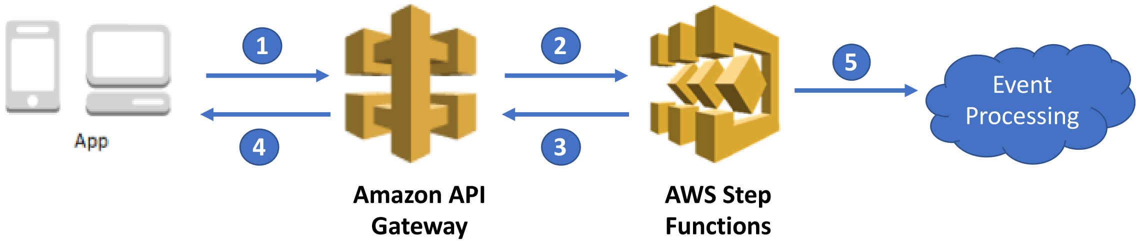


Figure 24: Asynchronous workflow with Step Functions state machines

Another common scenario is integrating API Gateway directly with SQS or Kinesis as a scaling layer. A Lambda function would only be necessary if additional business information or a custom request ID format is expected from the caller.

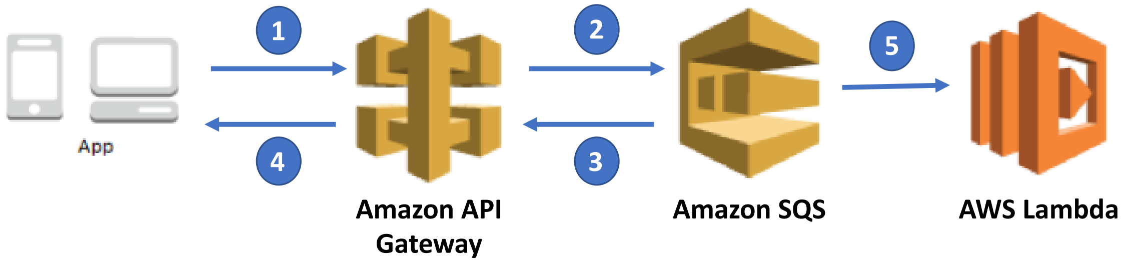


Figure 25: Asynchronous workflow using a queue as a scaling layer

In this second example, SQS serves multiple purposes:

1. Storing the request record durably is important because the client can confidently proceed throughout the workflow knowing that the request will eventually be processed
2. Upon a burst of events that may temporarily overwhelm the backend, the request can be polled for processing when resources become available.

Compared to the first example without a queue, Step Functions is storing the data durably without the need for a queue or state-tracking data sources. In both examples, the best practice is to pursue an asynchronous workflow after the client submits the request and avoiding the resulting response as blocking code if completion can take several minutes.

With WebSockets, AWS AppSync provides this capability out of the box via GraphQL subscriptions. With subscriptions, an authorized client could listen for data mutations they're interested in. This is ideal for data that is streaming or may yield more than a single response.

With AWS AppSync, as status updates change in DynamoDB, clients can automatically subscribe and receive updates as they occur and it's the perfect pattern for when data drives the user interface.

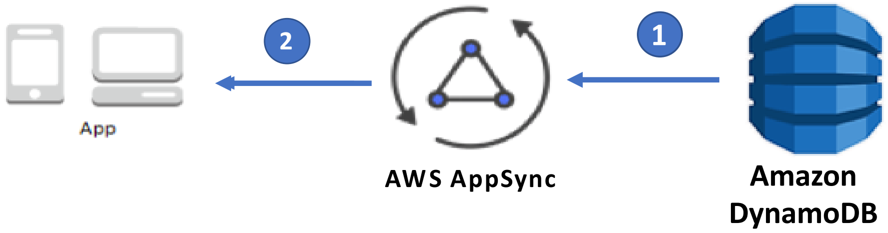


Figure 26: Asynchronous updates via WebSockets with AWS AppSync and GraphQL

Web Hooks can be implemented with SNS Topic HTTP subscriptions. Consumers can host an HTTP endpoint that SNS will call back via a POST method upon an event (for example, a data file arriving in Amazon S3). This pattern is ideal when the clients are configurable such as another microservice, which could host an endpoint. Alternatively, [Step Functions supports callbacks](#) where a state machine will block until it receives a response for a given task.

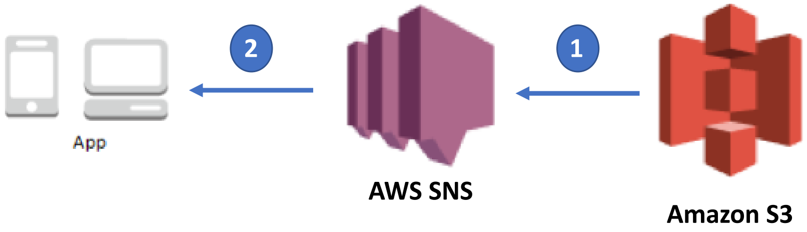


Figure 27: Asynchronous notification via Webhook with SNS

Lastly, polling could be costly from both a cost- and resource-perspective due to multiple clients constantly polling an API for status. If polling is the only option due to environment constraints, it's a best practice to establish SLAs with the clients to limit the number of "empty polls".

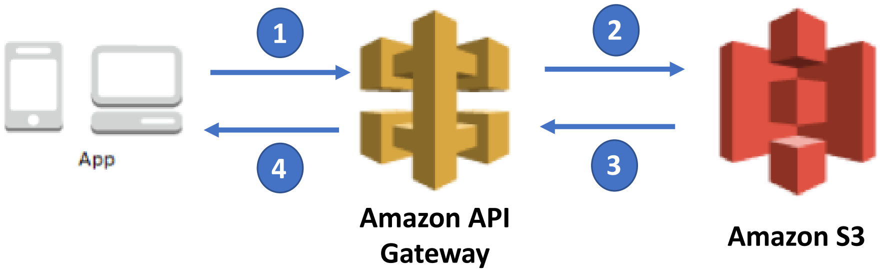


Figure 28: Client polling for updates on transaction recently made

For example, if a large data warehouse query takes an average of two minutes for a response, the client should poll the API after two minutes with exponential backoff if the data is not available. There are two common patterns to ensure that clients aren't polling more frequently than expected: Throttling and Timestamp for when is safe to poll again.

For timestamps, the system being polled can return an extra field with a timestamp or time period as to when it is safe for the consumer to poll once again. This approach follows an optimistic scenario where the consumer will respect and use this wisely and in the event of abuse you can also employ throttling for a more complete implementation.

Review

See the AWS Well-Architected Framework whitepaper for best practices in the **review** area for performance efficiency that apply to serverless applications.

Monitoring

See the AWS Well-Architected Framework whitepaper for best practices in the **monitoring** area for performance efficiency that apply to serverless applications.

Tradeoffs

See the AWS Well-Architected Framework whitepaper for best practices in the **tradeoffs** area for performance efficiency that apply to serverless applications.

Key AWS Services

Key AWS Services for performance efficiency are DynamoDB Accelerator, API Gateway, Step Functions, NAT gateway, Amazon VPC, and Lambda.

Resources

Refer to the following resources to learn more about our best practices for performance efficiency.

Documentation & Blogs

- AWS Lambda FAQs[55]
- Best Practices for Working with AWS Lambda Functions[56]
- AWS Lambda: How It Works[57]
- Understanding Container Reuse in AWS Lambda[58]
- Configuring a Lambda Function to Access Resources in an Amazon VPC[59]
- Enable API Caching to Enhance Responsiveness[60]
- DynamoDB: Global Secondary Indexes[61]
- Amazon DynamoDB Accelerator (DAX)[62]
- Developer Guide: Kinesis Streams[63]
- Java SDK: Performance improvement configuration
- Node.js SDK: Enabling HTTP Keep Alive
- Node.js SDK: Improving Imports
- Using Amazon SQS queues and AWS Lambda for high throughput
- Increasing stream processing performance with enhanced fan-out
- Lambda Power Tuning
- When to use Amazon DynamoDB on-demand and provisioned mode
- Analyzing Log Data with Amazon CloudWatch Logs Insights
- Integrating multiple data sources with AWS AppSync
- Step Functions Service Integrations
- Caching patterns
- Caching Serverless Applications
- Best Practices for Amazon Athena and AWS Glue

Cost Optimization Pillar

The **cost optimization** pillar includes the continual process of refinement and improvement of a system over its entire lifecycle. From the initial design of your first proof of concept to the ongoing operation of production workloads, adopting the practices in this document will enable you to build and operate cost-aware systems that achieve business outcomes and minimize costs, thus allowing your business to maximize its return on investment.

Definition

There are four best practice areas for cost optimization in the cloud:

- Cost-effective resources
- Matching supply and demand
- Expenditure awareness
- Optimizing over time

As with the other pillars, there are tradeoffs to consider. For example, do you want to optimize for speed to market or for cost? In some cases, it’s best to optimize for speed — going to market quickly, shipping new features, or simply meeting a deadline rather than investing in upfront cost optimization.

Design decisions are sometimes guided by haste as opposed to empirical data, as the temptation always exists to overcompensate “just in case” rather than spend time benchmarking for the most cost-optimal deployment.

This often leads to drastically over-provisioned and under-optimized deployments. The following sections provide techniques and strategic guidance for the initial and ongoing cost optimization of your deployment.

Generally, serverless architectures tend to reduce costs because some of the services, such as AWS Lambda, don’t cost anything while they’re idle. However, following certain best practices and making tradeoffs will help you reduce the cost of these solutions even more.

Best Practices

COST 1: How do you optimize your costs?

Cost-Effective Resources

Serverless architectures are easier to manage in terms of correct resource allocation. Due to its pay-per-value pricing model and scale based on demand, serverless effectively reduces the capacity planning effort.

As covered in the operational excellence and performance pillars, optimizing your serverless application has a direct impact on the value it produces and its cost.

As Lambda proportionally allocates CPU, network, and storage IOPS based on memory, the faster the execution the cheaper and more value your function produces due to 100-ms billing incremental dimension.

Matching Supply and Demand

The AWS serverless architecture is designed to scale based on demand and as such there are no applicable practices to be followed.

Expenditure Awareness

As covered in the AWS Well-Architected Framework, the increased flexibility and agility that the cloud enables encourages innovation and fast-paced development and deployment. It eliminates the manual processes and time associated with provisioning on-premises infrastructure, including identifying hardware specifications, negotiating price quotations, managing purchase orders, scheduling shipments, and then deploying the resources.

As your serverless architecture grows, the number of Lambda functions, APIs, stages, and other assets will multiply. Most of these architectures need to be budgeted and forecasted in terms of costs and resource management–tagging can help you here. You can allocate costs from your AWS bill to individual functions and APIs and obtain a granulated view of your costs per project in AWS Cost Explorer.

A good implementation is to share the same key-value tag for assets that belong to the project programmatically and create custom reports based on the tags that you have created. This feature will help you not only allocate your costs, but also identify which resources belong to which projects.

Optimizing Over Time

See the AWS Well-Architected Framework whitepaper for best practices in the **Optimizing Over Time** area for cost optimization that apply to serverless applications.

Logging Ingestion and Storage

AWS Lambda uses CloudWatch Logs to store the output of the executions to identify and troubleshoot problems on executions as well as monitoring the serverless application. These will impact the cost in the CloudWatch Logs service in two dimensions: ingestion and storage.

Set appropriate logging levels and remove unnecessary logging information to optimize log ingestion. Use environment variables to control application logging level and sample logging in DEBUG mode to ensure you have additional insight when necessary.

Set log retention periods for new and existing CloudWatch Logs groups. For log archival, export and set cost-effective storage classes that best suit your needs.

Direct Integrations

If your Lambda function is not performing custom logic while integrating with other AWS services, chances are that it may be unnecessary.

API Gateway, AWS AppSync, Step Functions, EventBridge, and Lambda Destinations can directly integrate with a number of services and provide you more value and less operational overhead.

Most public serverless applications provide an API with an agnostic implementation of the contract provided, as described in the Microservices scenario.

An example scenario where a direct integration is a better fit is ingesting click stream data through a REST API.

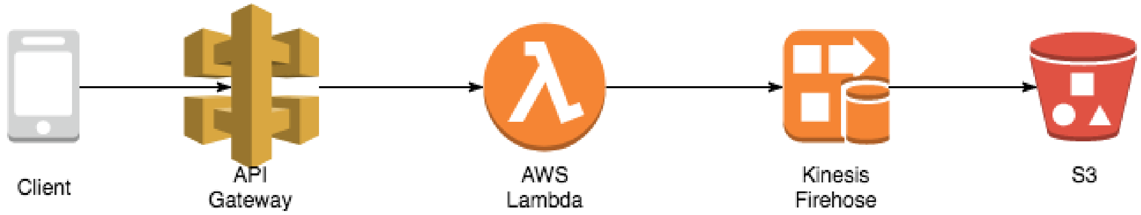


Figure 13: Sending data to Amazon S3 using Kinesis Data Firehose

In this scenario, API Gateway will execute a Lambda function that will simply ingest the incoming record into Kinesis Data Firehose that subsequently batches records before storing into a S3 bucket. As no additional logic is necessary for this example, we can use an API Gateway service proxy to directly integrate with Kinesis Data Firehose.

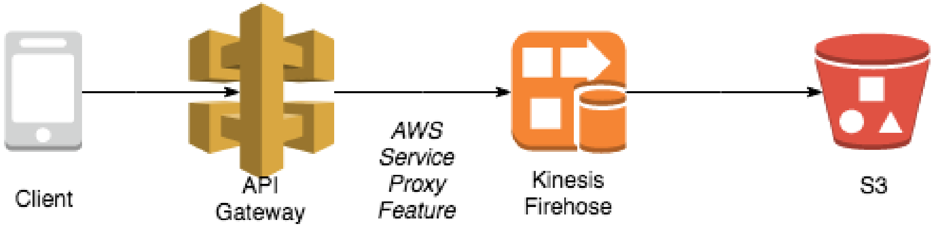


Figure 14: Reducing cost of sending data to Amazon S3 by implementing AWS service proxy

With this approach, we remove the cost of using Lambda and unnecessary invocations by implementing the AWS Service Proxy within API Gateway. As a tradeoff, this might introduce some extra complexity if multiple shards are necessary to meet the ingestion rate.

If latency sensitive, you can stream data directly to your Kinesis Data Firehose by having the correct credentials at the expense of abstraction, contract, and API features.

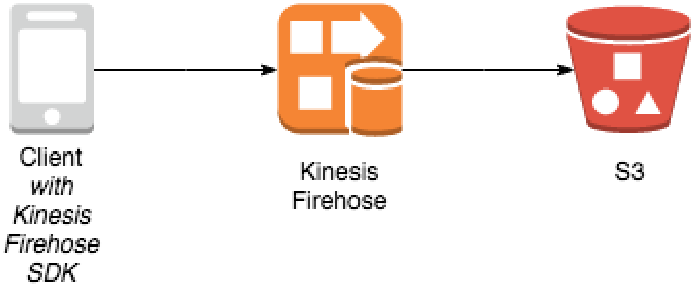


Figure 15: Reducing cost of sending data to Amazon S3 by streaming directly using the Kinesis Data Firehose SDK

For scenarios where you need to connect with internal resources within your VPC or on-premises and no custom logic is required, use API Gateway private integration.

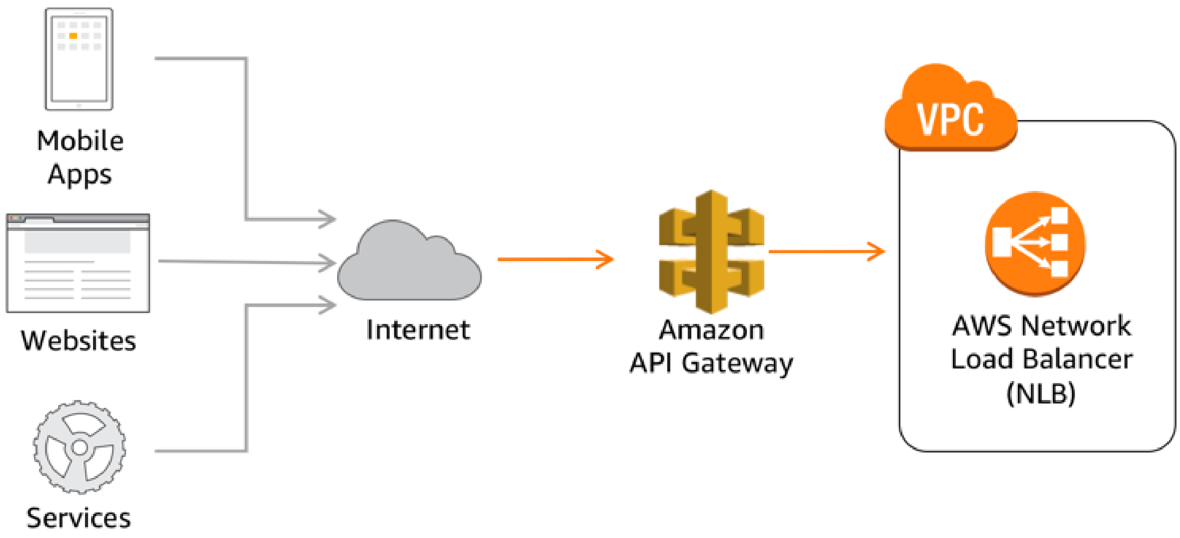


Figure 32: Amazon API Gateway private integration over Lambda in VPC to access private resources

With this approach, API Gateway sends each incoming request to an Internal Network Load Balancer that you own in your VPC which can forward the traffic to any backend, either in the same VPC or on-premises via IP address.

This approach has both cost and performance benefits as you don't need an additional hop to send requests to a private backend with the added benefits of authorization, throttling, and caching mechanisms.

Another scenario is a fan-out pattern where Amazon SNS broadcasts messages to all of its subscribers. This approach requires additional application logic to filter and avoid an unnecessary Lambda invocation.

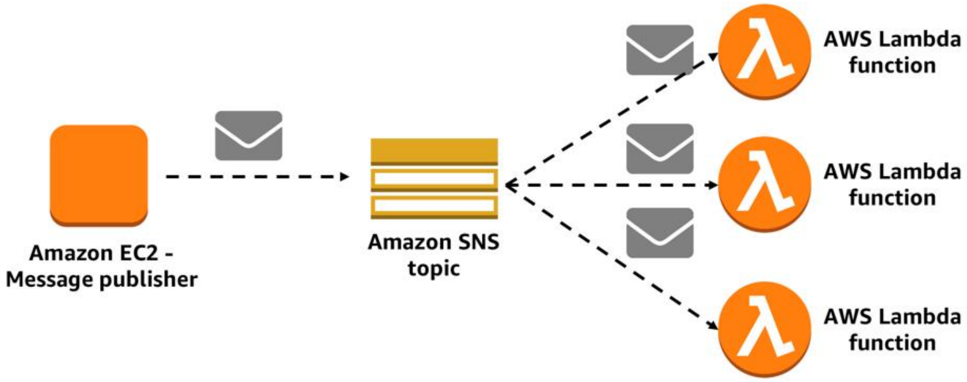


Figure 33: Amazon SNS without message attribute filtering

SNS can filter events based on message attributes and more efficiently deliver the message to the correct subscriber.

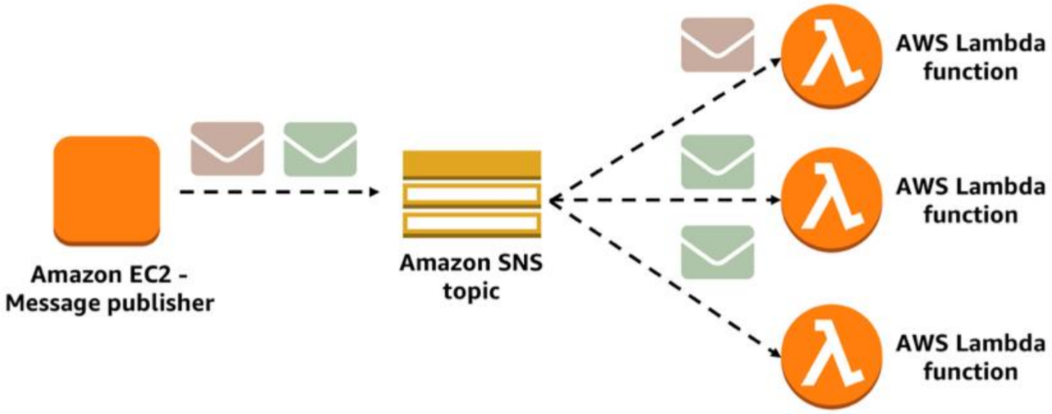


Figure 34: Amazon SNS with message attribute filtering

Another example is long running processing tasks where you may need to wait for task completion before proceeding to the next step. This wait state may be implemented within the Lambda code, however, it’s far more efficient to either transform to asynchronous processing using events, or implement the waiting state using Step Functions.

For example, in the following image, we poll an AWS Batch job and review its state every 30 seconds to see if it has finished. Instead of coding this wait within the Lambda function, we implement a poll (*GetJobStatus*) + wait (*Wait30Seconds*) + decider (*CheckJobStatus*).

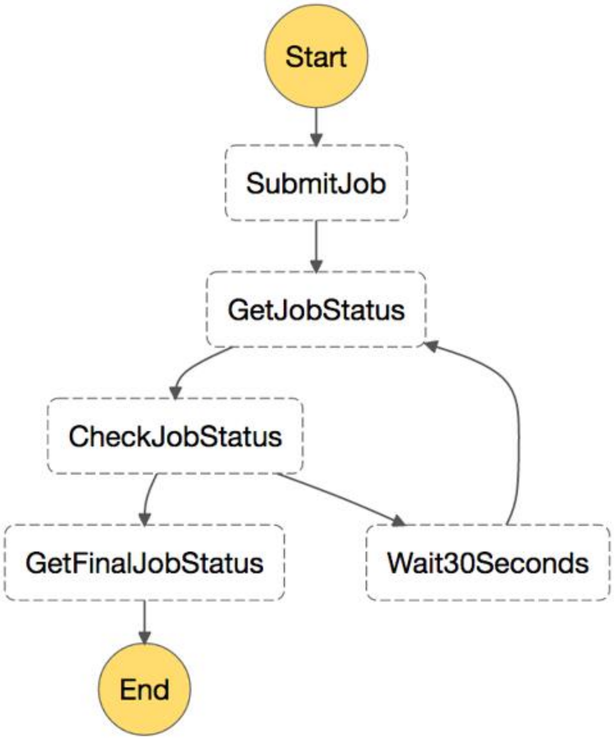


Figure 16: Implementing a wait state with AWS Step Functions

Implementing a wait state with Step Functions won’t incur any further cost as the pricing model for Step Functions is based on transitions between states and not on the time spent within a state.

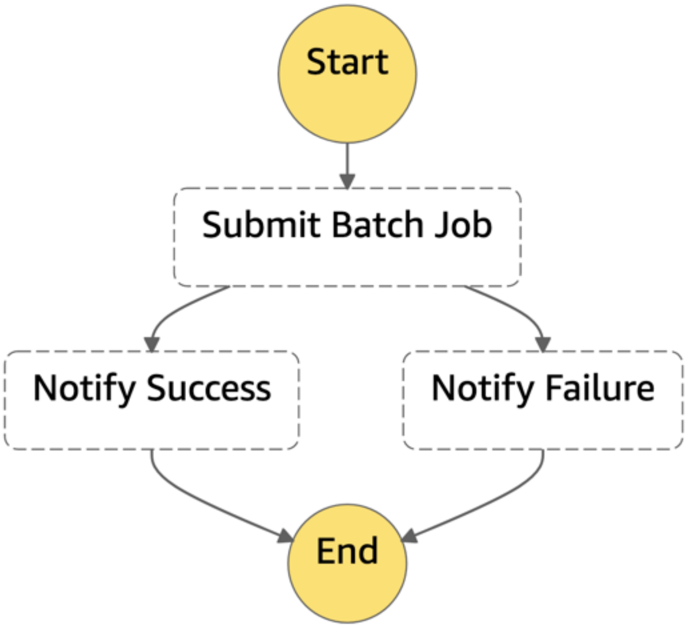


Figure 17: Step Functions service integration synchronous wait

Depending on the integration you have to wait, Step Functions can wait synchronously before moving to the next task saving you an additional transition.

Code optimization

As covered in the performance pillar, optimizing your serverless application can effectively improve the value it produces per execution.

The use of global variables to maintain connections to your data stores or other services and resources will increase performance and reduce execution time, which also reduces the cost. For more information, see the performance pillar section.

An example where the use of managed service features can improve the value per execution is retrieving and filtering objects from Amazon S3, since fetching large objects from Amazon S3 requires higher memory for Lambda functions.

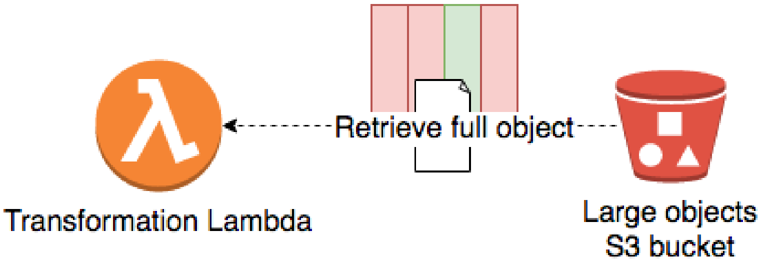


Figure 37: Lambda function retrieving full S3 object

In the previous diagram, we can see that, when retrieving large objects from Amazon S3, we might increase the memory consumption of the Lambda, increase the execution time (so the function can transform, iterate, or collect required data) and, in some cases, only part of this information is needed.

This is represented with three columns in red (data not required) and one column in green (data required). Using Athena SQL queries to gather granular information needed for your execution reduces the retrieval time and object size on which perform transformations.

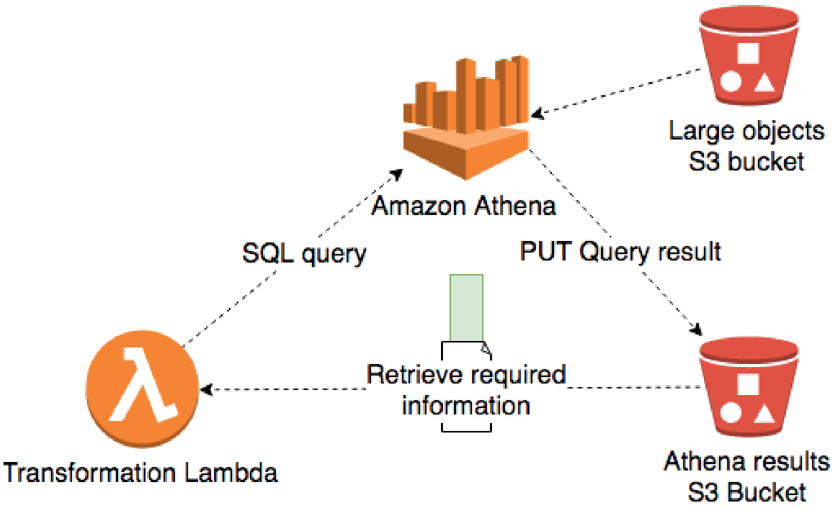
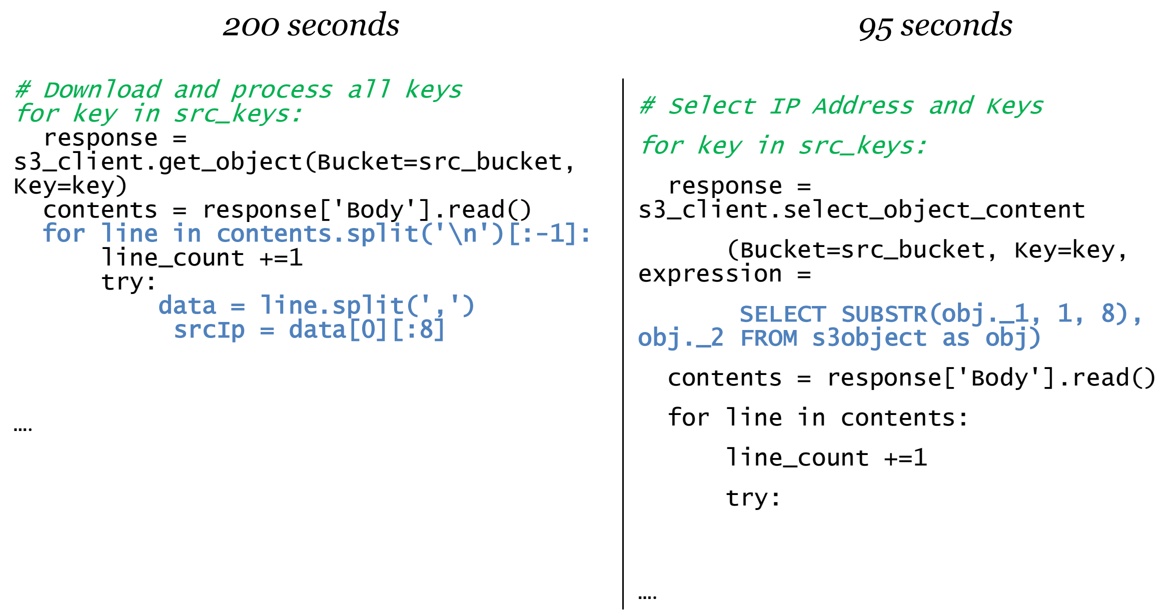


Figure 38: Lambda with Athena object retrieval

In the next diagram, we can see that, by querying Athena to get the specific data, we reduce the size of the object retrieved and, as an extra benefit, we can reuse that content since Athena saves its query results in a S3 bucket and invoke the Lambda invocation as the results land in Amazon S3 asynchronously.

A similar approach could be using with S3 Select. S3 Select enables applications to retrieve only a subset of data from an object by using simple SQL expressions. As in the previous example with Athena, retrieving a smaller object from Amazon S3 reduces execution time and the memory used by the Lambda function.



200 seconds

```
# Download and process all keys
for key in src_keys:
  response =
s3_client.get_object(Bucket=src_bucket,
Key=key)
  contents = response['Body'].read()
  for line in contents.split('\n')[:-1]:
      line_count +=1
      try:
          data = line.split(',')
           srcIp = data[0][:8]
```

….

95 seconds

```
# Select IP Address and Keys
for key in src_keys:
  response =
s3_client.select_object_content
      (Bucket=src_bucket, Key=key,
expression =
         SELECT SUBSTR(obj._1, 1, 8),
obj._2 FROM s3object as obj)
  contents = response['Body'].read()
  for line in contents:
      line_count +=1
      try:
```

….

Figure 18: Lambda performance statistics using Amazon S3 vs S3 Select

Resources

Refer to the following resources to learn more about our best practices for cost optimization.

Documentation & Blogs

- CloudWatch Logs Retention[64]
- Exporting CloudWatch Logs to Amazon S3[65]
- Streaming CloudWatch Logs to Amazon ES[66]
- Defining wait states in Step Functions state machines[67]
- Coca-Cola Vending Pass State Machine Powered by Step Functions[68]
- Building high throughput genomics batch workflows on AWS[69]
- Simplify your Pub/Sub Messaging with Amazon SNS Message Filtering
- S3 Select and Glacier Select

- Lambda Reference Architecture for MapReduce
- Serverless Application Repository App – Auto-set CloudWatch Logs group retention
- Ten resources every Serverless Architect should know

Whitepaper

- Optimizing Enterprise Economics with Serverless Architectures[70]

Conclusion

While serverless applications take the undifferentiated heavy-lifting off developers, there are still important principles to apply.

For reliability, by regularly testing failure pathways you will be more likely to catch errors before they reach production. For performance, starting backward from customer expectation will allow you to design for optimal experience. There are a number of AWS tools to help optimize performance as well.

For cost optimization, you can reduce unnecessary waste within your serverless application by sizing resources in accordance with traffic demand, and improve value by optimizing your application. For operations, your architecture should strive toward automation in responding to events.

Finally, a secure application will protect your organization’s sensitive information assets and meet any compliance requirements at every layer.

The landscape of serverless applications is continuing to evolve with the ecosystem of tooling and processes growing and maturing. As this occurs, we will continue to update this paper to help you ensure that your serverless applications are well-architected.

Contributors

The following individuals and organizations contributed to this document:

- Adam Westrich: Sr Solutions Architect, Amazon Web Services
- Mark Bunch: Enterprise Solutions Architect, Amazon Web Services
- Ignacio Garcia Alonso: Solutions Architect, Amazon Web Services

- Heitor Lessa: Principal Serverless Lead Well-Architected, Amazon Web Services
- Philip Fitzsimons: Sr. Manager Well-Architected, Amazon Web Services
- Dave Walker: Principal Specialist Solutions Architect, Amazon Web Services
- Richard Threlkeld: Sr. Product Manager Mobile, Amazon Web Services
- Julian Hambleton-Jones: Sr. Solutions Architect, Amazon Web Services

Further Reading

For additional information, see the following:

- AWS Well-Architected Framework[71]

Document Revisions

Date	Description
December 2019	Updates throughout for new features and evolution of best practice.
November 2018	New scenarios for Alexa and Mobile, and updates throughout to reflect new features and evolution of best practice.
November 2017	Initial publication.

Notes

[1] https://aws.amazon.com/well-architected

[2] http://d0.awsstatic.com/whitepapers/architecture/AWS_Well-Architected_Framework.pdf

[3] https://github.com/alexcasalboni/aws-lambda-power-tuning

[4] http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/BestPractices.html

[5] http://docs.aws.amazon.com/elasticsearch-service/latest/developerguide/es-managedomains.html

[6] https://www.elastic.co/guide/en/elasticsearch/guide/current/scale.html

[7] http://docs.aws.amazon.com/streams/latest/dev/kinesis-record-processor-scaling.html

[8] https://d0.awsstatic.com/whitepapers/whitepaper-streaming-data-solutions-on-aws-with-amazon-kinesis.pdf

[9] http://docs.aws.amazon.com/kinesis/latest/APIReference/API_PutRecords.html

[10] http://docs.aws.amazon.com/streams/latest/dev/kinesis-record-processor-duplicates.html

[11] http://docs.aws.amazon.com/lambda/latest/dg/best-practices.html#stream-events

[12] http://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-api-usage-plans.html

[13] http://docs.aws.amazon.com/apigateway/latest/developerguide/stage-variables.html

[14] http://docs.aws.amazon.com/lambda/latest/dg/env_variables.html

[15] https://github.com/awslabs/serverless-application-model

[16] https://aws.amazon.com/blogs/aws/latency-distribution-graph-in-aws-x-ray/

[17] http://docs.aws.amazon.com/lambda/latest/dg/lambda-x-ray.html

[18] http://docs.aws.amazon.com/systems-manager/latest/userguide/systems-manager-paramstore.html

[19] https://aws.amazon.com/blogs/compute/continuous-deployment-for-serverless-applications/

[20] https://github.com/awslabs/aws-serverless-samfarm

[21] https://d0.awsstatic.com/whitepapers/DevOps/practicing-continuous-integration-continuous-delivery-on-AWS.pdf

[22] https://aws.amazon.com/serverless/developer-tools/

[23] http://docs.aws.amazon.com/lambda/latest/dg/with-s3-example-create-iam-role.html

[24] http://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-method-request-validation.html

[25] http://docs.aws.amazon.com/apigateway/latest/developerguide/use-custom-authorizer.html

[26] https://aws.amazon.com/blogs/compute/secure-api-access-with-amazon-cognito-federated-identities-amazon-cognito-user-pools-and-amazon-api-gateway/

[27] http://docs.aws.amazon.com/lambda/latest/dg/vpc.html

[28] https://aws.amazon.com/pt/articles/using-squid-proxy-instances-for-web-service-access-in-amazon-vpc-another-example-with-aws-codedeploy-and-amazon-cloudwatch/

[29] https://www.owasp.org/images/0/08/OWASP_SCP_Quick_Reference_Guide_v2.pdf

[30] https://d0.awsstatic.com/whitepapers/Security/AWS_Security_Best_Practices.pdf

[31] https://www.twistlock.com/products/serverless-security/

[32] https://snyk.io/

[33] https://www.owasp.org/index.php/OWASP_Dependency_Check

[34] http://theburningmonk.com/2017/07/applying-the-saga-pattern-with-aws-lambda-and-step-functions/

[35] http://docs.aws.amazon.com/lambda/latest/dg/limits.html

[36] http://docs.aws.amazon.com/apigateway/latest/developerguide/limits.html#api-gateway-limits

[37] http://docs.aws.amazon.com/streams/latest/dev/service-sizes-and-limits.html

[38] http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Limits.html

[39] http://docs.aws.amazon.com/step-functions/latest/dg/limits.html

[40] https://aws.amazon.com/blogs/compute/error-handling-patterns-in-amazon-api-gateway-and-aws-lambda/

[41] https://aws.amazon.com/blogs/compute/serverless-testing-with-aws-lambda/

[42] http://docs.aws.amazon.com/lambda/latest/dg/monitoring-functions-logs.html

[43] http://docs.aws.amazon.com/lambda/latest/dg/versioning-aliases.html

[44] http://docs.aws.amazon.com/apigateway/latest/developerguide/stages.html

[45] http://docs.aws.amazon.com/general/latest/gr/api-retries.html

[46] http://docs.aws.amazon.com/step-functions/latest/dg/tutorial-handling-error-conditions.html#using-state-machine-error-conditions-step-4

[47] http://docs.aws.amazon.com/xray/latest/devguide/xray-services-lambda.html

[48] http://docs.aws.amazon.com/lambda/latest/dg/dlq.html

[49] https://aws.amazon.com/blogs/compute/error-handling-patterns-in-amazon-api-gateway-and-aws-lambda/

[50] http://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-wait-state.html

[51] http://microservices.io/patterns/data/saga.html

[52] http://theburningmonk.com/2017/07/applying-the-saga-pattern-with-aws-lambda-and-step-functions/

[53] https://d0.awsstatic.com/whitepapers/microservices-on-aws.pdf

[54] http://docs.aws.amazon.com/lambda/latest/dg/best-practices.html

[55] https://aws.amazon.com/lambda/faqs/

[56] http://docs.aws.amazon.com/lambda/latest/dg/best-practices.html

[57] http://docs.aws.amazon.com/lambda/latest/dg/lambda-introduction.html

[58] https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/

[59] http://docs.aws.amazon.com/lambda/latest/dg/vpc.html

[60] http://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-caching.html

[61] http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GSI.html

[62] https://aws.amazon.com/dynamodb/dax/

[63] http://docs.aws.amazon.com/streams/latest/dev/amazon-kinesis-streams.html

[64] http://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/SettingLogRetention.html

[65] http://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/S3ExportTasksConsole.html

[66] http://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/CWL_ES_Stream.html

[67] http://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-wait-state.html

[68] https://aws.amazon.com/blogs/aws/things-go-better-with-step-functions/

[69] https://aws.amazon.com/blogs/compute/building-high-throughput-genomics-batch-workflows-on-aws-workflow-layer-part-4-of-4/

[70] https://d0.awsstatic.com/whitepapers/optimizing-enterprise-economics-serverless-architectures.pdf

[71] https://aws.amazon.com/well-architected