AWS Well-Architected Framework

Serverless Applications Lens

Serverless Applications Lens: AWS Well-Architected Framework

Table of Contents

Serverless Applications Lens - AWS Well-Architected Framework

Publication date: **July 14, 2022** (*Document revisions*)

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. 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.

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.

Custom lens availability

Custom lenses extend the best practice guidance provided by AWS Well-Architected Tool. AWS WA Tool allows you to create your own custom lenses, or to use lenses created by others that have been shared with you.

To determine if a custom lens is available for the lens described in this whitepaper, reach out to your Technical Account Manager (TAM), Solutions Architect (SA), or Support.

Definitions

The AWS Well-Architected Framework is based on six pillars: operational excellence, security, reliability, performance efficiency, cost optimization, and sustainability. 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 eight areas that you should consider when building a serverless workload:

Topics

Compute layer

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

AWS Lambda lets you run stateless serverless applications on a managed platform that supports microservice 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 apply 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 OpenSearch Service (OpenSearch Service) makes it easy to deploy, secure, operate, and scale OpenSearch for log analytics, full-text search, application monitoring, and more. OpenSearch Service 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, OpenSearch Service, 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, you can run standard SQL, or build entire streaming applications using SQL.

Amazon Data Firehose captures, transforms, and loads streaming data into Managed Service for Apache Flink, Amazon S3, Amazon Redshift, and OpenSearch Service, 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 let 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 helps 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 Serverless Application Model 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-averse 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 (or Linear)	1–10% typical initial traffic shift, then	Revert 100% of traffic to previous deployment	Better for high concurrency workloads	Minutes to hours

Deployment	Consumer Impact	Rollback	Event Model Factors	Deployment Speed
	phased increases , or all at once			

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. Use this deployment model for non-critical environments, such as development, where impact to customers is not a risk.

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 a way for you to gradually release new software in a coordinated and safe way that enable rapid deployment cycles. Canary deployments involve deploying a percentage of requests to new code, and monitoring for errors, degradations, or regressions.

You can use Lambda function aliases with AWS CodeDeploy to support various canary deployment strategies. AWS SAM comes with built-in support for CodeDeploy, which makes Canary deployments even simpler. Operators can further control gradual deployments by leveraging pre-traffic and post-traffic deployment hooks and CloudWatch alarms to trigger automated rollback.

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. AWS 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 AWS CloudTrail. As a best practice in production, customers should enable versioning to use a reliable architecture.

To simplify deployment operations and reduce the risk of error, Lambda function aliases activate 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 activates a safe deployment when promoting a new version to the live environment because the Lambda alias within the caller configuration remains static, thus there are fewer changes to make.

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 initiations 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. Use code or dependencies that are hardware-agnostic. 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 achieve lean service design.
- **Design for failures and duplicates**: Operations triggered from requests or events must be idempotent, as failures can occur and a given request or event can be delivered more than once. Include appropriate retries for downstream calls.

Scenarios

In this section, we cover the six 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.

Key scenarios

- RESTful microservices
- Alexa skills
- Mobile backend
- Streaming processing
- Web application
- Event-driven architectures

RESTful microservices

When building a microservice, think about how a business context can be delivered as a reusable 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 use 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 and response mapping, and performance optimizations.
3. **AWS Lambda** contains the business logic to process incoming API calls and use 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

- Use 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 OpenSearch Service 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 or 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 behave as expected. It's essential to start by designing a voice interaction model and working backwards since some users may say too little, too much, or possibly something unexpected. The voice design process involves creating, scripting, and planning for expected as well as unexpected utterances.

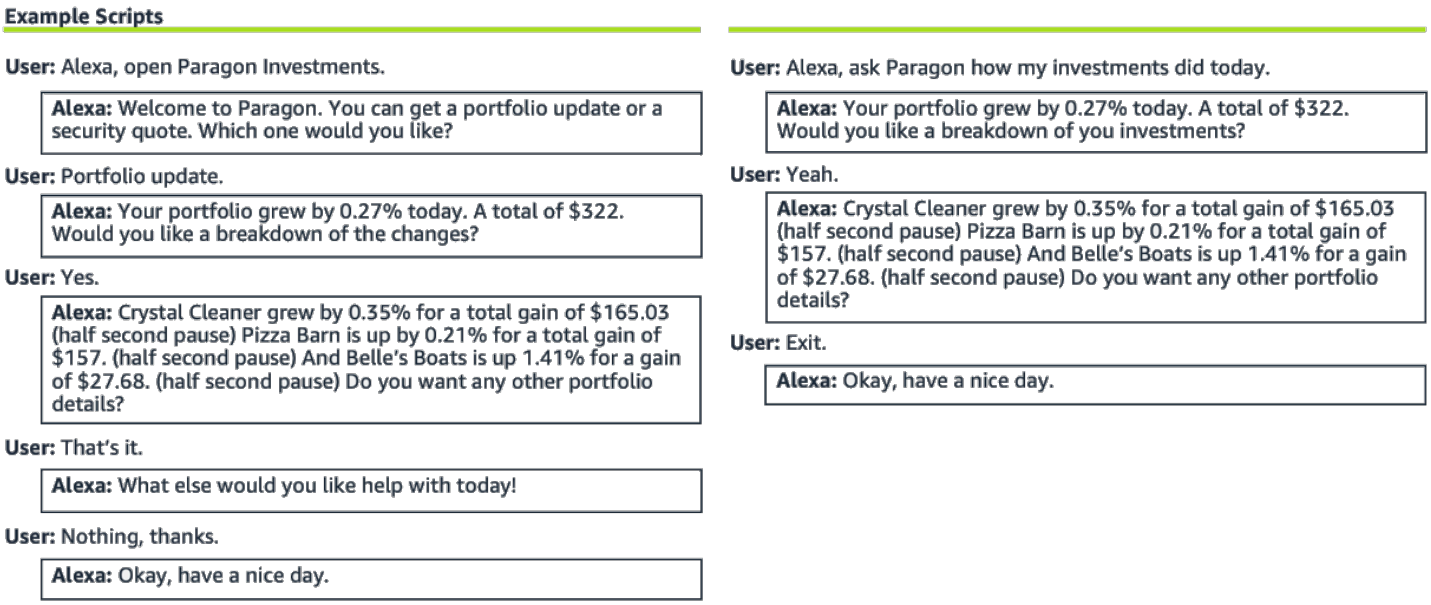


Figure 2: Alexa Skill example design script

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

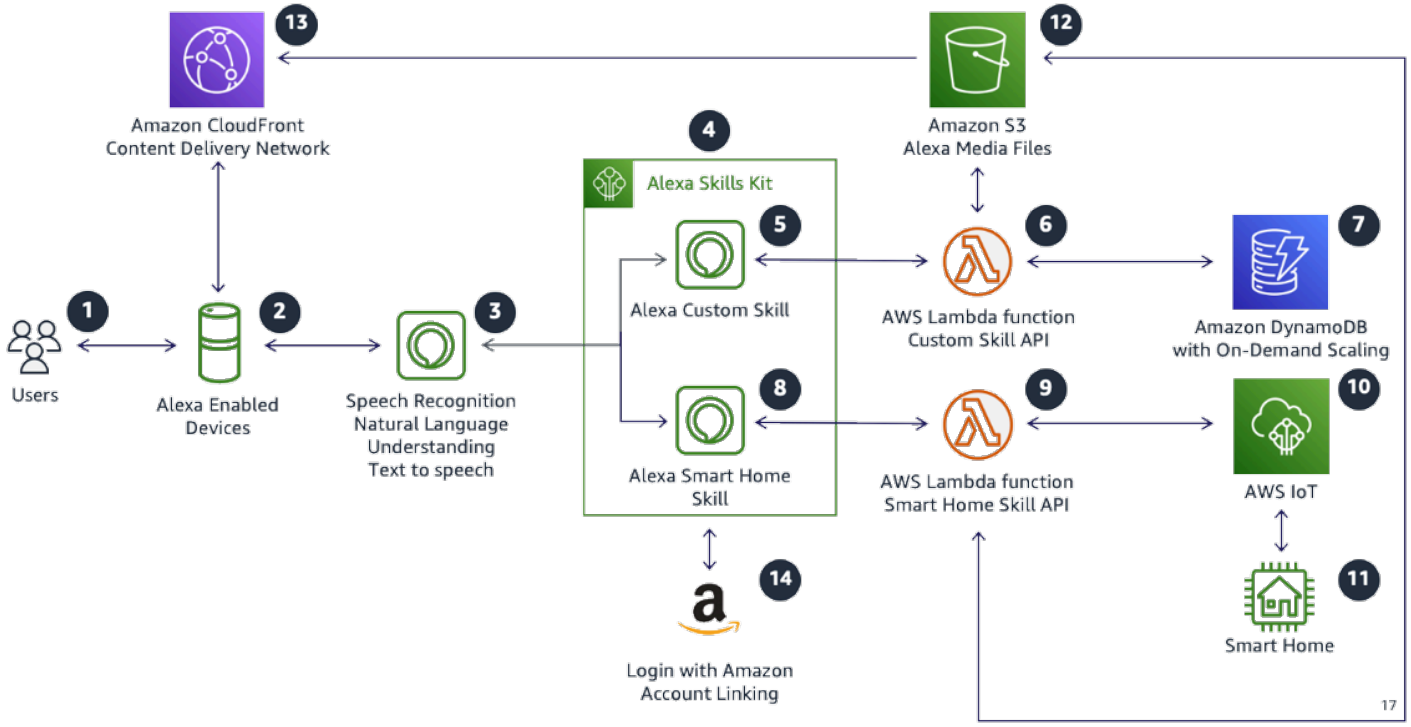


Figure 3: Reference architecture for an Alexa Skill

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. Using DynamoDB's on-demand scaling mechanism offers simple pay-per-request pricing for read and write requests so that you only pay for what you use, and you do not need to worry about forecasting read and write throughput. DynamoDB is commonly used by Alexa skills to persist user state and sessions.
8. **Alexa Smart Home Skill** allows you to control devices such as lights, thermostats, and smart TVs using the Smart Home API. Smart Home skills are simpler to build than custom skills since they 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 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 eight seconds.)
- Follow best practices when creating your DynamoDB tables. Use on-demand tables when you are not certain how much read or write capacity you need. You can use provisioned capacity with automatic scaling enabled if you know read and write capacity and do not expect large spikes in traffic. For Skills that are heavy on reads, 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 use 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 instead of paying for idle resources.

Reference architecture

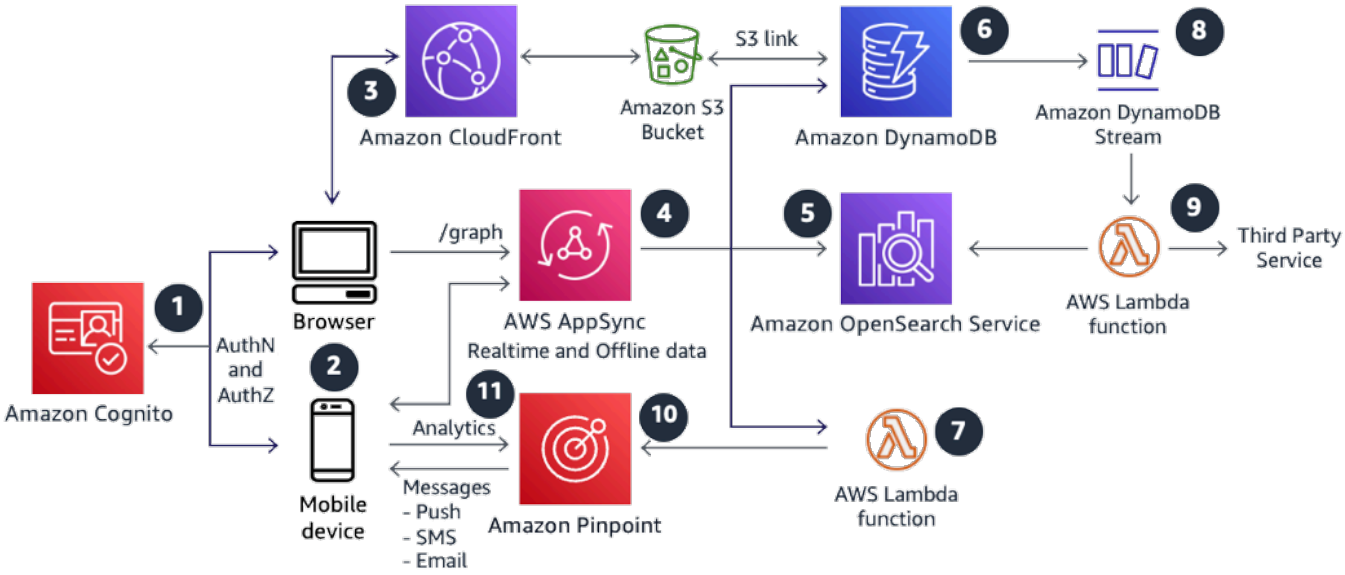


Figure 4: Reference architecture for a mobile backend

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 in real-time when devices are connected, and data is available offline

as well. Data sources for this scenario are Amazon DynamoDB, Amazon OpenSearch Service, or AWS Lambda functions.

5. **Amazon OpenSearch Service** acts as a main search engine for your mobile application as well as analytics.
6. **Amazon 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. An **AWS 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. **Amazon DynamoDB Streams** captures item-level changes and enables a Lambda function to update additional data sources.
9. An **AWS Lambda** function manages streaming data between DynamoDB and OpenSearch Service, 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 or 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 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 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 and 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 when managing Amazon OpenSearch Service domains. Additionally, Amazon OpenSearch Service provides an extensive guide 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.

Streaming 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:

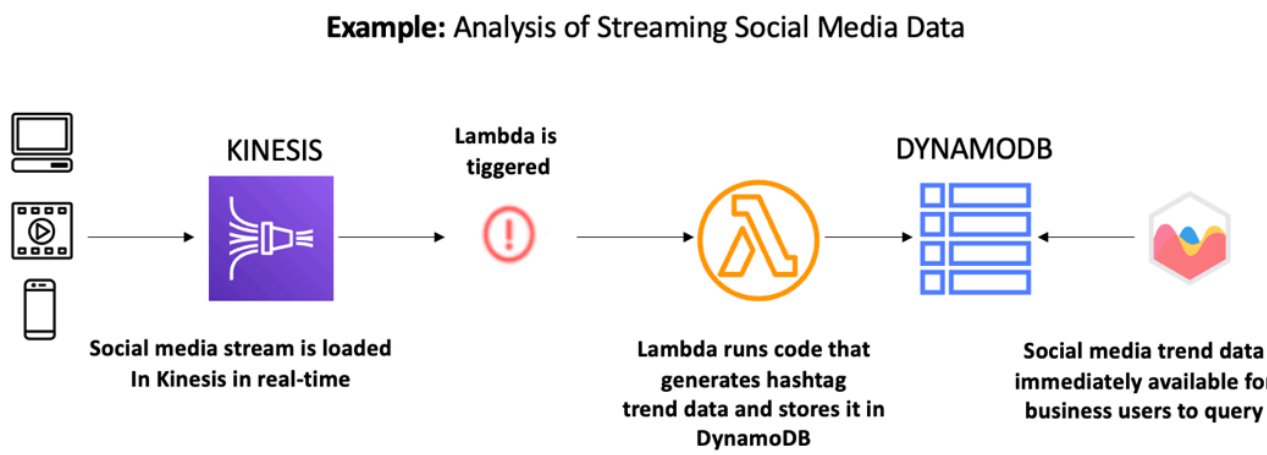


Figure 5: Reference architecture for stream processing

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 or 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** can use a reporting interface on top of DynamoDB to gather insights out of social media trend data.

Configuration notes

- Consider reviewing the Streaming Data Solutions whitepaper for batch processing, analytics on streams, and other useful patterns.
- Consider using Firehose over Lambda when ingested data needs to be continuously loaded into Amazon S3, Amazon Redshift, or Amazon OpenSearch Service.

- Consider using Managed Service for Apache Flink over Lambda when standard SQL or Apache Flink could be used to query streaming data, and load only its results into Amazon S3, Amazon Redshift, OpenSearch Service, or Kinesis Data Streams.
- 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. In addition consider using Lambda's custom checkpoint feature, where you can have more precise control over how you choose to process batches containing failed messages.
- Duplicated records may occur, and you must use both retries and idempotency within your application for both consumers and producers.
- Follow best practices for AWS Lambda stream-based invocation since that covers the effects on batch size, concurrency per shard, and monitoring stream processing in more detail.
- 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 upon ingestion time.
- If you are using Kinesis Data Streams in **provision capacity mode**, follow best practices when re-sharding Kinesis Data 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 using filtering event sources for AWS Lambda functions. Event filtering helps reduce requests made to your Lambda functions, may simplify code, and can reduce overall cost. At the time of writing, event filtering is only natively supported in CloudFormation and not in AWS CDK. If you are using AWS CDK you can still support Lambda Event filtering by using Escape Hatches to extend some functionality not directly available in CDK constructs.
- Consider using tumbling windows for AWS Lambda functions when you need to continuously calculate aggregates over a time period, such as 30-second averages. This feature allows you to do these types of aggregates without implementing a temporary datastore or using complicated streaming analytics frameworks.

Web application

Web applications often have demanding requirements to ensure a consistent, secure, and reliable user experience. Workloads which need to scale to thousands or millions of users require provisioning infrastructure for peak loads or sophisticated auto-scaling mechanisms, when available. On-premises workloads require significant capital expenditures and long lead times for capacity provisioning.

By taking a serverless-first approach on AWS you free yourself from the burden of managing servers, perfecting auto-scaling policies or paying for idle resources. Serverless workloads on AWS can provide the same, or better, security, reliability or performance when compared with server-based workloads.

Characteristics

- You want a scalable, resilient, and highly-available web application that can go global in minutes.
- You are seeking to reduce operational overhead by using managed services.
- You want to optimize your costs based on user demand and usage, instead of 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

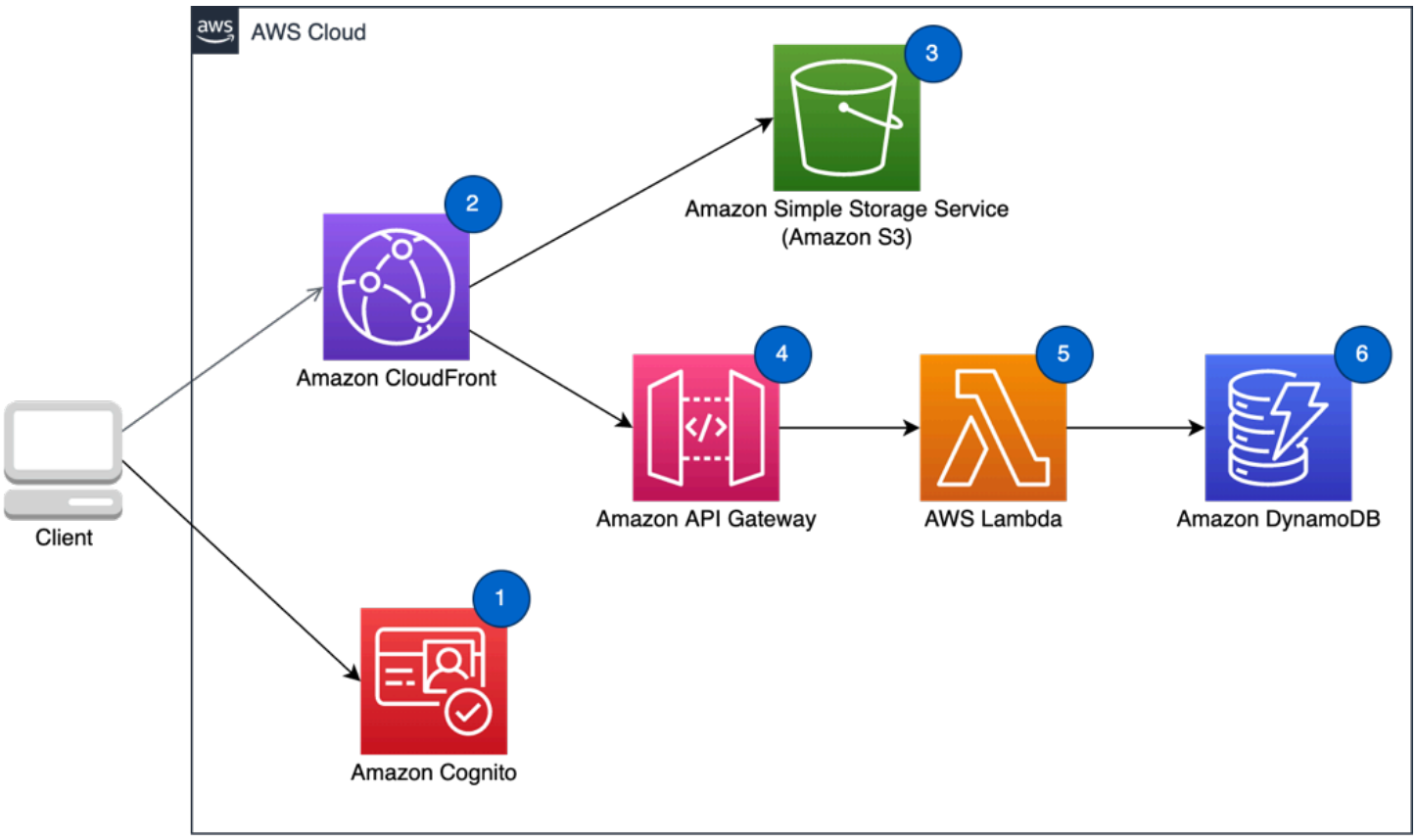


Figure 6: Reference architecture for a web application

1. **Amazon Cognito user pools** provides user management and identity provider features for your web application. Tokens issued by Amazon Cognito are used to authenticate users when making request to Amazon API Gateway.
2. **Amazon CloudFront** provides a better user experience by accelerating content delivery of static assets and calls to your backend compute layer. CloudFront brings content closer to clients using AWS's global Points of Presence (PoPs). CloudFront can also cache API calls to reduce calls to compute backends while also providing optimal network routing for non-cacheable API calls.
3. **Amazon S3** hosts web application static assets such as HTML, CSS, JavaScript and images. Content is securely served through CloudFront.
4. **Amazon API Gateway** serves as the secure HTTPS endpoint. Web applications make REST API calls to a public HTTPS endpoint using either a custom domain name or a unique API Gateway-provided domain.
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 a NoSQL data store which elastically scales with 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 you can use AWS Amplify Hosting to manage atomic deployments, cache expiration and custom domains.
- 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 use API Gateway usage plans. You can use Amazon Cognito user pools to scope users to specific resources or functionality. For example, a premium user may have higher throughput for API calls, access to additional APIs and additional storage.
- Refer to the Mobile Backend scenario if your application uses search capabilities that are not covered in this scenario.

Event-driven architectures

Event-driven architectures are becoming a popular and preferable way of building large distributed microservice-based applications. This approach helps you build scalable, resilient, agile and cost-effective solutions.

Using AWS serverless services to implement event-driven approach will allow you to build scalable, fault tolerant applications. You can use messaging services like Amazon SQS for reliable and durable communication between microservices. For fan out of the events you can use Amazon SNS topics. If you need event filtering and routing you can utilize Amazon EventBridge.

Typical use cases for event-driven architectures:

- Communication between microservices
- Integration with third-party SaaS applications
- Cross-account and cross region data replication
- Parallel event processing, fanout

Reference architecture

Every event-driven architecture consists of three main parts:

- Event sources
- Event routers
- Event destinations

The most common event sources could be other AWS services, your microservices or applications, or third-party SaaS offerings. For routing those events, you can create rules matching specific parts of the event and provide destinations of where to send them. You describe the rules and destinations with the help of Amazon EventBridge.

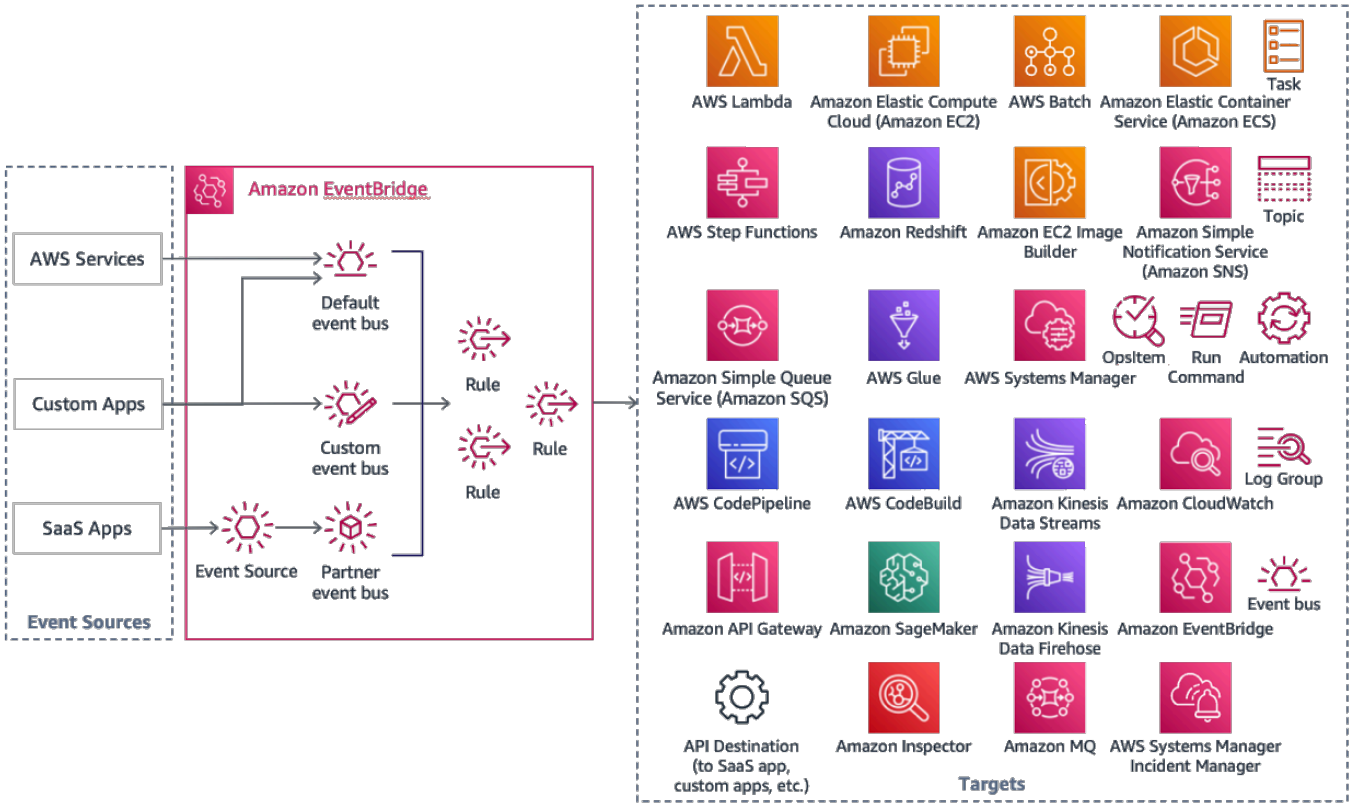


Figure 7: Reference architecture for EventBridge deployment

When you are building event-driven microservices applications, it's important to agree on the data contract for the event producers and consumers. This will help to validate the events and automatically generate bindings for the used programming language. Amazon EventBridge allows you to use schemas in OpenAPI 3 and JSONSchema Draft4 formats.

As all the event-driven applications are distributed it is important to use tracing to understand and observe service dependencies and diagnose any bottlenecks and issues in the application. To use tracing, you can enable integration between EventBridge and AWS X-Ray.

Configuration notes

EventBridge can introduce additional latency to the application. If latency is a concern, consider using Amazon SNS and Amazon SQS for event filtering and routing.

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.

Pillars:

- Operational excellence pillar
- Security pillar
- Reliability pillar
- Performance efficiency pillar
- Cost optimization pillar
- Sustainability pillar

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.

There are four best practices for operational excellence in the cloud:

- Organization
- Prepare
- Operate
- Evolve

The Well-Architected Framework Operational Excellence pillar covers many details and practices for operational health. There are specific areas where you can look to drive operational excellence within serverless applications.

Organization

There are no operational practices unique to serverless applications for this best practice.

Prepare

There are no operational practices unique to serverless applications for this best practice.

Operate

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

See the AWS Well-Architected Framework whitepaper for operational excellence best practices in the Operate section that apply to serverless applications.

Topics

- Metrics and alerts
- Centralized and structured logging
- Distributed tracing
- Prototyping
- Configuration
- Testing
- Deploying

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. Use Lambda Powertools for supported languages to create and capture custom CloudWatch metrics. When Lambda Powertools is not

available in your programming language of choice, use Amazon CloudWatch Embedded Metric Format (EMF) libraries. EMF logs emitted by Lambda are processed asynchronously by CloudWatch and do not impact 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 or credit card operations, flights purchased
- **Customer experience metrics**
  - Customer experience data dictates not only the overall effectiveness of its UI and 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
- **System metrics**
  - Vendor and application metrics are important to understand the health of your system, uncover root causes from the metrics above, and gain insight into customer experience.
  - Examples: Percentage of HTTP errors and successes, memory utilization, function duration, error, or throttling, queue length, stream records length, integration latency
- **Operational metrics**
  - Operational metrics are equally important to understand sustainability and maintenance of a given system. These metrics are also crucial to pinpoint how stability progressed or degraded over time.
  - Examples: Number of tickets (successful and unsuccessful resolutions), number of times people on-call were paged, availability, CI/CD pipeline stats (successful and failed deployments, feedback time, cycle and lead time)

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 are not limited to the following metrics:

- **AWS Lambda**: `Duration`, `Errors`, `Throttles`, and `ConcurrentExecutions`. For stream-based invocations, alert on `IteratorAge`. For asynchronous invocations, alert on `DeadLetterErrors`. When provisioned concurrency is enabled, use `ProvisionedConcurrencySpilloverInvocations`.
- **Amazon API Gateway**: `IntegrationLatency`, `Latency`, `5XXError`. For WebSocket API, use `ClientError`, `IntegrationError` and `ExecutionError`.
- **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`, `RenderingFailures`.
- **AWS Step Functions:** `ExecutionThrottled`, `ExecutionsFailed`, `ExecutionsTimedOut`, `ActivitiesTimedOut`, `LambdaFunctionsTimedOut`.
- **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 using

structured logging. Unstructured logging using `print` or `console.log` statements is unfavorable as they are difficult to interpret and analyze programmatically, hard to add contextual information to, and inconsistent. Structured logging libraries are advantageous because of configurable logging levels, API consistency and common output formats, among other things. Use logging utilities from Lambda Powertools to further simplify and enhance application logging.

JSON is a ubiquitous format which is often used as an output format and supported across logging services. CloudWatch Logs Insights automatically discovers values in JSON which makes querying and filtering simple. Judicious event logging from your application provides the ability to answer arbitrary questions about the state of your workload such as user behavior, state of your system and anomalous events, among other things. CloudWatch Logs Insights also facilitates finding Lambda performance data from default log events.

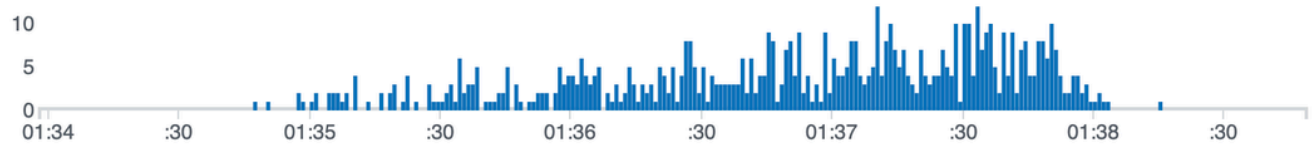
```
filter @type ="REPORT"
| parse @message /Init Duration: (?<init>\S+)/
| stats count() as total,
  count(init) as coldStarts,
  avg(init) as avgInitDuration,
  max(init) as maxInitDuration,
  avg(@maxMemoryUsed)/1000/1000 as memoryused
by bin (1min)
```

Run query | Save | History

Logs | Visualization | Export results | Add to dashboard

Showing 5 of 672 records matched | Hide histogram

9,708 records (4.9 MB) scanned in 3.6s @ 2,676 records/s (1.3 MB/s)



#	bin (1min)	total	coldStarts	avgInitDuration	maxInitDuration
1	2021-08-25T13:38:00.000-06:00	6	1	701.08	701.08
2	2021-08-25T13:37:00.000-06:00	339	61	731.2239	1104.92
3	2021-08-25T13:36:00.000-06:00	223	56	746.0004	1397.44
4	2021-08-25T13:35:00.000-06:00	99	52	720.6308	1181.89
5	2021-08-25T13:34:00.000-06:00	5	5	708.602	744.56

Figure 8: CloudWatch Logs Insights query to find statistics on cold starts

Consistently logging of correlation IDs and passing them to downstream systems allows tracing and tracking of individual requests or invocations. 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. Log level configuration should be passed to downstream systems for consistent tracing in a microservice architecture.

The Lambda Logs API can be used to send Lambda logs to locations other than CloudWatch. A number of partner solutions provide Lambda layers which use the Lambda Logs API and make integration with their systems easier. The recommendations and guidance here applies uniformly regardless of log destination.

The following 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
            }
        }
    }
}
```

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.

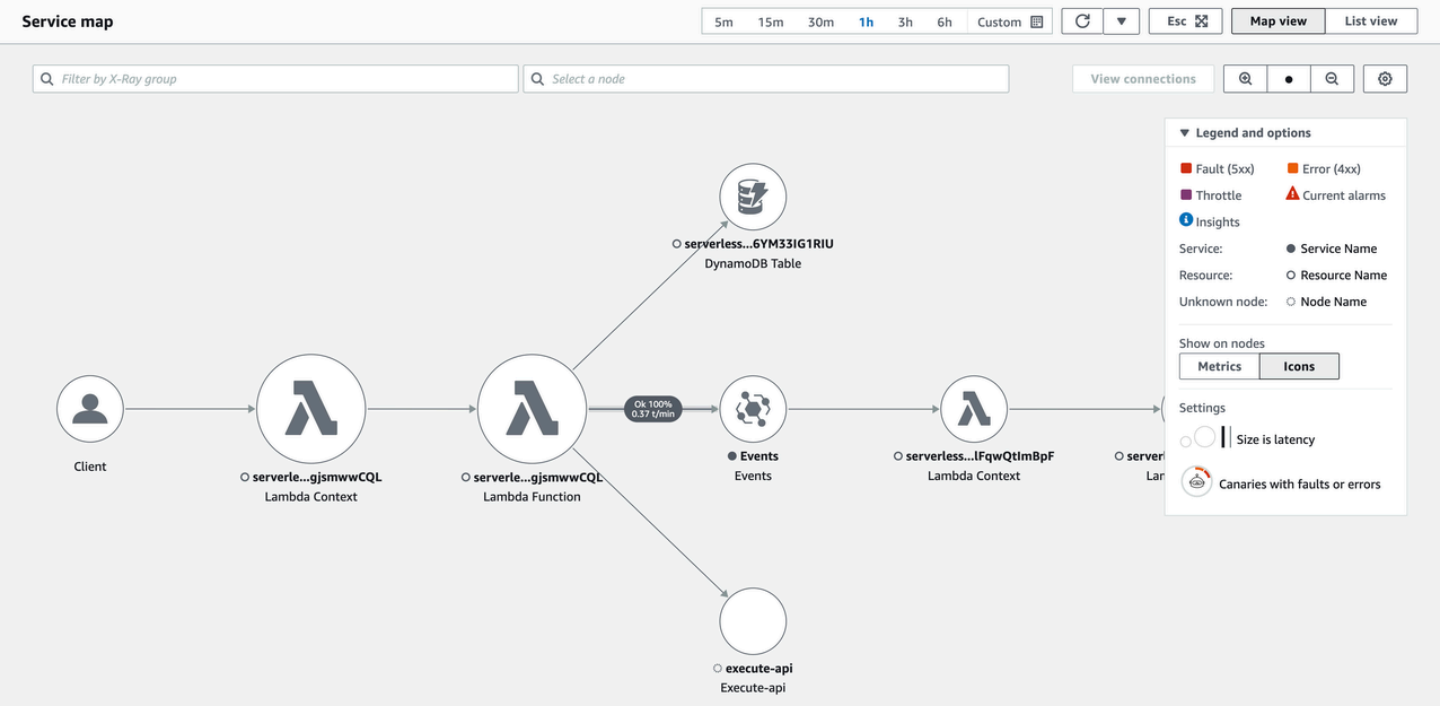


Figure 9: AWS X-Ray Service Map visualizing a workload using AWS Lambda, Amazon DynamoDB and Amazon EventBridge

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, subsegments and annotations can help you quickly identify performance statistics on specific operations and business transactions. Examples are a database query duration, or the durations of a supporting function which parses an image.

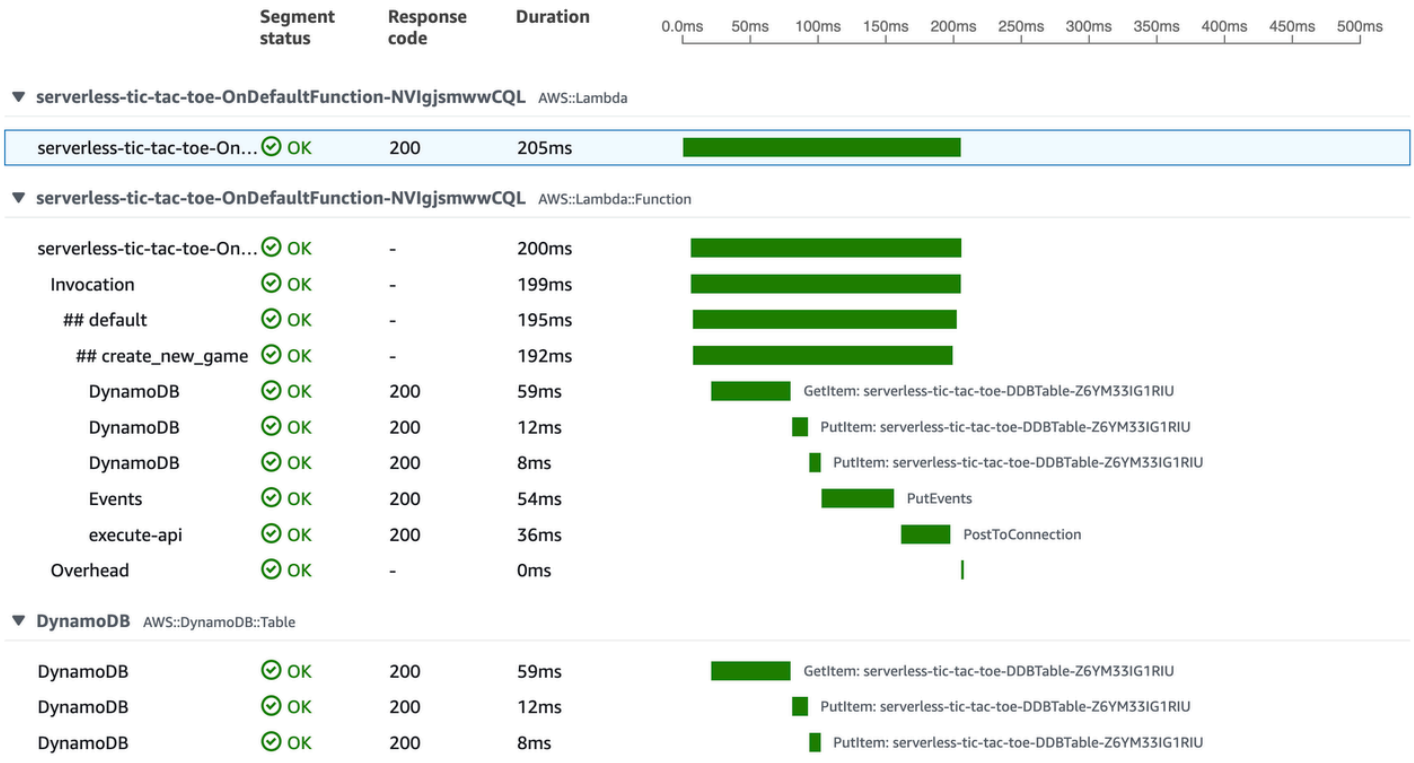


Figure 10: AWS X-Ray Trace with subsegments beginning with ##

Prototyping

OPS 2: How do you approach application lifecycle management?

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.

Configuration

For configuration management, use environment variables for infrequent changes, such as logging level and database connection strings. Use AWS Systems Manager Parameter Store (SSM) or AWS AppConfig for dynamic configuration, such as feature toggles. Store sensitive data using AWS Secrets Manager. In Lambda functions, lookup values by reference from these external systems (SSM, AWS AppConfig, Secrets Manager) in the function's global scope outside the handler to reduce API calls. You can achieve the same goal of reducing API calls to configuration and secrets stores using Lambda extensions which provide more fine-grained controls and the ability to re-fetch values. Lambda extensions are powerful and flexible yet bring additional considerations and challenges including integrations with unit tests and consistent delivery across functions and runtimes. Lambda Powertools offer similar functionality to retrieve values from various providers including SSM, AWS AppConfig, Secrets Manager, DynamoDB or custom stores.

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 be designed to 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 maximum memory used and `init` duration are available in CloudWatch Metrics. For more information, see the performance pillar section.

If your Lambda function is attached to Amazon Virtual Private Cloud (Amazon VPC), pay attention to the 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 or initialization code (global scope, outside the handler) will ensure that external connection thresholds are not reached if the Lambda execution environment is reused.

Use a DynamoDB on-demand table unless your performance tests exceed current quotas in your account.

Take into account any other service quotas 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 Serverless Application Model or Serverless Framework. With infrastructure as code (IaC) and a framework, you can add parameters to your serverless application and its dependencies to ease deployment across isolated stages and across AWS accounts.

Infrastructure frameworks like AWS Cloud Development Kit (AWS CDK) and Terraform play important roles when managing AWS resources. Serverless-specific tools like AWS SAM and the

Serverless Framework bring unique features to speed day-to-day development and are purpose-built to minimize the code, test, and deploy loop.

Create separate stages or environments using CI/CD pipelines (for example, Gamma, Dev, and Prod). A CI/CD pipeline can create the following resources in a beta AWS account: `OrderAPIBeta`, `OrderServiceBeta`, `OrderStateMachineBeta`, `OrderBucketBeta`, and `OrderTableBeta`. Similar, yet separate, resources can be created across different environments which might reside in separate AWS accounts.

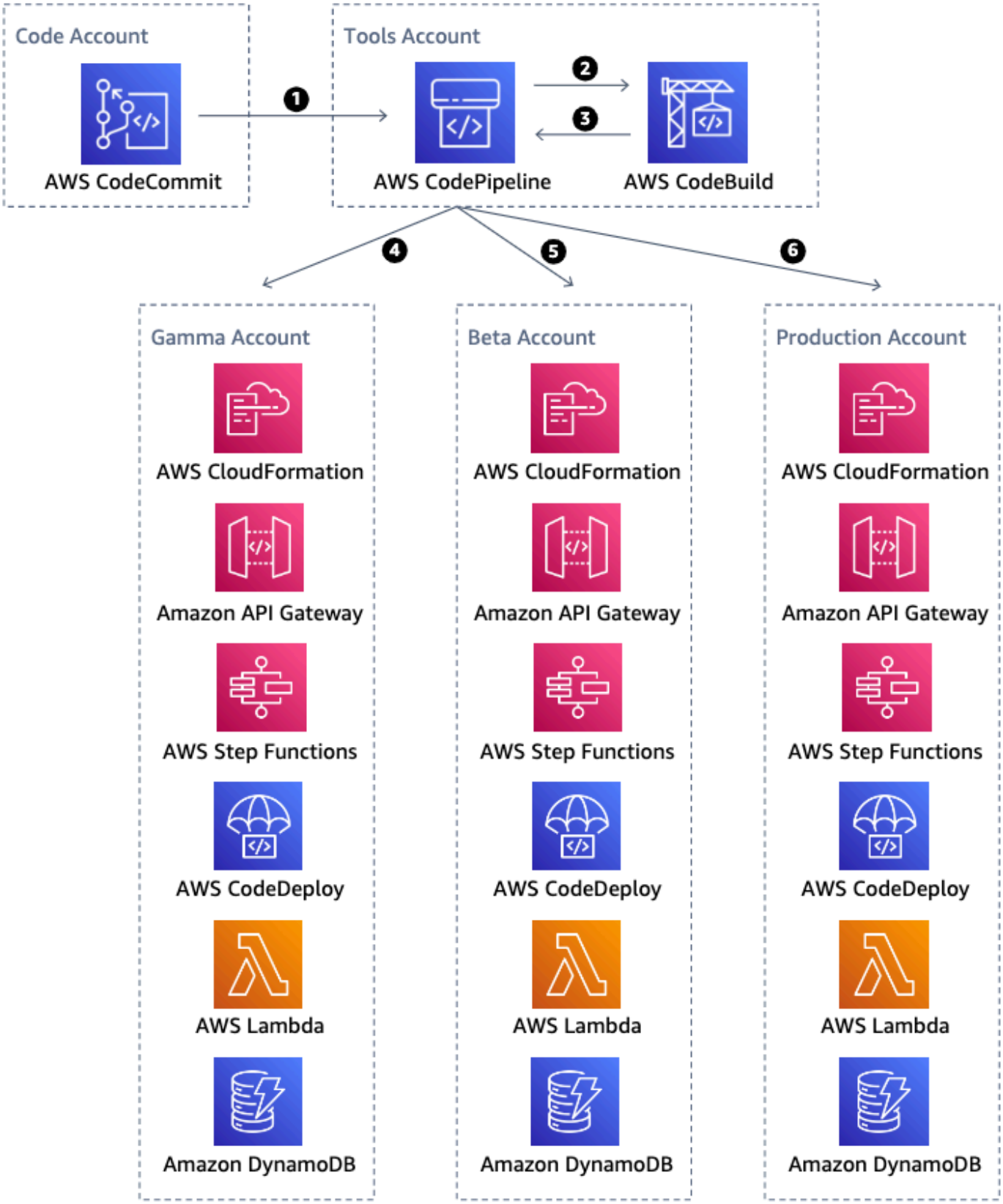


Figure 11: CI/CD pipeline for multiple accounts

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.

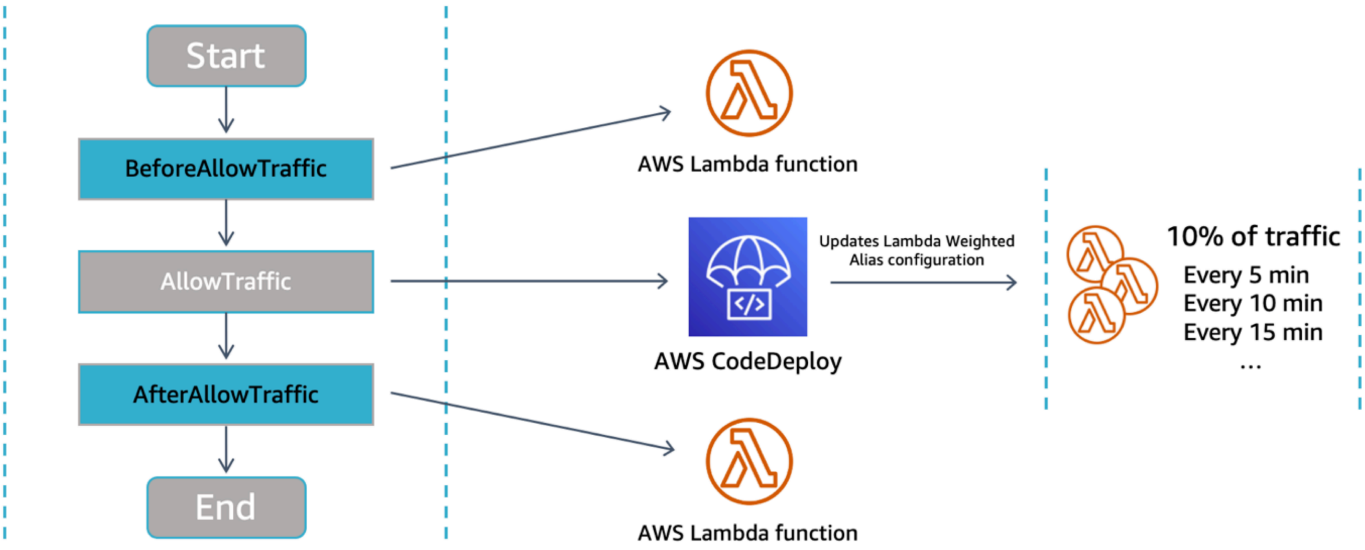


Figure 12: AWS CodeDeploy Lambda deployment and hooks

Evolve

There are no operational practices unique to serverless applications for this best practice.

Key AWS services

Key AWS services for operational excellence include AWS Systems Manager Parameter Store, AWS Serverless Application Model, 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 and blogs

- AWS SAM
- API Gateway stage variables
- Lambda environment variables
- Powertools for AWS Lambda (Python)
- Powertools for AWS Lambda (TypeScript)
- Powertools for AWS Lambda (Java)
- Powertools for AWS Lambda (.NET)
- CloudWatch Embedded Metric Format library for Python
- CloudWatch Embedded Metric Format library for Node.js
- CloudWatch Embedded Metric Format library for Java
- CloudWatch Embedded Metric Format library for .NET
- Operating Lambda: Logging and custom metrics
- Operating Lambda: Using CloudWatch Logs Insights
- Common CloudWatch Logs Insights queries
- Using AWS Lambda extensions to send logs to custom destinations
- Building well-architected serverless applications blog series
- X-Ray latency distribution
- Troubleshooting Lambda-based applications with X-Ray
- System Manager (SSM) Parameter Store
- AWS AppConfig integration with Lambda Extensions
- AWS Secrets Manager
- Cache secrets using AWS Lambda extensions
- Serverless Application example using CI/CD
- CI/CD for Serverless Applications - Workshop
- Serverless CI/CD for the Enterprise on AWS - Reference Deployment
- Using GitHub Actions to deploy serverless applications
- Serverless Application example automating Alerts and Dashboard
- AWS service quotas

- Stackery: Multi-Account Best Practices

Whitepapers

- Practicing Continuous Integration/Continuous Delivery on AWS

Third-party tools

- Serverless Developer Tools page including third-party frameworks/tools
- 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.

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 because it removes infrastructure management tasks such as operating system patching and updating binaries. 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 reviewed frequently.

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

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 and authorization perspective, there are currently five mechanisms to authorize an API call within API Gateway:

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

It is important to understand if, and how, any of these mechanisms are implemented. For consumers who are currently 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:

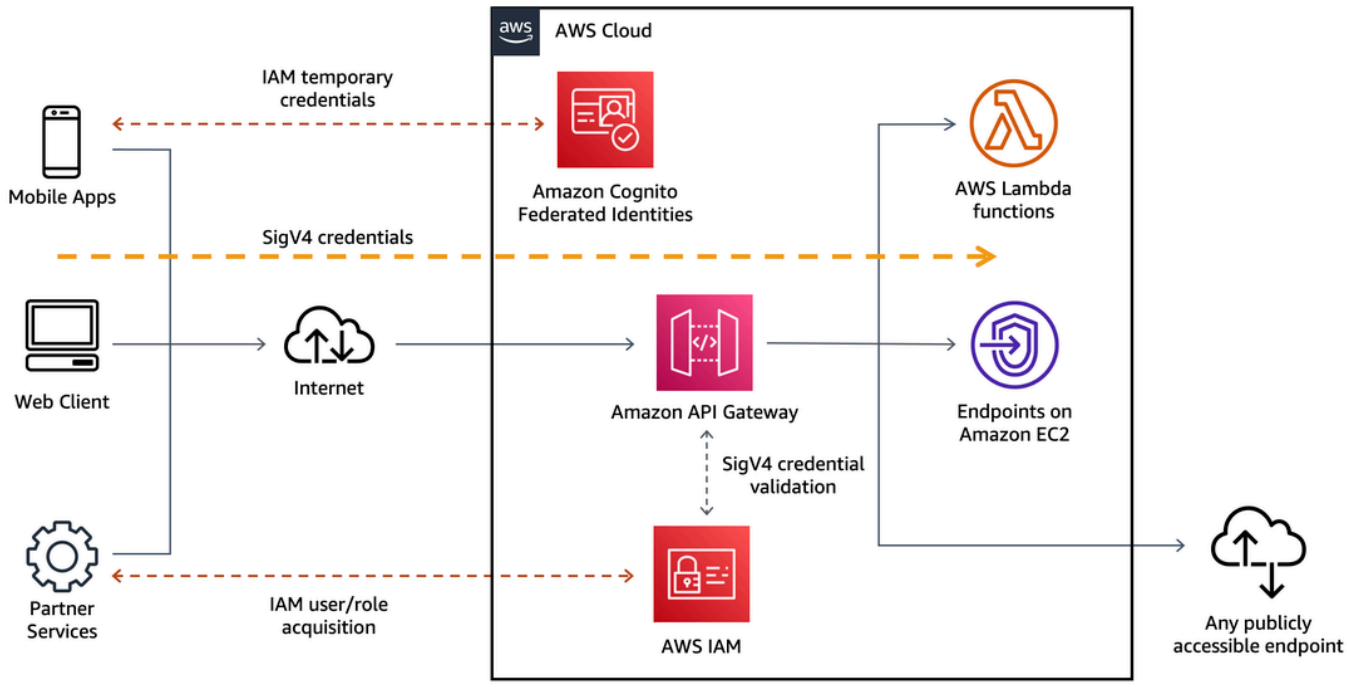


Figure 13: AWS_IAM authorization

To add granularity into your IAM authorization you can implement tag-based access control, which allows for better API-level control on the resources and actions.

If you have an existing Identity Provider (IdP), you can use an API Gateway Lambda authorizer to invoke a Lambda function to authenticate or 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.

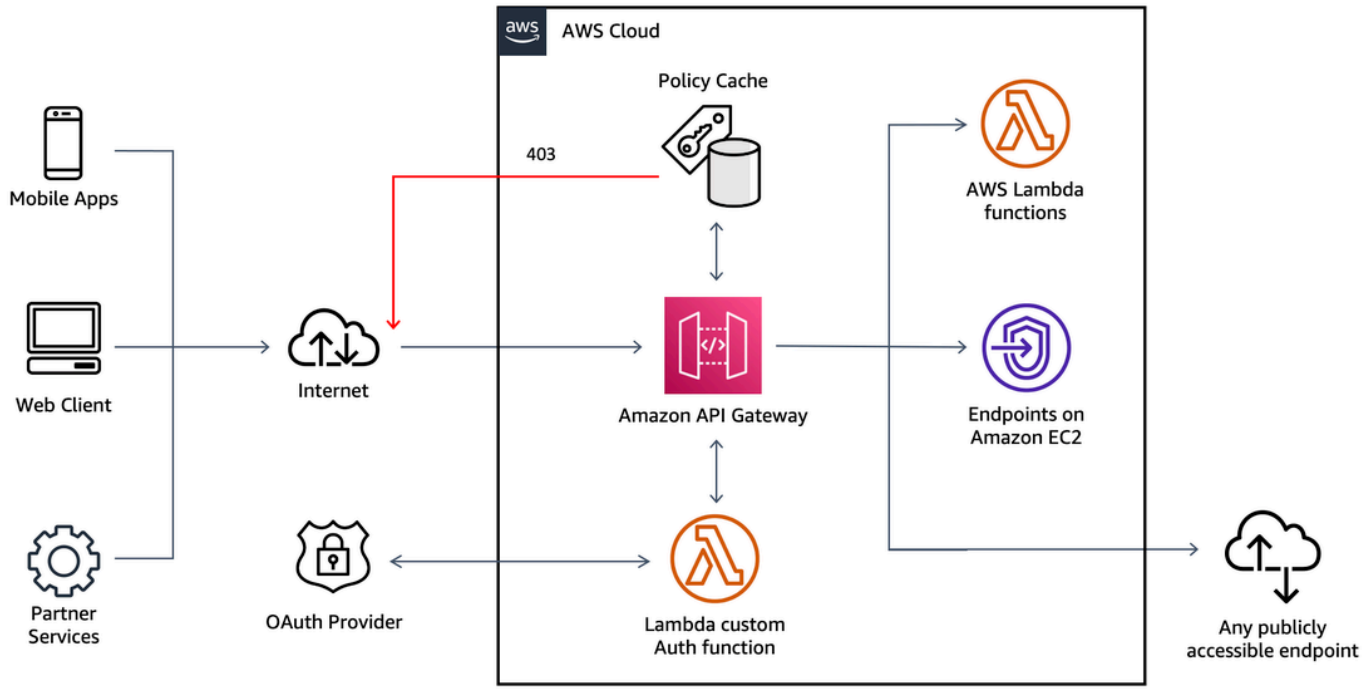


Figure 14: API Gateway Lambda authorizer

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 to register or sign in with their email address or username. This approach also provides granular authorization through OAuth Scopes.

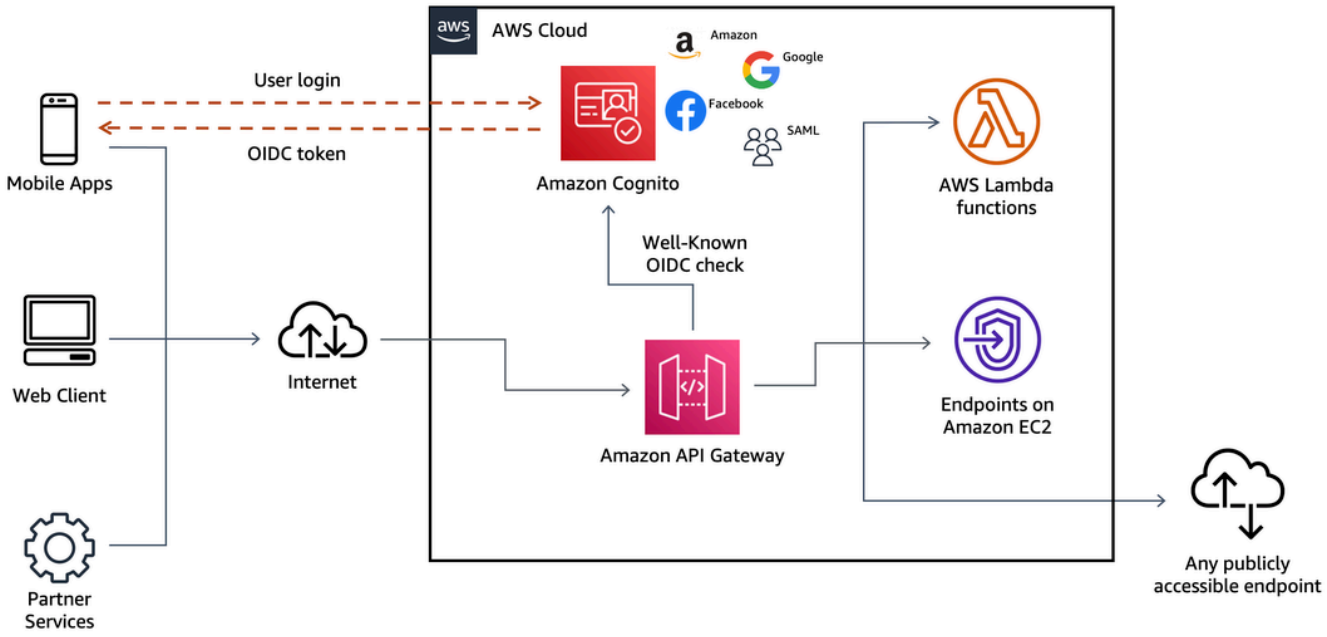


Figure 15: Amazon Cognito user pools

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 through 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.

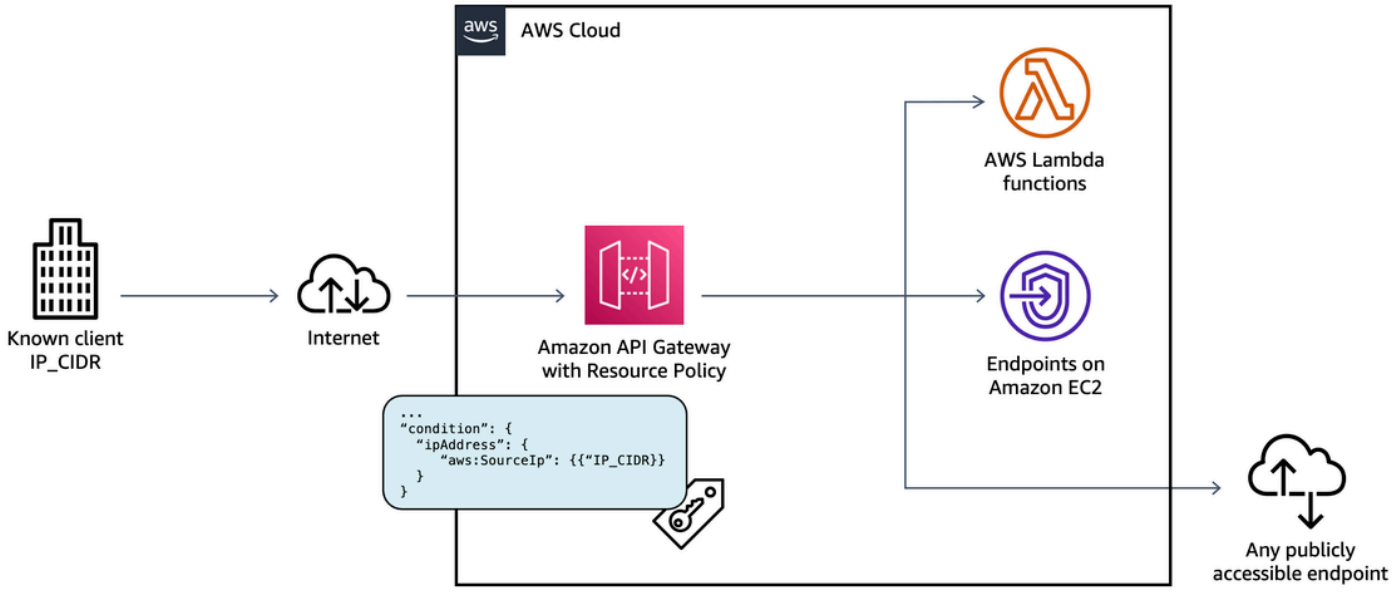


Figure 16: Amazon API Gateway Resource Policy based on IP CIDR

With private endpoints, API Gateway will restrict access to services and resources inside your VPC, or those connected through Direct Connect to your own data centers. To control access to the VPC Endpoint you can add VPC endpoint policies so that you can grant or deny the access to a particular APIs for the traffic going in your internal network. Combining private endpoints, endpoint policies, and resource policies, an API can be limited to specific resource invocations within a specific private IP range from a specific VPC endpoint. This combination is mostly used on internal microservices where they may be in the same account, or another account. If you are using API Gateway as a main endpoint to your backend HTTP(s) services you can enable client-side SSL certificates so that the backend services can authenticate and verify requests from API Gateway. When it comes to large deployments and multiple AWS accounts, organizations can use 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.

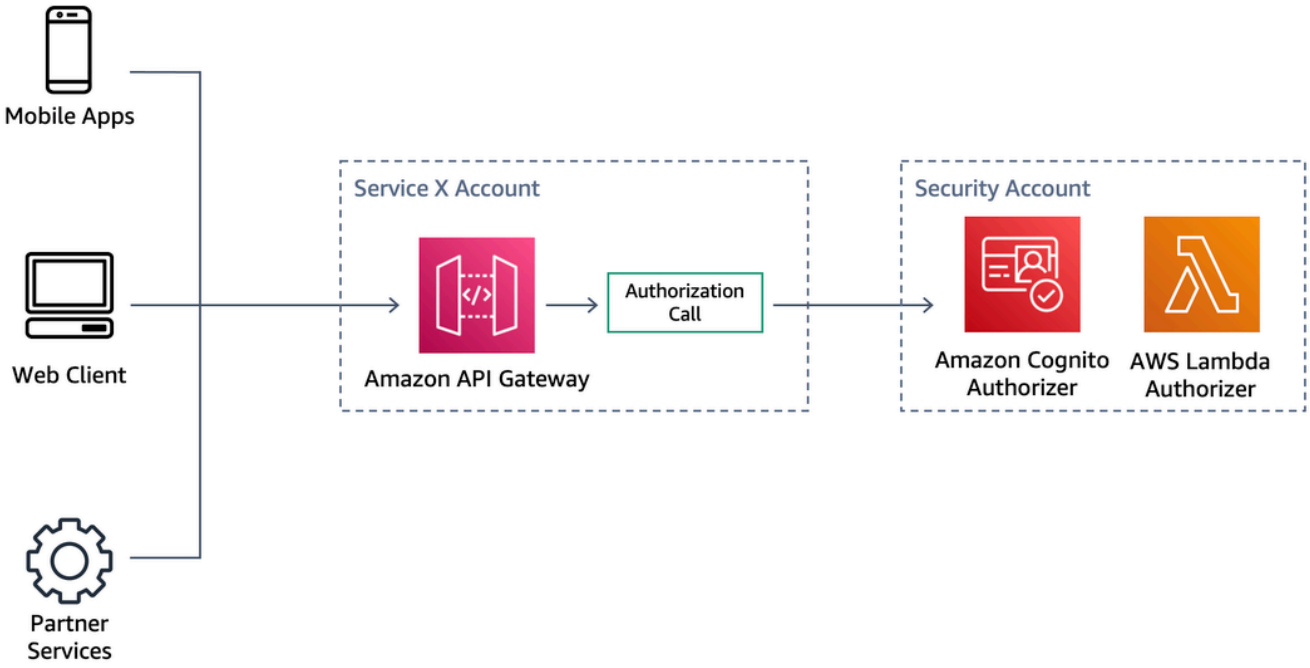


Figure 17: API Gateway cross-account authorizers

For cases like Internet of Things (IoT) or application-to-application authentication, you can configure a mutual TLS (mTLS) authentication. In this scenario, the client should present its certificate to verify its identity when accessing API Gateway endpoint. You can also combine mTLS with Lambda authorizers for a more granular authorization mechanism.

You can use AWS WAF to add protection of your APIs on the application network layer. You can use managed rule groups to protect your APIs against well known attacks like SQL injection and cross-site scripting (XSS), or if you have additional requirements you can also create your own rule groups.

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 and 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, OpenSearch Service, 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 or query strings, or in the 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.

To add protection for your code executing in Lambda runtime against any unintended and unauthorised changes while it is moving in your CI/CD pipelines, you can add code signature. Signing the code will confirm that it comes from a trusted source and is unaltered. AWS Signer integrates with AWS Lambda to sign the code and enforce that only trusted code is deployed into your runtime.

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.

Incident response

There are no security practices unique to serverless applications for this best practice.

Key AWS services

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

Resources

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

Documentation and blogs

- AWS Lambda permissions
- API Gateway Request Validation
- API Gateway Lambda Authorizers
- Building fine-grained authorization using Amazon Cognito, API Gateway, and IAM
- Configuring VPC Access for AWS Lambda
- Using AWS Secrets Manager with Lambda

- Caching data and configuration settings with AWS Lambda extensions
- Auditing Secrets with AWS Secrets Manager
- OWASP Input validation cheat sheet
- AWS Serverless Security Workshop
- Code signing for Lambda

Whitepapers

- Security Overview of AWS Lambda
- OWASP Top Ten
- OWASP Secure Coding Best Practices
- Snyk – Commercial Vulnerability DB and Dependency Check
- Using Hashicorp Vault with Lambda & API Gateway

Third-party tools

- OWASP Vulnerability Dependency Check

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.

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.

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 or 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 or 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 or 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 an HTTP 429 error, Asynchronous invocations will be queued and retried, while Stream-based event sources will retry up to their record expiration time.

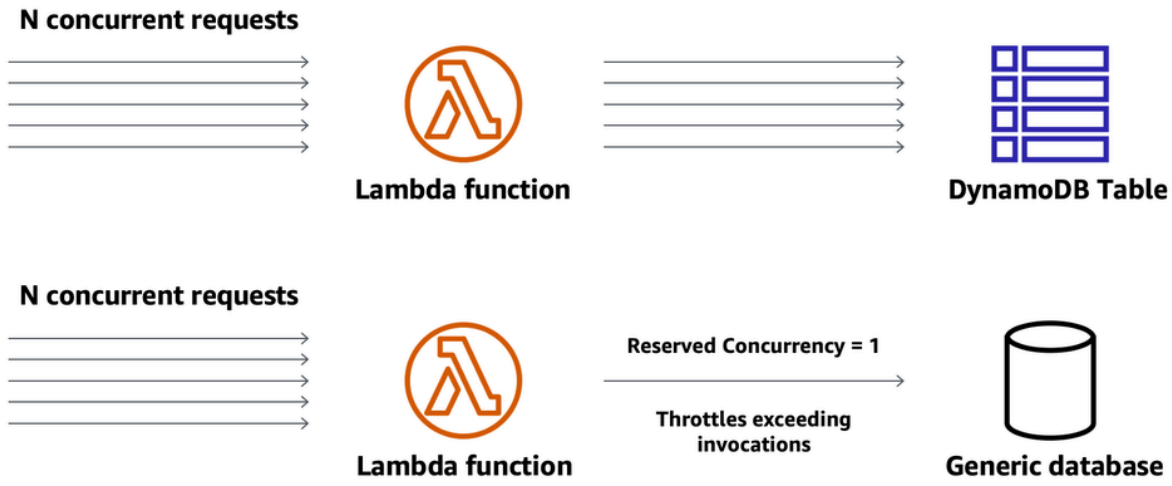


Figure 18: AWS Lambda concurrency controls

Controlling concurrency is particularly useful for the following scenarios:

- Sensitive backend or integrated systems that may have scaling limitations: In situations when your Lambda functions call some legacy or sensitive backend, they may put too much pressure on the downstream services since functions may scale too fast and produce many concurrent requests. It is a good idea to limit the concurrency of your functions so that you can control the amount of requests they produce.
- Protecting against recursive invocations: You may introduce a recursive call of your Lambda functions accidentally. One of the most common cases is when using S3 - Lambda - S3 pattern

reading, and then writing into the same S3 bucket. Limiting concurrency will let you decrease the implications of such recursive calls and help you detect and fix them earlier.

- Database Connection Pool restrictions, such as a relational database, which may impose concurrent limits: Many RDBMS have restrictions on the number of opened connections. Limiting concurrency of the Lambda functions will allow you to limit the number of opened connections. If using Amazon RDS databases consider using Amazon RDS Proxy as a connection pooling mechanism.
- Critical Path Services: Ensure that high priority Lambda functions, such as authorization, do not run out of concurrency due to runaway invocations from low priority functions (for example, backend asynchronous processes). Since Lambda concurrency quotas are applied per account and Region, it's possible for one function to consume concurrency such that other functions are throttled.
- Ability to disable Lambda function (`concurrency = 0`) in the event of anomalies: In case of failures, setting concurrency to zero will help you to immediately stop new invocations of your Lambda functions.
- Limiting desired execution concurrency to protect against Distributed Denial of Service (DDoS) attacks: Usually the protection against DDoS is done at the API Gateway level, but it is also a good idea to introduce an additional guard rail on the function level.

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.

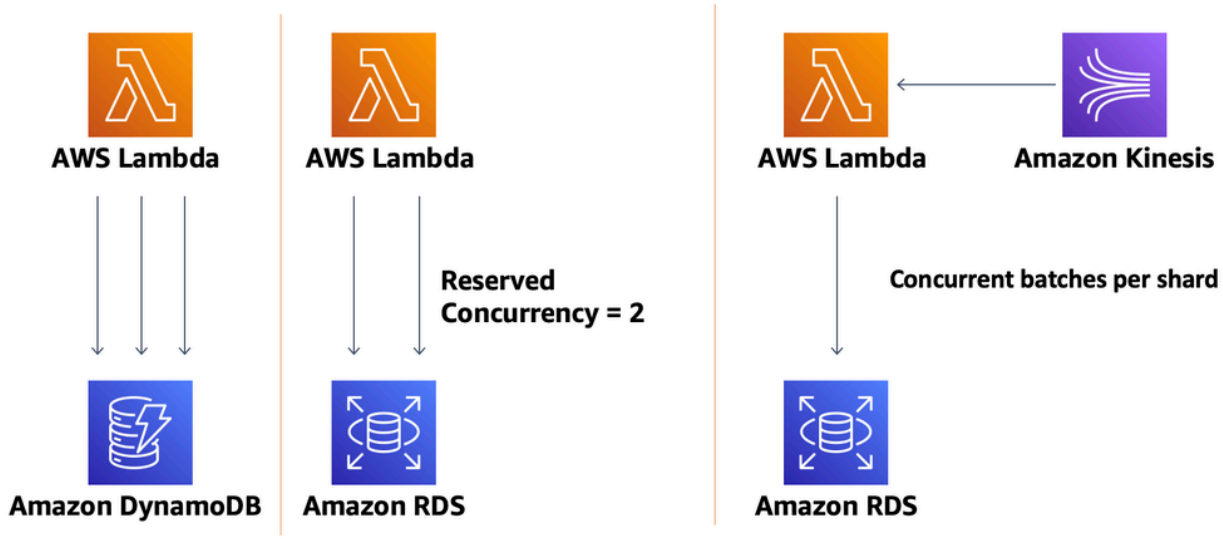


Figure 19: Concurrency controls for synchronous and asynchronous requests

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

Best practices

- Manage transaction, partial, and intermittent failures: Transaction failures might occur when components are under high load. Partial failures can occur during batch processing, while intermittent failures might occur due to network or other transient issues.
- Manage duplicate and unwanted events: Duplicate events can occur when a request is retried, multiple consumers process the same message from a queue or stream, or when a request is sent twice at different time intervals with the same parameters. Design your applications to process multiple identical requests to have the same effect as making a single request. Events not adhering to your schema should be discarded.
- Orchestrate long-running transactions: Long-running transactions can be processed by one or multiple components. Favor state machines for long-running transaction instead of handling

them within application code in a single component or multiple synchronous dependency call chains.

- Consider scaling patterns at burst rates: In addition to your baseline performance, consider evaluating how your workload handles initial burst rates that may be expected or unexpected peaks.

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 initiations 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 or 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, whether it's 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 run 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 states, and handle error handling that occurs along the request lifecycle.

Change management

There are no operational practices unique to serverless applications for this best practice.

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.

Use a dead-letter queue mechanism to retain, investigate, and retry failed transactions.

- AWS Lambda allows failed transactions to be sent to a dedicated Amazon SQS dead-letter queue on a per function basis.
- Amazon Kinesis Data Streams and Amazon DynamoDB Streams retry the entire batch of items. Repeated errors block processing of the affected shard until the error is resolved or the items expire.
- Within AWS Lambda, you can configure **Maximum Retry Attempts**, **Maximum Record Age** and **Destination on Failure** to respectively control retry while processing data records, and effectively remove poison-pill messages from the batch by sending its metadata to an Amazon SQS dead-letter queue for further analysis.

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 example, you can use a Step Functions integration to save failed state runs and their state into a DLQ. For more information on costs trade-offs, see the cost optimization pillar section.

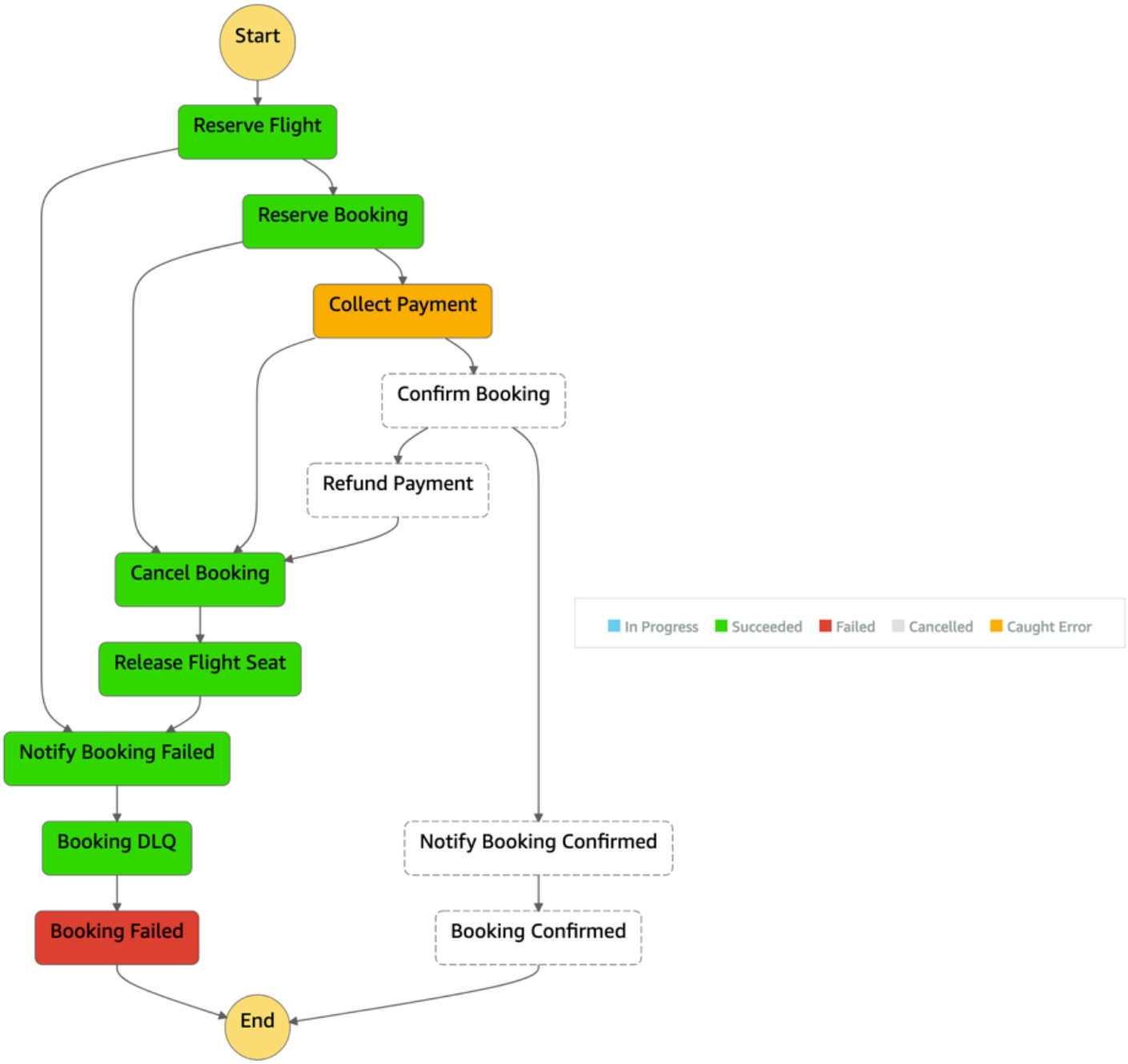


Figure 20: Step Functions state machine with DLQ step

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 also can be achieved by using Step Functions state machines, which will decouple and simplify the logic of your application.

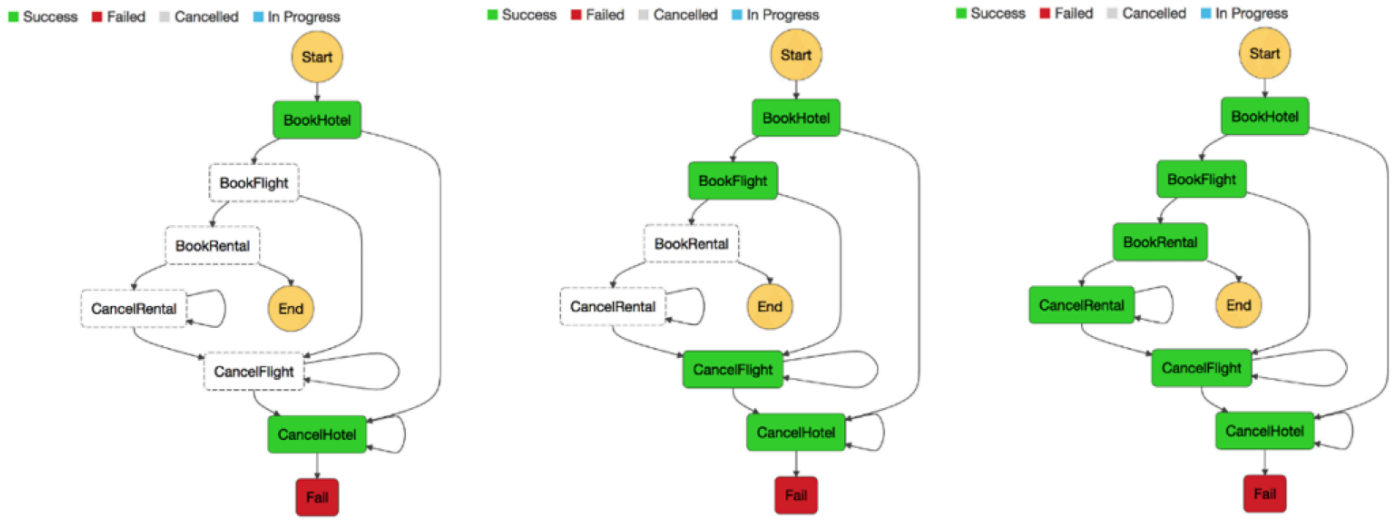


Figure 21: Step Functions state machine Saga pattern

Choose the Step Functions type based on your workload. For short-running synchronous and asynchronous high-volume workloads, use Step Functions - Sync Express. If you need to automate long-running workflows and want to have additional durability and audit go with Step Functions Standard.

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 testing 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 and blogs

- Quotas in Lambda
- Quotas in API Gateway
- Quotas and Limits in Kinesis Streams
- Service, Account, and Table Quotas in DynamoDB
- Quotas in Step Functions
- Getting started with testing serverless applications
- Monitoring Lambda Functions Logs
- Versioning Lambda
- Stages in API Gateway
- API Retries in AWS
- Step Functions error handling
- AWS Lambda and AWS X-Ray
- Error handling and automatic retries in AWS Lambda
- Lambda DLQ
- Lambda destinations
- Step Functions Wait state
- Step Functions Standard vs. Express Workflows
- Saga pattern
- Applying Saga pattern with Step Functions
- Designing durable serverless apps with DLQs for Amazon SNS, Amazon SQS, AWS Lambda
- Troubleshooting retry and timeout issues with AWS SDK
- Lambda resiliency controls for stream processing

Whitepapers

- Implementing Microservices on AWS
- Disaster Recovery of Workloads on AWS

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.

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.

Selection

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

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

- **Amazon API Gateway:** Use Edge endpoints for geographically dispersed customers. Use Regional for regional customers and when using other AWS services within the same Region.
- **AWS Lambda:** Test different memory settings since CPU, network, and storage IOPS are allocated proportionally. Optimize static initialization and consider provisioned concurrency.
- **AWS Step Functions**: Test Standard and Express Workflows, consider the per second rates for both execution start rate and state transition rate.

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

Amazon API Gateway

To build RESTful APIs, use REST APIs from Amazon API Gateway. REST APIs are intended for APIs that require API proxy functionality and API management features in a single solution.

Amazon API Gateway Edge-optimized APIs provide a fully managed Amazon 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.

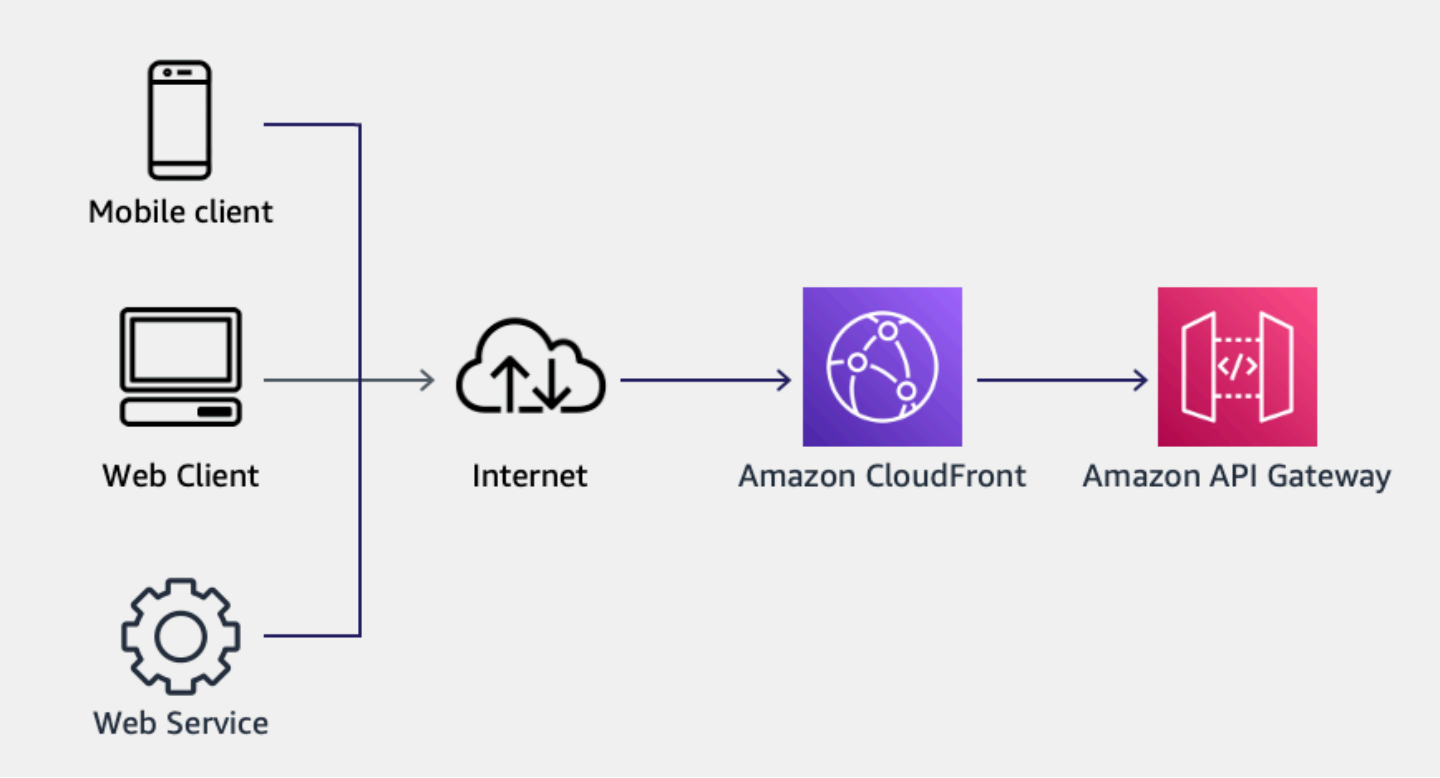


Figure 22: Edge-optimized API Gateway deployment

The 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.

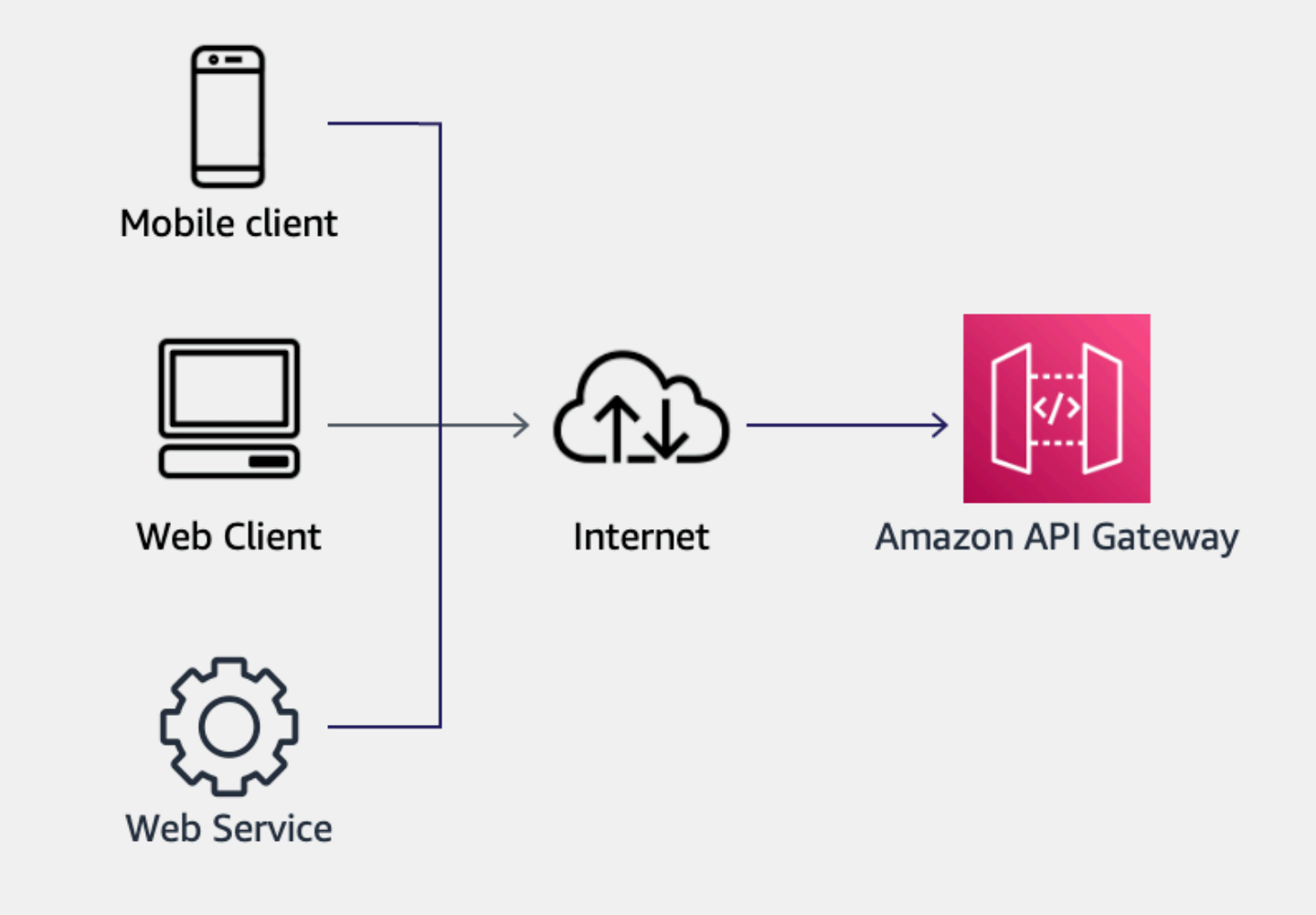


Figure 23: Regional Endpoint API Gateway deployment

This table can help you decide whether to deploy an 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		X

	Edge-optimized API	Regional API Endpoint
the same Region as API is deployed.		
Ability to associate own CloudFront distribution.		X

AWS Lambda

Provisioned concurrency initializes a requested number of execution environments so that they are prepared to respond immediately to your function's invocations. To enable your function to scale without fluctuations in latency, use provisioned concurrency. By allocating provisioned concurrency before an increase in invocations, you can ensure that all requests are served by initialized instances with very low latency. AWS Lambda also integrates with Application Auto Scaling. You can configure Application Auto Scaling to manage provisioned concurrency on a schedule or based on utilization. Use scheduled scaling to increase provisioned concurrency in anticipation of peak traffic.

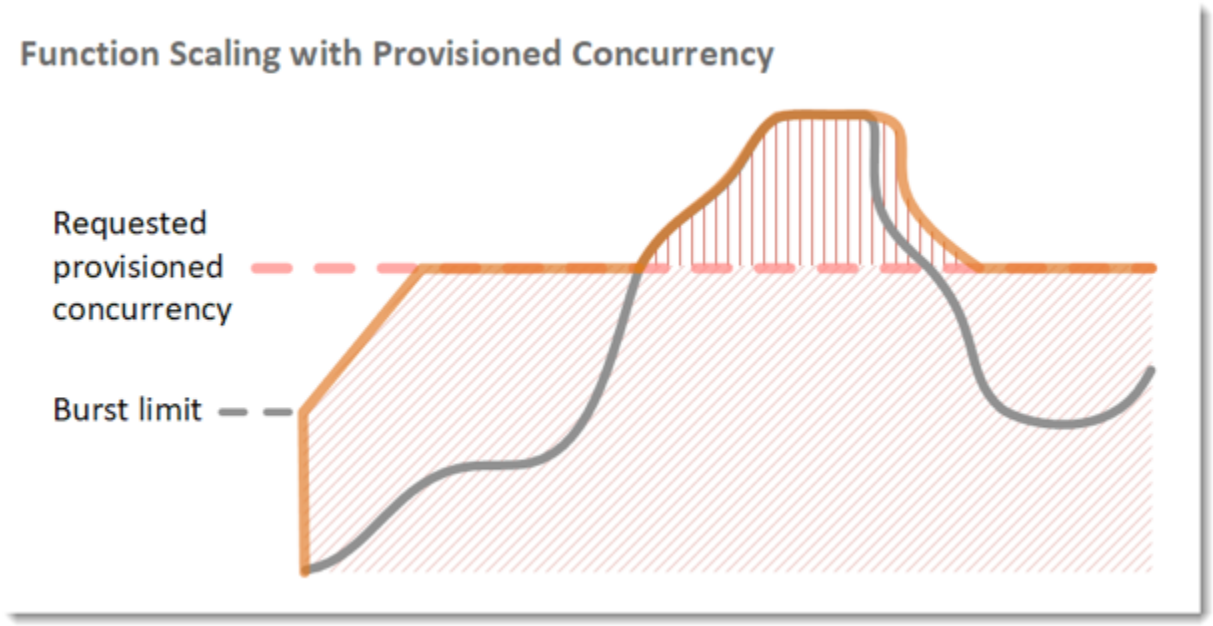


Figure 24: Provisioned concurrency initializes a requested number of execution environments to respond immediately to function's invocations

To optimize latency, you can customize the initialization behavior for functions that use provisioned concurrency. You can run initialization code for provisioned concurrency instances

without impacting latency, because the initialization code runs at allocation time. Configure Amazon 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. Be sure to check both the Security Group and network Access Control List (ACL) to allow outbound requests from your Lambda function. As covered in the AWS Well-Architected Framework, configure your NAT gateway, or NAT instances across multiple Availability Zones for high availability and performance. This decision tree can help you decide when to deploy your Lambda function in a VPC.

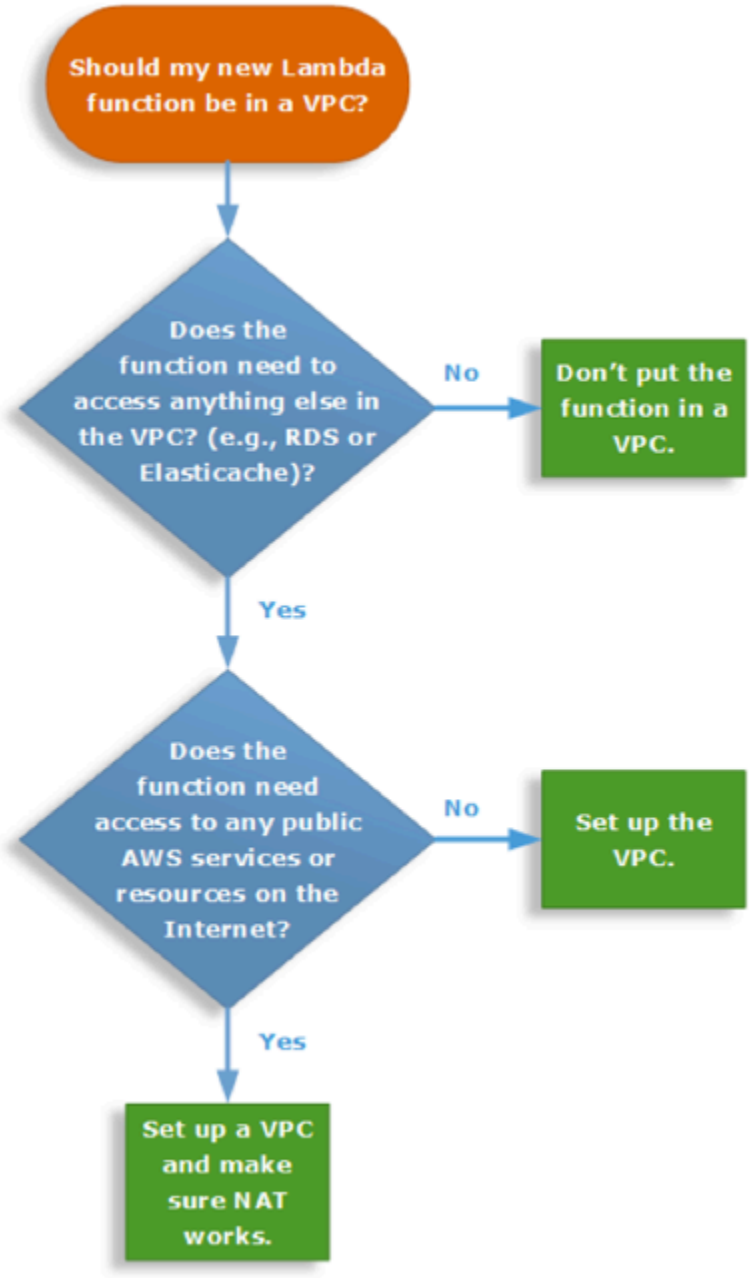


Figure 25: Decision tree for deploying a AWS Lambda function in an Amazon VPC

For Lambda functions in VPC, avoid DNS resolution of public host names for 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.

AWS Step Functions

AWS Step Functions offers both Standard or Express Workflow types. Standard Workflows are ideal for long-running, durable, and auditable workflows. Express Workflows are ideal for high-volume, event-processing workloads such as IoT data ingestion, streaming data processing and transformation, and mobile application backends. A Standard Workflow has a maximum duration of 1 year, compared to 5 minutes for an Express Workflow. Both Standard and Express Workflows support execution history logging to Amazon CloudWatch Logs. Publishing logs doesn't block or slow down executions, allowing you to select the log level required for the workflow. When inspecting a workflow consider using Amazon CloudWatch Logs Insights to interactively search and analyze your workflow log data. Using the GetExecutionHistory API to explore the execution history for Standard Workflows may save you from writing code. AWS Step Functions state transitions are throttled using a token bucket scheme. Estimate the state transitions expected and match to quotas for bucket size and refill rates. Trade off between Standard Workflow with throttling, or Express Workflow with unlimited bucket size and refill rate.

An Express Workflow can start either synchronously or asynchronously. Select a synchronous invocation when you can wait for the result and prefer to develop applications without the need to develop additional code to handle errors, retries, or execute parallel tasks. Synchronous Express execution API calls do not contribute to the existing account capacity limits. Step Functions will provide capacity on demand and will automatically scale with sustained workloads. Surges in workloads may be throttled until capacity is available. A Synchronous Express Workflow can be invoked from Amazon API Gateway, AWS Lambda, or by using the StartSyncExecution API call.

Invoke an Asynchronous Express Workflow if you don’t require an immediate response output such as messaging services, or data processing that other services don’t depend on. An Asynchronous Express Workflow returns a confirmation the workflow has started, and you poll Amazon CloudWatch Logs for the result. An Asynchronous Express Workflow can be started in response to an event, by a nested workflow in Step Functions, or by using the StartExecution API call.

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.

Topics

Amazon API Gateway

Amazon 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.

AWS Lambda

Set your AWS Lambda 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 Amazon 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 if 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 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 the application SLA agreed to with other teams.

AWS Lambda Extensions count towards the deployment package size limit of your function. They also can impact the performance of your function because they share function resources such as CPU, memory, and storage. Account for the additional resources used when adding Lambda extensions through Lambda layers or functions deployed as container images. If your extension performs compute-intensive operations, you may see your function's execution duration increase.

Serverless applications may begin modeling monolithic applications, represented by a single AWS Lambda function performing multiple tasks. Serverless applications may adopt this monolithic approach as an easier way to get started, or developers may follow existing development practices and paradigms, or simple applications may grow more complex over time. As you optimize your serverless application, this monolithic approach may be less performant due to the bundle of dependencies for everything that is not used on every execution path. Consider breaking down your serverless application into microservices and remove unused dependencies from these discrete functions. You will also gain performance in adapting new features and opting for code optimized for the function use-case or integration.

Take advantage of Amazon API Gateway native routing functionality instead of using the routing of web frameworks, which are well suited for web servers. Web frameworks inside the Lambda function increases the size of the deployment package.

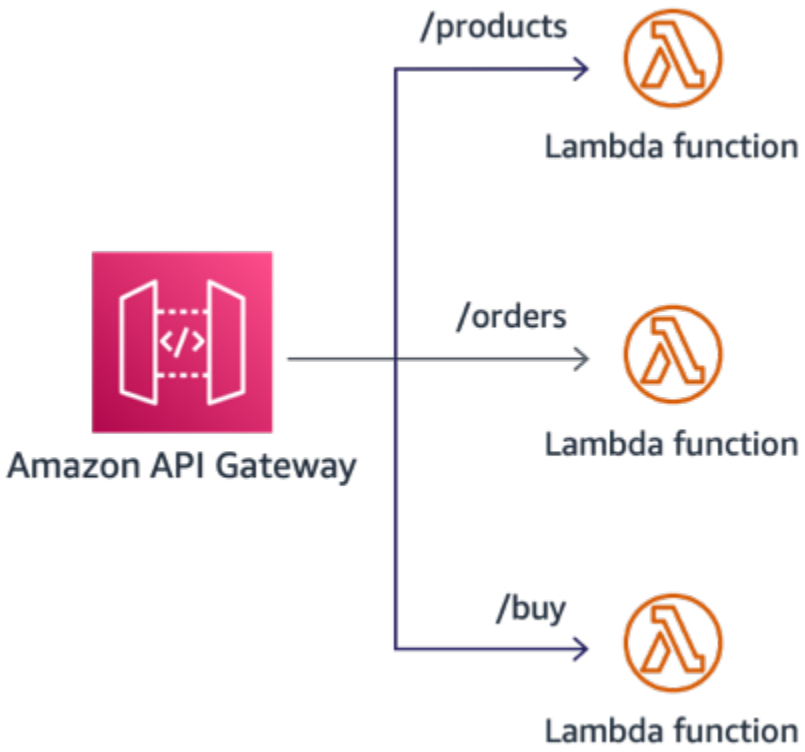


Figure 26: Amazon API Gateway simplified routing architecture

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.

Consider connection pooling with Amazon RDS Proxy for your Lambda functions that interact using SQL calls with your database instance. Amazon RDS Proxy handles the connection pooling necessary for scaling simultaneous connections created by concurrent AWS Lambda functions. This allows for reuse of existing connections, rather than creating new connections for every function invocation.

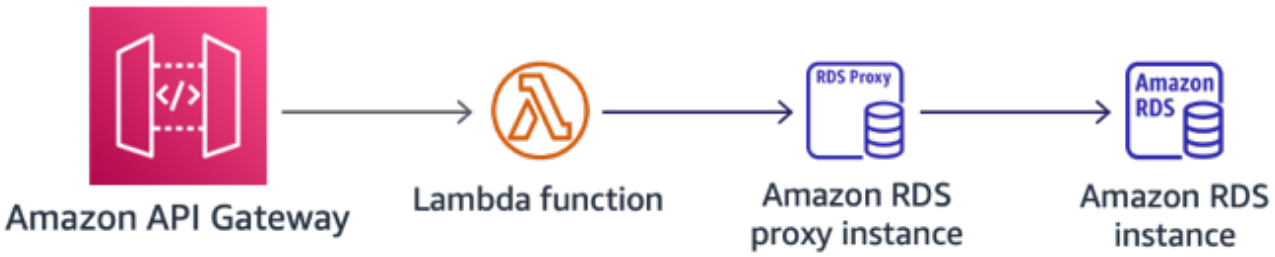


Figure 27: Amazon RDS Proxy allows you to efficiently scale to many more connections from your serverless application

AWS Step Functions

AWS Step Functions monitor the Amazon CloudWatch metric ExecutionThrottled which reports throttling on state transition, the number of `StateEntered` events, and retries that have been throttled. Use this metric to determine if a quota increase for a Standard Workflow is required.

If an Express Workflow execution runs for more than the 5-minute maximum, it will fail with a `States.Timeout` error and emit a `ExecutionsTimedOut` CloudWatch metric. Make use of timeouts in your task to avoid an execution stuck waiting for a response. Specify a reasonable `TimeoutSeconds` when you create the task. If you are receiving `States.Timeout` errors, consider breaking the workflow into multiple workflow executions, revising your task code or creating a Standard Workflow.

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 customer 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 or 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.

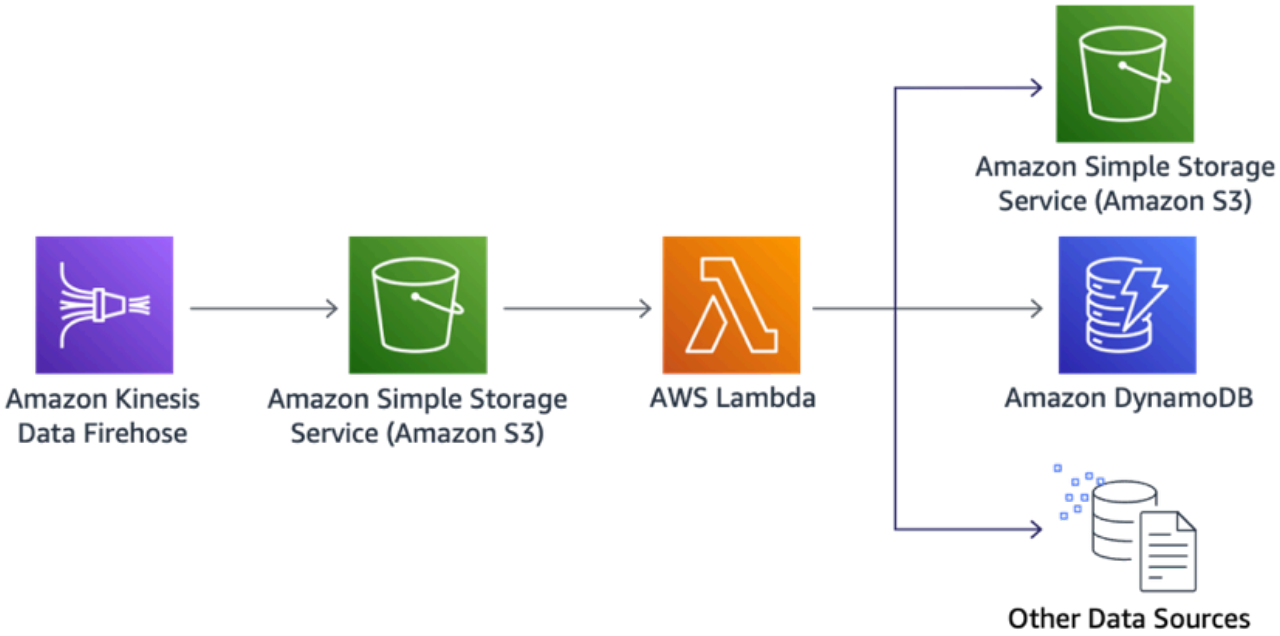


Figure 28: Asynchronous data ingestion

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 or System logs to CSV, JSON, JSON to Parquet, or ORC.

A Kinesis data stream is a set of shards, each shard contains a sequence of data records. Lambda reads records from the data stream and invokes your function synchronously with an event that contains stream records. Lambda reads records in batches and invokes your function to process records from the batch. Each batch contains records from a single shard or data stream.

To minimize latency and maximize read throughput of processing data from a Kinesis data stream, build your consumer with the enhanced fan-out feature. This throughput is dedicated, which means that consumers that use enhanced fan-out don't have to contend with other consumers that are receiving data from the stream.

Avoid invoking your function with a small number of records. You can configure the event source to buffer records for up to five minutes by configuring a CreateEventSourceMapping *batch window* (`MaximumBatchingWindowInSeconds`). Lambda continues to read records from the stream until it has gathered a full batch, or until the batch window expires.

Configure the CreateEventSourceMapping *batch size* (`BatchSize`) to control the maximum number of records that can be sent to your function with each invoke. A larger batch size can often more efficiently absorb the invoke overhead across a larger set of records, increasing your throughput. Avoid stalled shards by configuring the event source mapping to retry with a smaller batch size, limit the number of retries, or discard records that are too old. To retain discarded events, configure the event source mapping to send details about failed batches to an Amazon SQS queue or Amazon SNS topic.

Increase your Kinesis stream processing throughput using the CreateEventSourceMapping `ParallelizationFactor` setting to increase concurrency by processing multiple batches from each shard in parallel. Lambda can process up to 10 batches in each shard simultaneously keeping in-order processing at the partition-key level. Increase the number of shards to directly increase the number of maximum concurrent Lambda function invocations.

Use the Lambda emitted IteratorAge metric to estimate the latency between when a record is added and when the function processes it.

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 less than ideal 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).

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 29: Asynchronous workflow with Step Functions state machines

Another common scenario is integrating API Gateway directly with Amazon 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 30: Asynchronous workflow using a queue as a scaling layer

In this second example, Amazon 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 Standard Workflow 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 with 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 that 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. With AWS AppSync you power your application with the right data, from one or more data sources with a single network request using GraphQL. GraphQL works at the application layer and provides a type system for defining schemas. These schemas serve as specifications to define how operations should be performed on the data and how the data should be structured when retrieved.

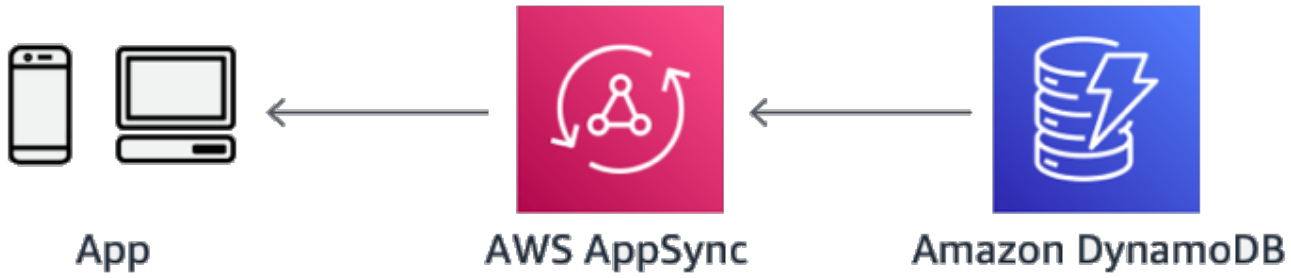


Figure 31: Asynchronous updates via WebSockets with AWS AppSync and GraphQL

Web Hooks can be implemented with Amazon SNS Topic HTTP subscriptions. Consumers can host an HTTP endpoint that Amazon SNS will call back through 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 32: Asynchronous notification via Webhook with Amazon 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 33: 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 showing 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

There are no performance efficiency practices unique to serverless applications for this best practice.

Monitoring

Monitor the Amazon CloudWatch AWS Lambda performance and concurrency metrics to understand performance details about a single invocation and the number of instances processing events across a function.

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

View the AWS Compute Optimizer identified recommendations for AWS Lambda function memory sizes. AWS Compute Optimizer uses machine learning to analyze historical utilization metrics.

Tradeoffs

Configuring AWS Lambda-provisioned concurrency incurs charges to your AWS account. Consider the functions you need to scale without fluctuations in latency. You can configure provisioned concurrency on a version of a function, or on an alias.

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 Amazon DynamoDB Accelerator, Amazon API Gateway, AWS Step Functions, Amazon VPC, NAT gateway and AWS Lambda.

Resources

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

Documentation and blogs

- Operating Lambda: Performance optimization – Part 1
- Operating Lambda: Performance optimization – Part 2
- Operating Lambda: Performance optimization – Part 3
- Using Amazon RDS Proxy with AWS Lambda
- Understanding Container Reuse in AWS Lambda
- New for AWS Lambda – Predictable start-up times with Provisioned Concurrency
- Introducing AWS Lambda Extensions
- Best practices for organizing larger serverless applications
- Caching data and configuration settings with AWS Lambda extensions
- Best Practices When Using Athena with AWS Glue
- Analyzing log data with CloudWatch Logs Insights
- Serverless Patterns Collection
- AWS Lambda Power Tuning
- Caching Best Practices

Developer guides

- What is AWS Lambda?
- Best practices for working with AWS Lambda functions
- Configuring a Lambda function to access resources in a VPC
- Using Lambda extensions - Impact on performance and resources
- Using AWS Lambda with Amazon SQS
- Managing concurrency for a Lambda function
- AWS Lambda FAQs
- Choosing between HTTP APIs and REST APIs
- Enabling API caching to enhance responsiveness
- Read/Write Capacity Mode
- Using Global Secondary Indexes in DynamoDB
- In-Memory Acceleration with DynamoDB Accelerator (DAX)
- Standard vs. Express Workflows
- Using AWS Step Functions with other services
- What Is Amazon Kinesis Data Streams?
- AppSync Data sources and resolvers
- Optimizing cold start performance for AWS Lambda

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.

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

- Cost-effective resources
- Matching supply and demand
- Expenditure and usage 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.

Cost-effective resources

COST 1: How do you optimize your costs?

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 initiation, the cheaper and more value your function produces due to 1-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 and usage 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, so 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 and usage 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. To gain practical experience on this topic refer to the Well Architected Labs. You can find many cost optimization walkthroughs that include tagging as well.

Optimizing over time

See the AWS Well-Architected Framework whitepaper for cost optimization best practices in the Optimizing over time section that apply to serverless applications.

Topics

- Lambda cost and performance optimization
- Logging ingestion and storage
- Leverage VPC endpoints
- DynamoDB on-demand and provisioned capacity
- AWS Step Functions Express Workflows
- Direct integrations
- Code optimization

Lambda cost and performance optimization

With Lambda, there are no servers to manage, it scales automatically, and you only pay for what you use. However, choosing the right memory size settings for a Lambda function is still an important task. AWS Compute Optimizer supports Lambda functions and uses machine-learning to provide memory size recommendations for Lambda functions.

This allows you to reduce costs and increase performance for your Lambda-based serverless workloads.

These recommendations are available through the Compute Optimizer console, AWS CLI, AWS SDK, and the Lambda console. Compute Optimizer continuously monitors Lambda functions, using historical performance metrics to improve recommendations over time.

In addition, consider configuring new and existing functions to run on ARM or Graviton processors. If your functions or dependencies do not require a given processor architecture (x86, ARM), you might benefit in cost and performance by switching your functions architecture. We always recommend load testing as results might vary for each use case, dependency, and runtime. For example, you could create two versions of your function: one for x86 and one for ARM. With the Alias feature, you could distribute a percentage of your traffic to a different processor architecture, and use CloudWatch Metrics to measure duration and latency efficiency.

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 the 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.

If you are using CloudWatch to record metrics in your Lambda Environment consider using the CloudWatch Embedded Metric Format (EMF) instead of using the CloudWatch PutMetricData API.

EMF enables you to ingest complex high-cardinality application data in the form of logs and easily generate actionable metrics from them. The embedded metric format is a JSON specification used

to instruct CloudWatch Logs to automatically extract metric values embedded in structured log events.

In such high-cardinality environments you might observe cost savings by having your Lambda functions leverage the CloudWatch Embedded Metric Format since with EMF you do not pay the per request charge of the CloudWatch PutMetricData API. With EMF you are only charged for Data Ingestion per GB, Data Archival per GB and per Custom Metric.

The metrics created with EMF are created asynchronously by the CloudWatch service. This means that by using EMF when processing logs might also reduce the execution duration of your Lambda functions compared to using the PutMetricData API which is a synchronous call.

If you need to have a precise timestamp for each individual metric or you have dimensions with the same key but different values, at the time of writing you will need to log separate EMF blobs. This means increased data ingestion and storage per GB CloudWatch cost.

In those cases we recommend to evaluate if the increased log ingestion and storage cost of EMF will be more expensive versus the benefit of not paying for the per request charge of the CloudWatch PutMetricData API. Moreover if you need high resolution metrics CloudWatch PutMetricData API might be a better fit versus EMF.

Leverage VPC endpoints

If you use Amazon Virtual Private Cloud (Amazon VPC) to host your AWS resources, you can establish a connection between your VPC and serverless services like AWS Lambda and AWS Step Functions. You can use this connection to invoke your Serverless resources without crossing the public internet.

To establish a private connection between your VPC and serverless resources, you can create an interface VPC endpoint. Interface endpoints are powered by AWS PrivateLink, which enables you to privately access APIs without needing an internet gateway or NAT device within your architecture.

Leveraging VPC endpoints will most likely contribute to cost savings if you are leveraging NAT and Internet gateways for the sole purpose of accessing Serverless APIs from AWS resources that do not have access to the internet. The cost optimisation is achieved from the fact that interface endpoints are more cost effective VPC structures compared to NAT and Internet gateways.

The example diagrams below show two different patterns of Lambda functions accessing the Amazon SNS service. In the first diagram, there are two NAT Gateways in two AZs for high availability and an Internet Gateway. In the second diagram, there are interface endpoints in two

AZs. The second pattern is more cost effective than the first one because interface endpoints are more cost effective than using NAT and Internet Gateways combined.

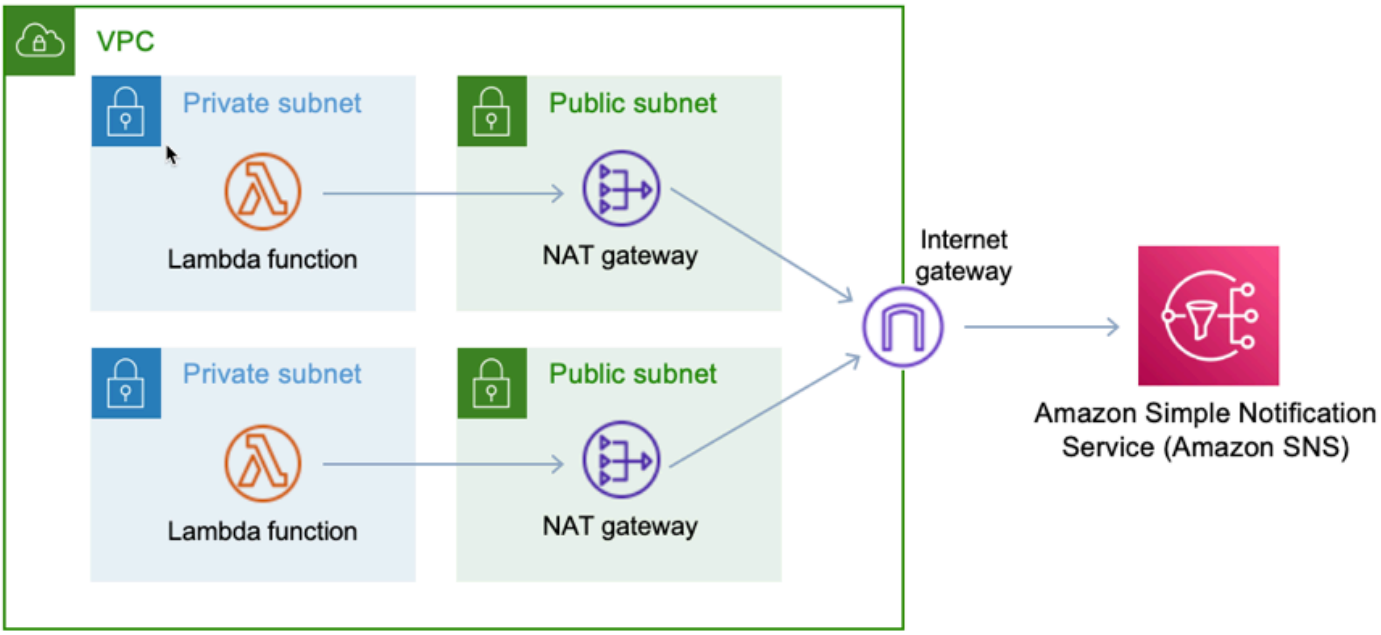


Figure 34: Lambda function accessing Amazon SNS via NAT and Internet Gateways

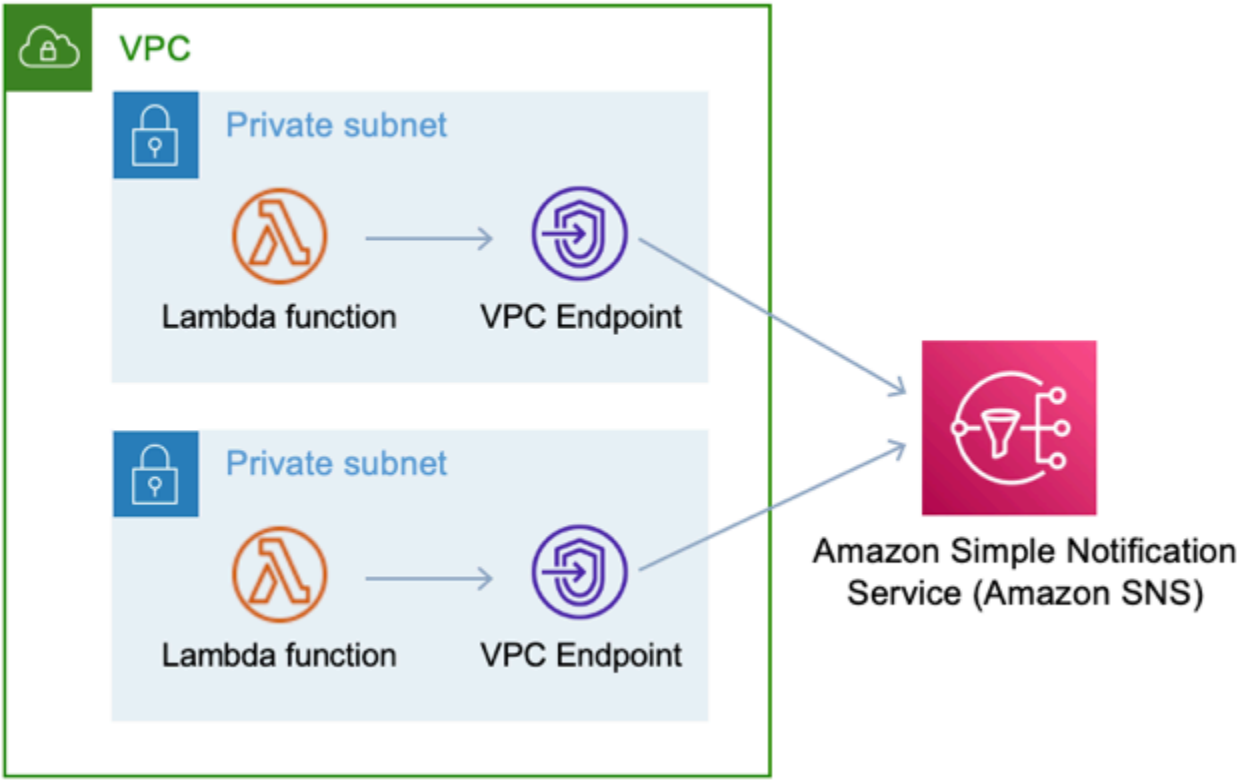


Figure 35: Lambda function accessing Amazon SNS via interface endpoints

DynamoDB on-demand and provisioned capacity

Amazon DynamoDB is a fully managed NoSQL database service with single-digit millisecond performance at any scale, and is often used for serverless applications. DynamoDB has two pricing models for read and write throughput: on-demand mode and provisioned mode.

On-demand mode

DynamoDB on-demand mode is a serverless throughput option that simplifies database management and automatically scales to support your most demanding applications. DynamoDB on-demand lets you create a table without worrying about capacity planning, monitoring usage, and configuring scaling policies. DynamoDB on-demand mode offers pay-per-request pricing for read and write requests so that you only pay for what you use. For on-demand mode tables, you don't need to specify how much read and write throughput you expect your application to perform.

On-demand mode is the default and recommended throughput option for most DynamoDB workloads. DynamoDB handles all aspects of throughput management and scaling to support workloads that can start small and scale to millions of requests per second. You can read and write to your DynamoDB tables as needed without managing throughput capacity on the table. For more information, see DynamoDB on-demand capacity mode.

Provisioned mode

In provisioned mode, you must specify the number of reads and writes per second that you require for your application. You'll be charged based on the hourly read and write capacity you have provisioned, not how much of that provisioned capacity you actually consumed. This helps you govern your DynamoDB use to stay at or below a defined request rate in order to obtain cost predictability.

You can choose to use provisioned capacity if you have steady workloads with predictable growth, and if you can reliably forecast capacity requirements for your application. For more information, see DynamoDB provisioned capacity mode.

AWS Step Functions Express Workflows

When you are creating a workflow with AWS Step Functions you will be given two options for the type of workflow that you want to create: Standard or Express.

Standard Workflows are ideal for long-running, durable, and auditable workflows. Standard Workflows can support an execution start rate of over 2K executions per second. They can run for

up to a year and you can retrieve the full execution history using the Step Functions API. Standard Workflows employ an exactly-once execution model, where your tasks and states are never started more than once unless you have specified the **Retry** behavior in your state machine. This makes them suited to orchestrating non-idempotent actions, such as starting an Amazon EMR cluster or processing payments. Standard Workflow executions are billed according to the number of state transitions processed.

Because the Standard Workflows pricing is based on state transitions, try to avoid the pattern of managing an asynchronous job by using a polling loop and prefer instead to use Callbacks or the .sync Service integration where possible. Using Callbacks or the .sync Service integration will most likely reduce the number of state transitions and cost. With the .sync Service integration in particular, you can have Step Functions wait for a request to complete before progressing to the next state. This is applicable for integrated services such as AWS Batch and Amazon ECS.

See diagrams below that describe each scenario:

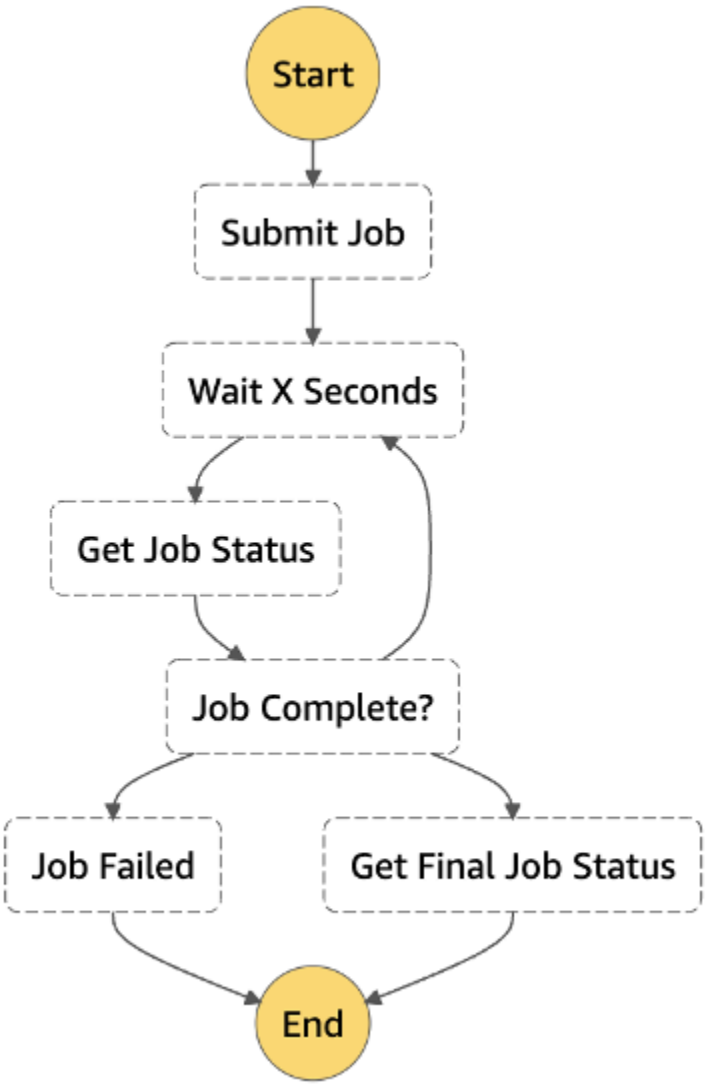


Figure 36: Job Poller

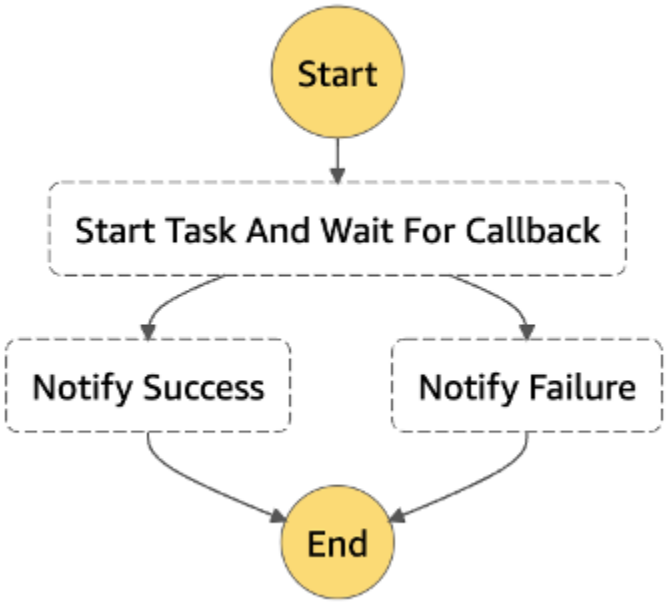


Figure 37: Wait for Callback

For example, pausing the workflow until a callback is received from an external service.

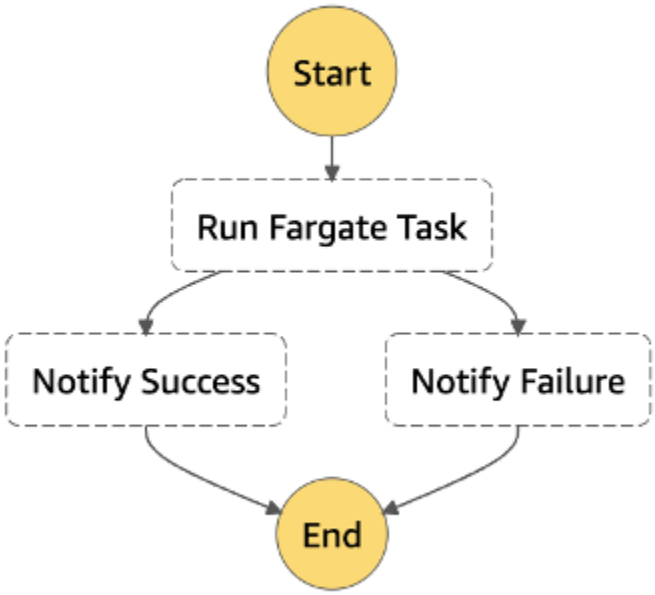


Figure 38: Using the .sync Service Integration and waiting for a Fargate task completion

Express Workflows are ideal for high-volume, event-processing workloads such as IoT data ingestion, streaming data processing and transformation, and mobile application backends. Express

Workflows can support an execution start rate of over 100K executions per second. They can run for up to five minutes. Express Workflows can run either synchronously or asynchronously and employ an at-most-once or at-least-once workflow execution model, respectively. This means that there is a possibility that an execution might be run more than once.

Ensure your Express Workflow state machine logic is idempotent and that it will not be affected adversely by multiple concurrent executions of the same input.

Good examples of using Express Workflows is orchestrating idempotent actions, such as transforming input data and storing with PUT in Amazon DynamoDB. Express Workflow executions are billed by the number of executions, the duration of execution, and the memory consumed. There are also cases where combining a Standard and an Express Workflow might offer a good combination of cost optimization and functionality. An example of a combining Standard and Express workflows is shown in the diagram below. More specifically, in the diagram below, the **Approve Order Request** state might be implemented by integrating with a service like Amazon SQS, and the workflow can be paused while waiting for a manual approval. This type of state would be good fit for a Standard Workflow. Whereas for the **Workflow to Update Backend Systems** state implementation you can start an execution of an Express Workflow to handle backend updates. Express Workflows can be fast and cost-effective for steps where checkpointing is not required.

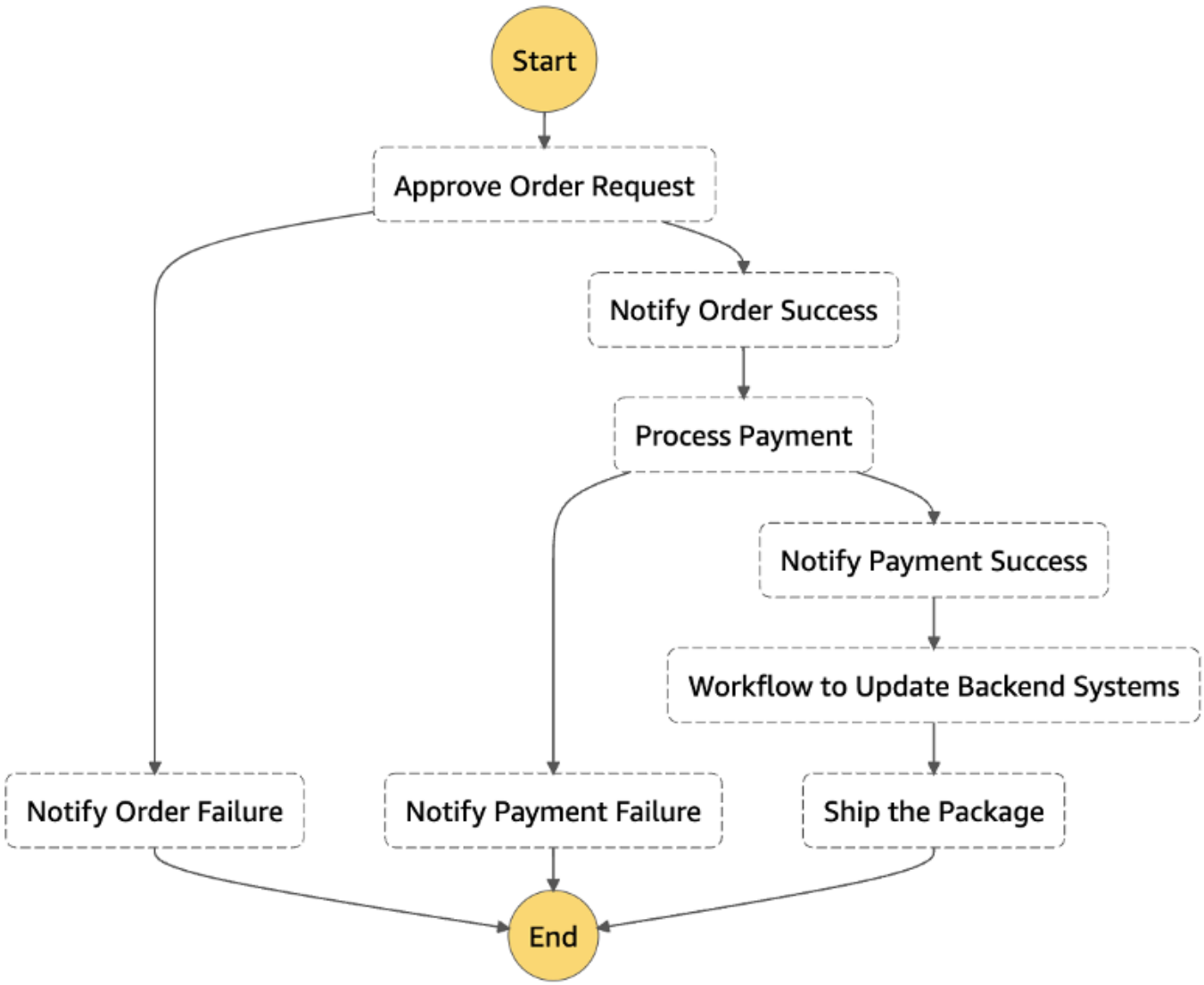


Figure 39: Express and Standard Workflows combined

In summary, deciding between Express and Standard Workflows largely depends your use case. Consider using Express Workflows for a high throughput system, as Express Workflows will probably be more cost-efficient compared to Standard Workflows for the same level of throughput. In order to be able to determine which type of workflow is best for you, consider the differences in execution semantics between Standard and Express Workflows on top of cost.

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

Figure 40: Sending data to Amazon S3 using Firehose

In this scenario, API Gateway will execute a Lambda function that will simply ingest the incoming record into Firehose, which 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 Firehose.

Figure 41: 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. A tradeoff when using the AWS Service Proxy is that a direct integration might introduce some extra complexity if multiple shards are necessary to meet the ingestion rate. In addition in the case that you need to transform the messages being sent to Firehose from API Gateway you will need to use mapping templates at the API Gateway layer. Adding mapping templates might introduce extra complexity on the debugging and testing side versus using a Lambda function instead. If latency-sensitive, you can stream data

directly to your Firehose by having the correct credentials at the expense of abstraction, contract, and API features.

Figure 42: Reducing cost of sending data to Amazon S3 by streaming directly using the 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.

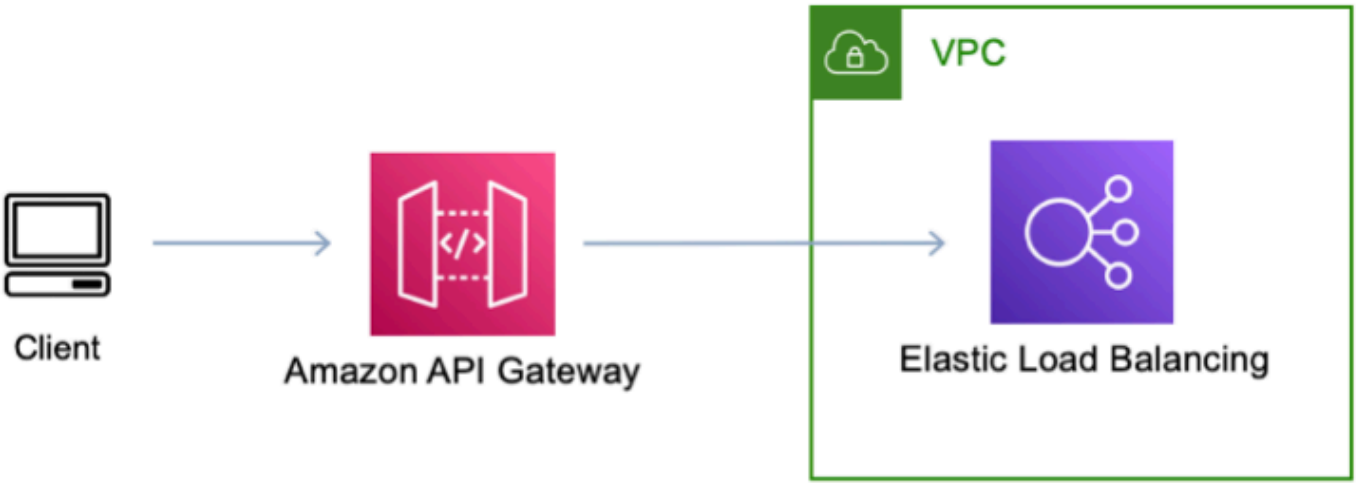


Figure 43: Amazon API Gateway private integration over Lambda in VPC to access private resources

With this approach, API Gateway sends each incoming request to an Elastic Load Balancer that you own in your VPC, which can forward the traffic to any backend, either in the same VPC or on-premises through an IP address. For REST APIs, Network Load Balancer is supported as a private integration. For HTTP APIs, both Application Load Balancer and Network Load Balancer are supported. 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.

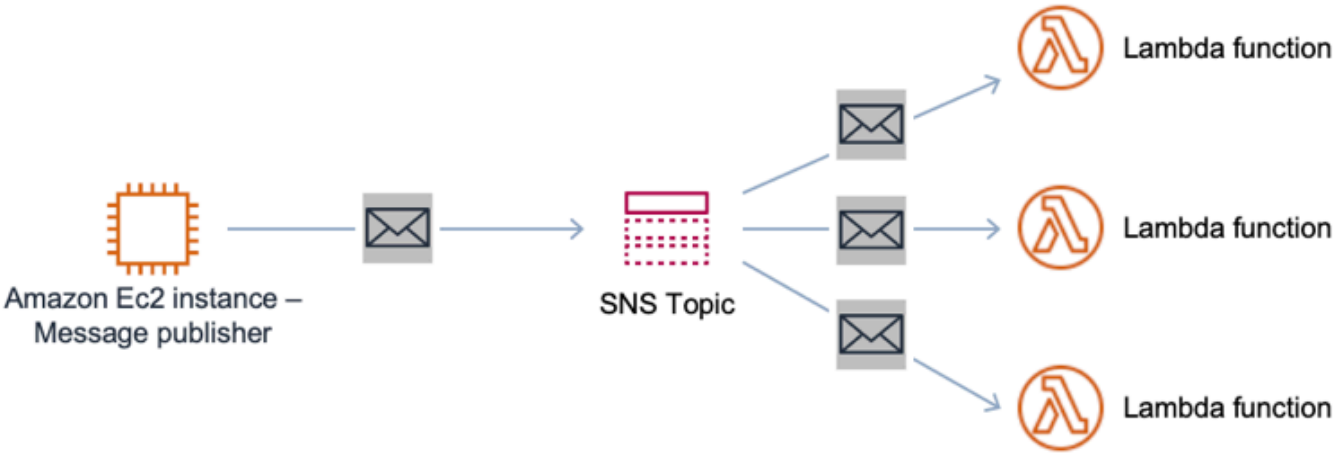


Figure 44: Amazon SNS without message attribute filtering

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

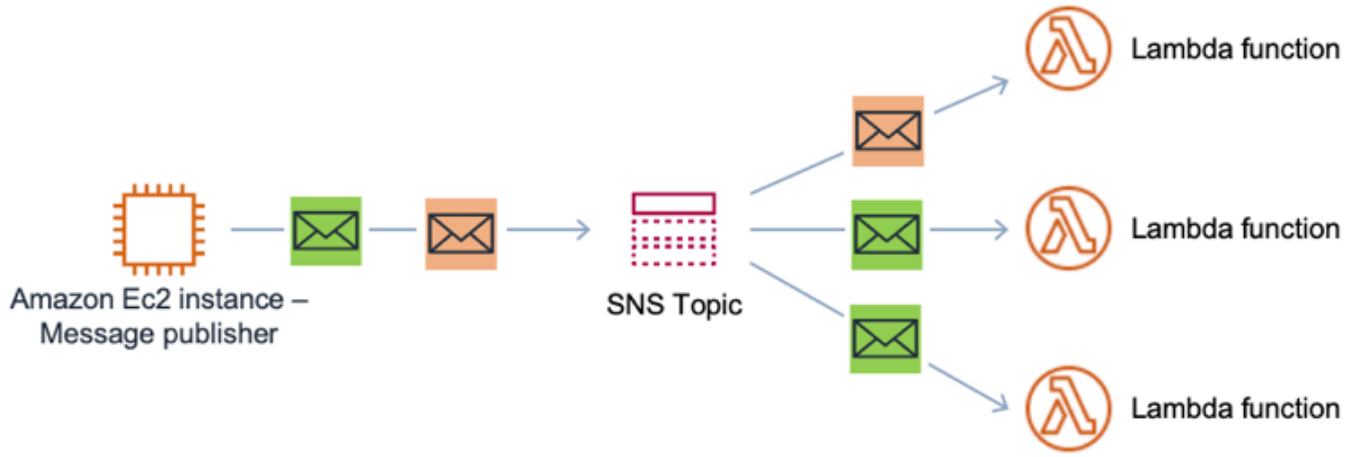


Figure 45: Amazon SNS with message attribute filtering

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. Moreover consider connection pooling with Amazon RDS Proxy for your Lambda functions that interact using SQL calls with your relational database instance. Please refer to the documentation for the database engines that are supported and also find more information, at the Serverless 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.

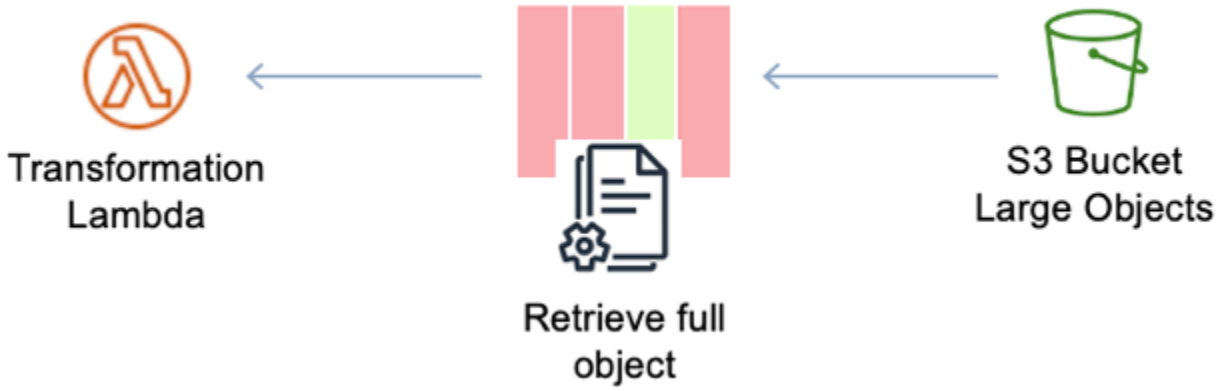


Figure 46: Lambda function retrieving full S3 object

The previous diagram shows that when retrieving large objects from Amazon S3, we might increase the memory consumption of the Lambda, increase the execution (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 upon which to perform transformations.

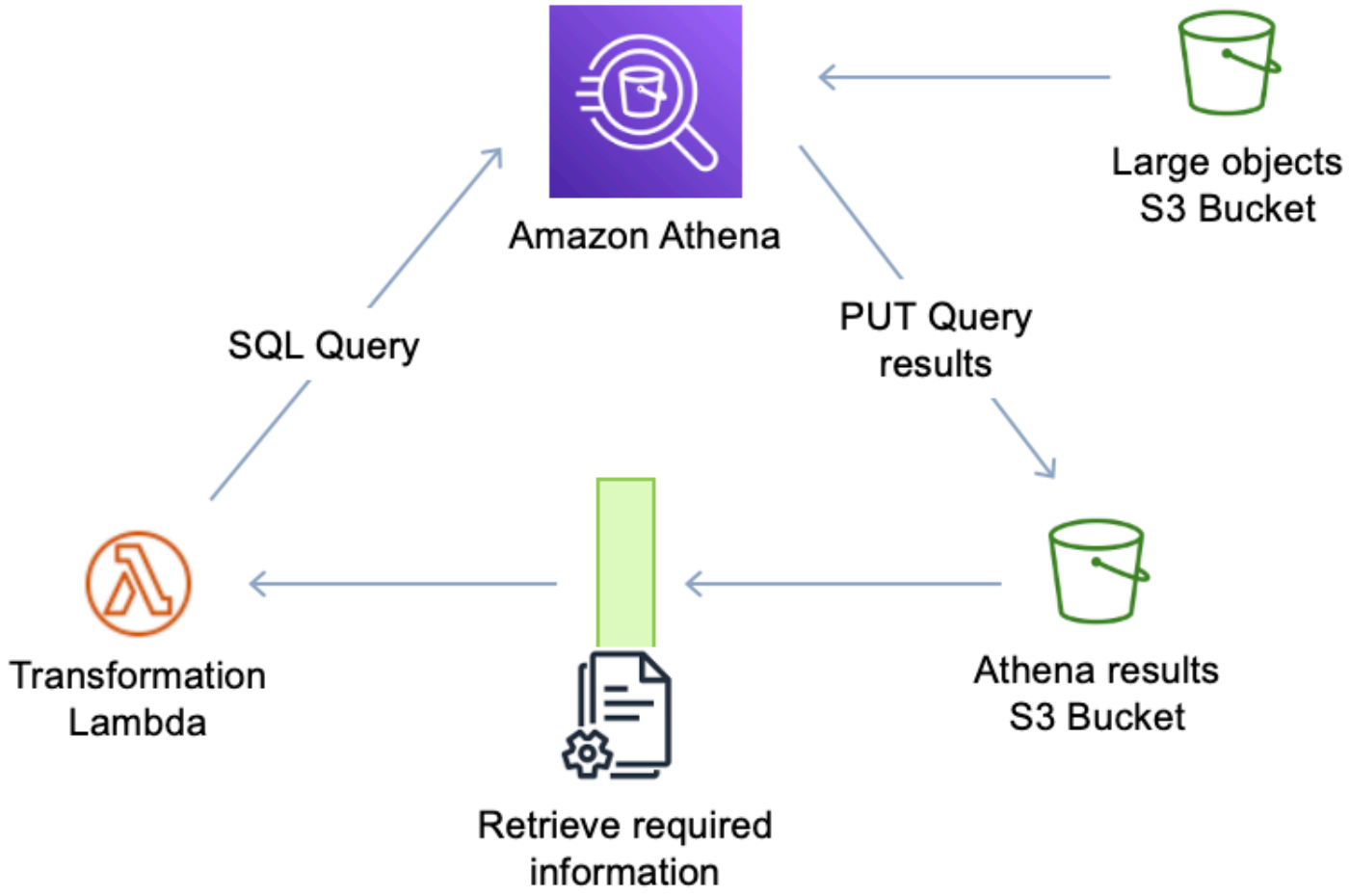


Figure 47: Lambda with Athena object retrieval

The next diagram shows 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 an S3 bucket and invokes the Lambda invocation as the results land in Amazon S3 asynchronously.

A similar approach could be using S3 Select, which 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.

Table: Lambda performance statistics using Amazon S3 vs S3 Select

200 seconds	*95 seconds*

`# Download and process all keys`
 	
`for key in src_keys:`
 	
`response = s3_client.get_obje ct(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]`
 	
…
	`# Select IP Address and Keys`
	
	`for key in src_keys:`
	
	`response = s3_client.select_o bject_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:`
	
	…

Resources

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

Documentation and blogs

- CloudWatch Logs Retention
- Exporting CloudWatch Logs to Amazon S3
- Streaming CloudWatch Logs to OpenSearch Service
- Defining wait states in Step Functions state machines
- Coca-Cola Vending Pass State Machine Powered by Step Functions
- Building high throughput genomics batch workflows on AWS

- 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

Whitepapers

- Optimizing Enterprise Economics with Serverless Architectures

Sustainability pillar

The sustainability pillar includes the ability to continually improve sustainability impacts by reducing energy consumption and increasing efficiency across all components of a workload by maximizing the benefits from the provisioned resources and minimizing the total resources required.

There are no sustainability practices unique to this lens. For information on Sustainability, refer to the Sustainability Pillar whitepaper.

Conclusion

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

For reliability, regular testing of failure paths provides you with a better chance of catching errors before they reach production. For performance, starting backward from customer expectations will allow you to design for optimal experience. There are a number of AWS tools to help optimize performance.

For cost optimization, you can reduce waste within your serverless application by right-sizing resources to support traffic demands, 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 serverless landscape continues to evolve with the growth and maturation of tooling, processes, and adoption. We will continue to update this paper to ensure that you have the resources and knowledge needed to build and operate well-architected serverless systems on AWS.

Contributors

The following individuals and organizations contributed to this document:

- Heitor Lessa, Principal Serverless Lead Well-Architected, Amazon Web Services
- Mark Bunch, Enterprise Solutions Architect, Amazon Web Services
- Dave Walker, Principal Specialist Solutions Architect, Amazon Web Services
- Richard Threlkeld, Sr. Product Manager Mobile, Amazon Web Services
- Roman Boiko, Sr. Serverless Solutions Architect, Amazon Web Services
- Leonidas Drakopoulos, Enterprise Solutions Architect, Amazon Web Services
- Karl Schween, Principal Solutions Architect, Amazon Web Services
- Brian Zambrano, Sr. Serverless Solutions Architect, Amazon Web Services
- Joe Mann, Sr. Partner Solutions Architect, Amazon Web Services
- Bruce Ross, Well-Architected Lens Leader, Amazon Web Services

Further reading

For additional information, see the following:

- AWS Well-Architected Framework
- AWS Architecture Center

Document revisions

To be notified about updates to this whitepaper, subscribe to the RSS feed.

Change	Description	Date
Major update	Updates throughout for new features and evolution of best practices.	July 14, 2022
Minor update	Updated links.	March 10, 2021
Minor update	Fixed formatting issue in HTML and minor editorial changes.	March 1, 2021
Minor update	Fixed missing figure in HTML and minor editorial changes.	July 15, 2020
Whitepaper updated	Updates throughout for new features and evolution of best practice.	December 19, 2019
Whitepaper updated	New scenarios for Alexa and Mobile, and updates throughout to reflect new features and evolution of best practice.	November 1, 2018
Initial publication	Serverless Applications Lens first published.	November 1, 2017

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.

AWS Glossary

For the latest AWS terminology, see the AWS glossary in the *AWS Glossary Reference*.