Serverless Applications Lens

AWS Well-Architected Framework

November 2017

Notices

This document is provided for informational purposes only. It represents AWS's current product offerings and practices as of the date of issue of this document, which are subject to change without notice. Customers are responsible for making their own independent assessment of the information in this document and any use of AWS's products or services, each of which is provided "as is" without warranty of any kind, whether express or implied. This document does not create any warranties, representations, contractual commitments, conditions or assurances from AWS, its affiliates, suppliers or licensors. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

Contents

Abstract

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

Introduction

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

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

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

Definitions

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

- Compute layer
- Data layer
- Messaging and streaming layer
- User management and identity layer
- Systems monitoring and deployment

- Edge layer

Compute Layer

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

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

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

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

Data Layer

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

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

With **Amazon Simple Storage Service** (Amazon S3)**,** you can build serverless web applications and websites by providing a highly available key-

value store, from which static assets can be served via a Content Delivery Network (CDN), such as **Amazon CloudFront**.

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

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 Analytics,** you can run standard SQL or build entire streaming applications using SQL.

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

User Management and Identity Layer

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

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

Systems Monitoring and Deployment

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

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

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

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

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.

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

General Design Principles

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

- **Speedy, simple, singular**: Functions are concise, short, single purpose and their environment may live up to their request lifecycle.

Transactions are efficiently cost aware and thus faster executions are preferred.

- **Think concurrent requests, not total requests**: Serverless applications take advantage of the concurrency model, and tradeoffs at the design level are evaluated based on concurrency.
- **Share nothing**: Function runtime environment and underlying infrastructure are short-lived, therefore local resources such as temporary storage are not guaranteed. State can be manipulated within a state machine execution lifecycle, and persistent storage is preferred for highly durable requirements.
- **Assume no hardware affinity**: Underlying infrastructure may change. Leverage code or dependencies that are hardware-agnostic as CPU flags, for example, may not be available consistently**.**
- **Orchestrate your application with state machines, not functions**: Chaining Lambda executions within the code to orchestrate the workflow of your application results in a monolithic and tightly coupled application. Instead, use a state machine to orchestrate transactions and communication flows.
- **Use events to trigger transactions**: Events such as writing a new Amazon S3 object or an update to a database allow for transaction execution in response to business functionalities. This asynchronous event behavior is often consumer agnostic and drives just-in-time processing to ensure lean service design.
- **Design for failures and duplicates**: Operations triggered from requests/events must be idempotent as failures can occur and a given request/event can be delivered more than once. Include appropriate retries for downstream calls.

Scenarios

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

RESTful Microservices

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

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

Characteristics:

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

Reference Architecture

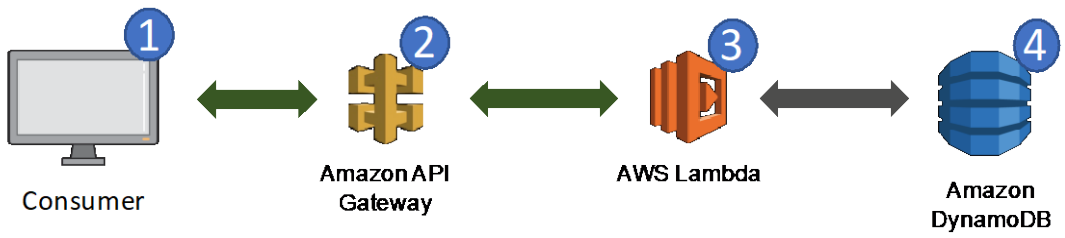

Figure 1: Reference architecture for RESTful microservices

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

Configuration notes:

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

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

Mobile Backend

Mobile application backend infrastructure requires scalability, elasticity, and low latency to support a dynamic user base. Unpredictable peak usage and a global footprint of mobile users requires mobile backends to be fast and flexible.

The growing demand from mobile users means applications need a rich set of mobile services that work together seamlessly without sacrificing control and flexibility of the backend infrastructure.

With AWS Lambda, you can build applications that automatically scale without provisioning or managing servers. Since many mobile applications today have a limited budget for upfront infrastructure, a cost-effective, event-driven mobile architecture allows you to pay only for what you use.

Characteristics:

- You want to create a complete serverless architecture without managing any instance and/or server.
- 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.

Reference Architecture

Here is a scenario for a common mobile backend, which doesn't take into consideration real-time/streaming use cases.

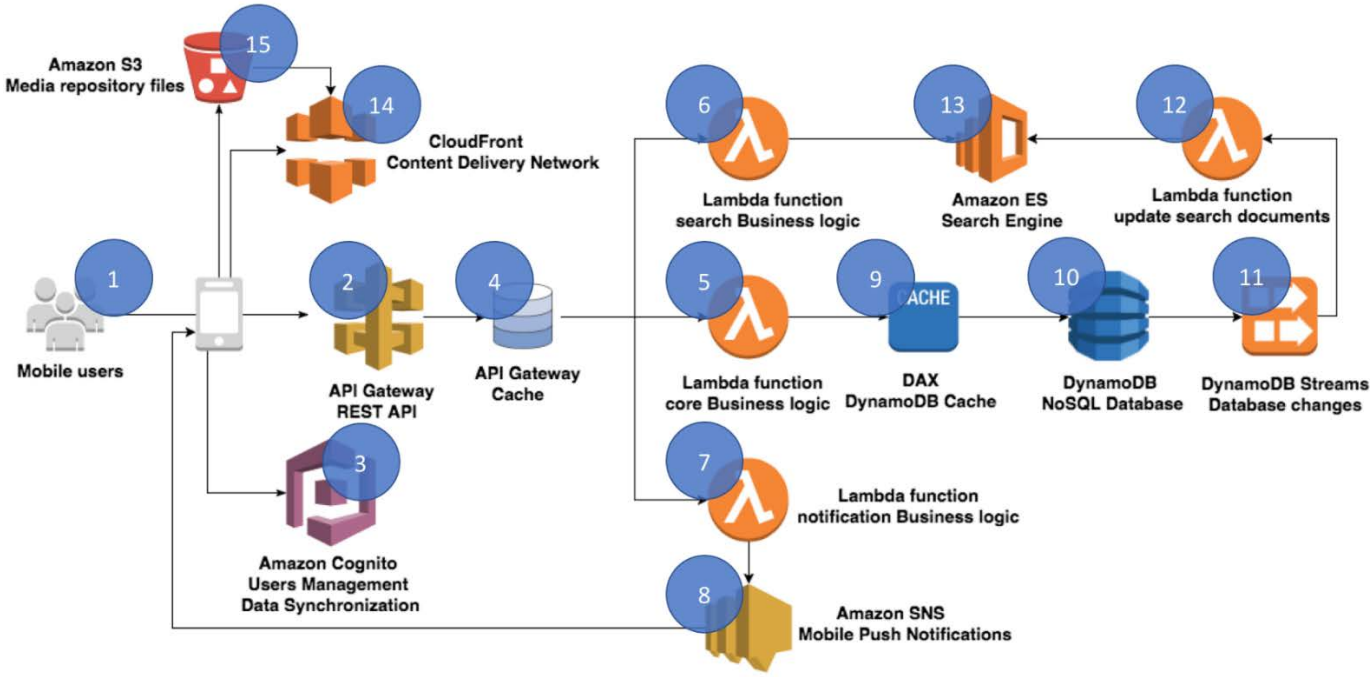

Figure 2: Reference architecture for a mobile backend

1. **Mobile users** interact with the mobile application backend by performing API calls against API Gateway and AWS service APIs (for example, Amazon S3 and Amazon Cognito).
2. **API Gateway** hosts RESTful HTTP requests and responses to mobile users. In this scenario, API Gateway provides the exact same feature set as described in the RESTful microservices scenario.
3. **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.
4. **API Gateway cache** is used to avoid unnecessary Lambda executions by caching the content or previous calls within the service and delivering it at a higher performance rate.
5. A **Lambda** function manages both POST and GET requests to either retrieve or update the data store.

6. A **Lambda** function handles search and queries to Amazon ES. These requests map with specific API methods and resources from the API Gateway.
7. A **Lambda** function implements the business logic of deciding when to send push notifications to the end user.
8. **Amazon SNS** delivers push notifications requested by the previous Lambda function.
9. **DAX** provides in-memory acceleration for DynamoDB tables, and it significantly improves the overall performance of this application including the heavy lifting of cache invalidation and cluster management.
10. **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.
11. **DynamoDB Streams** captures item-level changes and enables a Lambda function to update additional data sources.
12. A **Lambda** function acts as a consumer of DynamoDB Streams to update Amazon ES indexes allowing for analytics and additional richer queries such as full-text search for our mobile users.
13. **Amazon ES** acts as a main search engine for your mobile application as well as analytics.
14. **CloudFront** provides a CDN that serves content faster to geographically-distributed mobile users and includes security mechanisms to static assets in Amazon S3.
15. **Amazon S3** stores mobile application static assets including certain mobile user data such as profile images. Its contents are securely served via CloudFront.

Configuration notes:

- Validate request payloads by using the API Gateway request validation feature. Secure coding best practices such as input validation/sanitization still apply within your serverless application. This OWASP document[3] is a great starting point.

- Performance test[4] 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[5] when creating your DynamoDB tables. Make certain to calculate your read/write capacity and table partitioning to ensure reasonable response times.
- Follow best practices[6] when managing Amazon ES Domains. Additionally, Amazon ES provides an extensive guide[7] on designing concerning sharding and access patterns that also apply here.
- Reduce unnecessary Lambda function invocations by leveraging caching when possible at the API level.
- Leverage DAX for caching to significantly increase response times and reduce the number of calls to your DynamoDB table.
- For low-latency requirements where near-to-none business logic is required, Amazon Cognito Federated Identity can provide scoped credentials so that your mobile application can talk directly to an AWS service, for example, when uploading a user's profile picture, retrieve metadata files from Amazon S3 scoped to a user, etc.

Stream Processing

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

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

Characteristics:

- You want to create a complete serverless architecture without managing any instance or server for processing streaming data.

- You want to use the Amazon Kinesis Producer Library (KPL) to take care of data ingestion from a data producer-perspective.

Reference Architecture

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

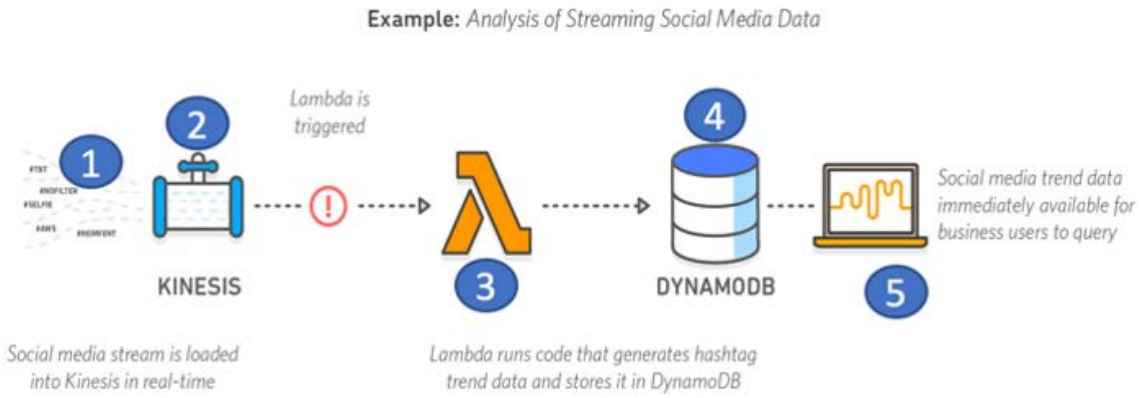

Figure 3: Reference architecture for stream processing

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

Configuration notes:

- Follow best practices[8] when re-sharding Kinesis streams in order to accommodate a higher ingestion rate. Concurrency for stream processing is dictated by the number of shards. Therefore, adjust it according to your throughput requirements.
- Consider reviewing the Streaming Data Solutions whitepaper[9] for batch processing, analytics on streams, and other useful patterns.
- When not using KPL, make certain to take into account partial failures for non-atomic operations such as PutRecords since the Kinesis API returns both successfully and unsuccessfully processed records[10] upon ingestion time.
- Duplicated records[11] may occur, and you must leverage both retries and idempotency within your application – both consumers and producers.
- Consider using Kinesis Firehose over Lambda when ingested data needs to be continuously loaded into Amazon S3, Amazon Redshift, or Amazon ES.
- Consider using Kinesis Analytics over Lambda when standard SQL could be used to query streaming data, and load only its results into Amazon S3, Amazon Redshift, Amazon ES, or Kinesis Streams.
- Follow best practices for AWS Lambda stream-based invocation[12] since that covers the effects on batch size, concurrency per shard, and monitoring stream processing in more detail.

Web Application

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

Using serverless computing on AWS, you can deploy your entire web application stack without performing the undifferentiated heavy lifting of

managing servers, guessing at provisioning capacity, or paying for idle resources. Additionally, you do not have to make any compromises on security, reliability, or performance.

Characteristics:

- You want a scalable web application that can go global in minutes with high levels of resiliency and availability.
- You want a consistent user experience with adequate response times.
- You are seeking to leverage managed services as much as possible for your platforms in order to limit the heavy lifting associated with managing common platforms.
- You want to optimize your costs based upon actual user demand versus paying for idle resources.
- You want to create a framework that is easy to set up and operate, and that you can extend with limited impact later.

Reference Architecture

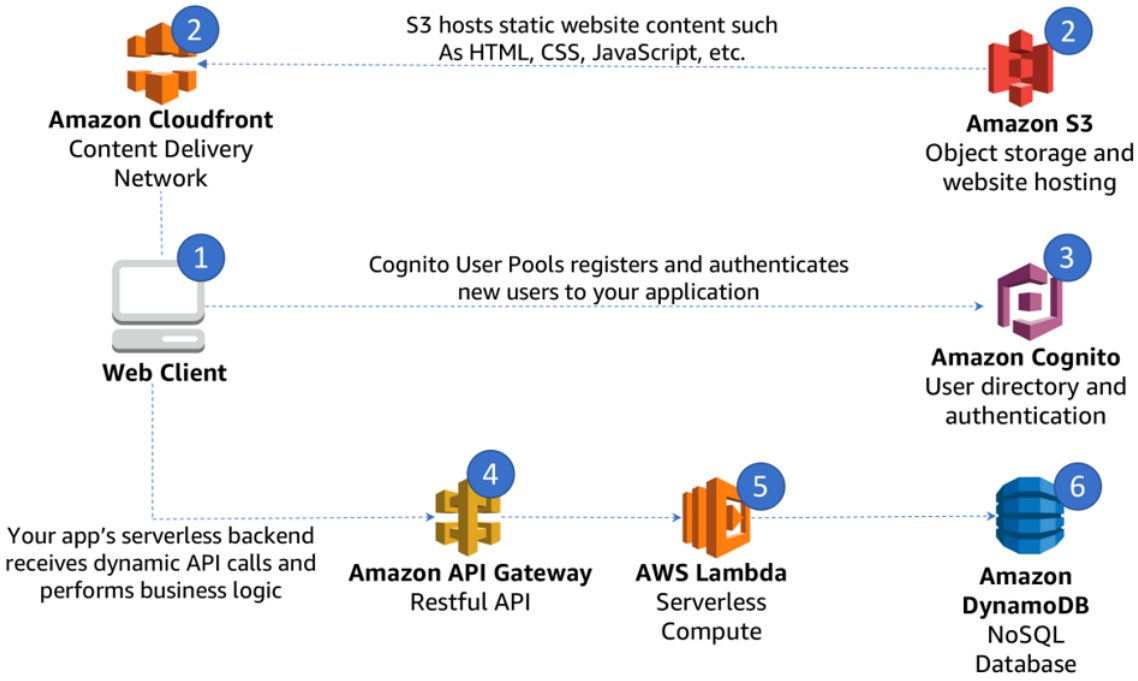

Figure 4: Reference architecture for a web application

1. **Consumers** of this web application may be geographically concentrated or worldwide. Leveraging Amazon CloudFront not only provides a better performance experience for these consumers through caching and optimal origin routing, but limits redundant calls to your backend.
2. **Amazon S3** hosts web application static assets and is securely served through CloudFront.
3. An **Amazon Cognito user pool** provides user management and identity provider feature for your web application.
4. As static content served by Amazon S3 is downloaded by the consumer, in many scenarios, dynamic content needs to be sent to or received by your application. For example, when a user submits data through a form, **Amazon API Gateway** serves as the secure endpoint to make these calls and return responses displayed through your web application.
5. An **AWS Lambda** function provides Create, Read, Update, Delete (CRUD) operations on top of DynamoDB for your web application.
6. **Amazon DynamoDB** can provide the backend NoSQL data store to elastically scale with the traffic of your web application.

Configuration Notes:

- Follow best practices for deploying your serverless web application frontend on AWS. More information on that can be found in the operational excellence pillar. Use Amazon S3 for hosting your static web content, and leverage CloudFront to securely deliver your content with low latency and high transfer speeds.
- For single-page web applications, make use of S3 object versioning, CloudFront cache expiration, and fine-tuned content TTL to reflect changes in deployments.
- Refer to the security pillar for recommendations for authentication and authorization.
- Refer to the RESTful Microservices scenario for recommendations on web application backend.
- For web applications that offer personalized services, you can leverage API Gateway usage plans[13] as well as Amazon Cognito user pools in order to scope what different sets of users have access to. For example, a

Premium user can have higher throughput for API calls, access to additional APIs, additional storage, etc.

- Refer to the Mobile Back End scenario if your application uses search capabilities that are not covered in this scenario.

The Pillars of the Well-Architected Framework

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

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

Operational Excellence Pillar

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

Definition

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

- Prepare
- Operate
- Evolve

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

Best Practices

Prepare

There are no operational practices unique to serverless applications that belong to this sub-section.

Operate

> SERVOPS 1: How are you monitoring and responding to anomalies in your serverless application?

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

Making changes to your serverless application entails many of the principles of deployment, change, and release management as 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.

Alarms should be configured at both individual and aggregated levels. Aggregate-level examples include alarming on the following metrics:

- **Lambda**: Throttling, Errors
- **Step Functions**: ActivitiesTimedOut, ActivitiesFailed, ActivitiesHeartbeatTimedOut
- **API Gateway**: 5XXError, 4XXError

An individual-level example is alarming on the *Duration* metric from Lambda and/or *IntegrationLatency* from API Gateway, since different parts of the application likely have different profiles. A bad deployment that makes a function execute for much longer than usual could be quickly captured in this instance.

Nevertheless, CloudWatch custom metrics that capture business as well as application insights still apply in serverless architectures.

> SERVOPS 2: How are you evolving your serverless application while minimizing the impact of change?

API Gateway stage variables help minimize the number of changes that need to be made to your API when releasing in the deployment lifecycle. For example, your stage variables can reference the name of your Lambda live alias, not $LATEST, and within the Swagger definition leverage stage variables over a hardcoded value.

Favor separate API Gateway endpoints, Lambda functions, and state machines for each stage over aliases and versions alone. Leverage Lambda versions and aliases to determine live from $LATEST. For example, a CI/CD pipeline Beta stage can create the following resources: OrderAPIBeta, OrderServiceBeta, OrderStateMachineWorkflowBeta, OrderBucketBeta, OrderTableBeta.

Use AWS SAM to package, deploy, and model serverless applications. SAM Local can also enable faster debugging cycles when developing Lambda functions locally. Additionally, there are a number of third-party serverless frameworks that can be used to package, deploy, and manage serverless solutions on AWS.

Lambda environment variables help separate source code from configuration. This can help you streamline deployments. For example, if a resource called within your Lambda function changes names, only the environment variables would need to change, not your code.

If you need more fine-grained control over configuration/secrets within serverless applications that you're sharing across multiple applications/functions, consider the Amazon EC2 Systems Manager (SSM) Parameter Store feature over environment variables. Parameter Store may incur additional latency, and so you should perform benchmarking when deciding to use SSM Parameter Store or environment variables.

Note: A/B Testing can be achieved by leveraging Lambda-weighted aliases. It's highly recommended to enable zero downtime changes.

SAM Local should not be used as a replacement for performance or regression testing since compute resources and network latency are substantially different in the AWS environment.

API Gateway stage variables and Lambda aliases/versions should not be used to separate stages as they can add additional operational and tooling complexity including reduced monitoring visibility as a unit of deployment.

It is not recommended to only rely on the CloudWatch metrics provided because X-Ray provides more insights into service metrics, such as AWS Lambda initialization and throttling across services used, that can be helpful when identifying and responding to anomalies.

Evolve

There are no operational practices unique to serverless applications that belong to this sub-section.

Key AWS Services

Key AWS services for operational excellence include Amazon EC2 Systems Manager Parameter Store, SAM, CloudWatch, AWS CodePipeline, AWS X-Ray, Lambda, and API Gateway.

Resources

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

Documentation & Blogs

- API Gateway stage variables[14]
- Lambda environment variables[15]
- SAM Local[16]
- X-Ray latency distribution[17]
- Troubleshooting Lambda-based applications with X-Ray[18]
- EC2 System Manager (SSM) Parameter Store[19]
- Continuous Deployment for Serverless applications blog post[20]

- SAM Farm: CI/CD example[21]

Whitepaper

- Practicing Continuous Integration/Continuous Delivery on AWS[22]

Third-Party Tools

- Serverless Developer Tools page including third-party frameworks/tools[23]

Security Pillar

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

Definition

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

- Identity and access management
- Detective controls
- Infrastructure protection
- Data protection
- Incident response

Serverless addresses some of today's biggest security concerns as it takes infrastructure management tasks away such as operating system patching, binaries, etc. Although the attack surface is reduced compared to non-serverless architectures, OWASP and application security best practices still apply.

The questions in this section are designed to help you address specific ways one attacker could try to gain access to or exploit misconfigured permissions that could lead to abuse. The practices described in this section strongly influence the security of your entire cloud platform and so should not only be validated carefully but also reviewed frequently.

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

Best Practices

Identity and Access Management

> SERVSEC 1: How do you authorize and authenticate access to your serverless API?

APIs are often targeted by attackers because of the valuable data they contain and operations that they can perform. There are various security best practices to defend against these attacks. From an authentication/authorization perspective, there are currently three mechanisms to authorize an API call within API Gateway:

- AWS_IAM authorization
- Amazon Cognito user pools
- API Gateway custom authorizer

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

Below is a diagram illustrating AWS_IAM authorization in this context:

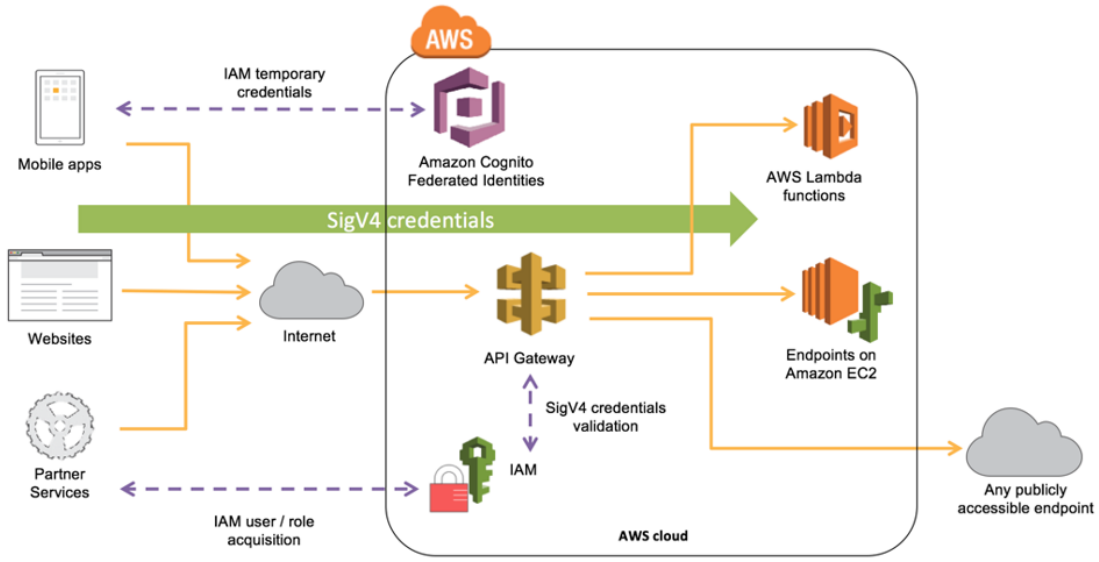

Figure 5: AWS_IAM authorization

For customers who currently have an existing Identity Provider (IdP), you can leverage an API Gateway custom authorizer to invoke a Lambda function to authenticate/validate a given user against your IdP. That is also commonly used when you want to perform additional logic on top of an existing IdP.

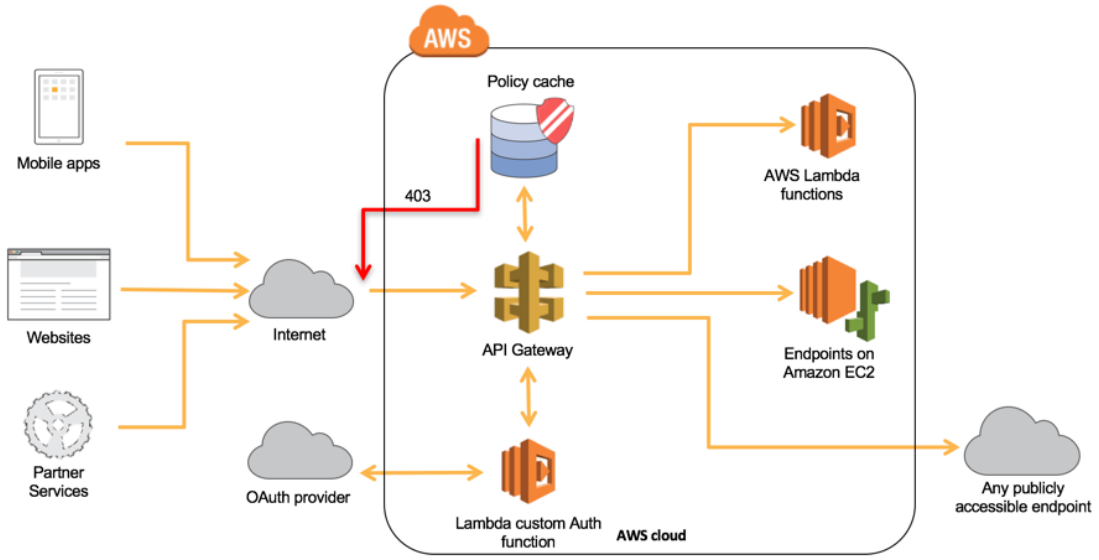

Figure 6: API Gateway custom authorizer

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

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

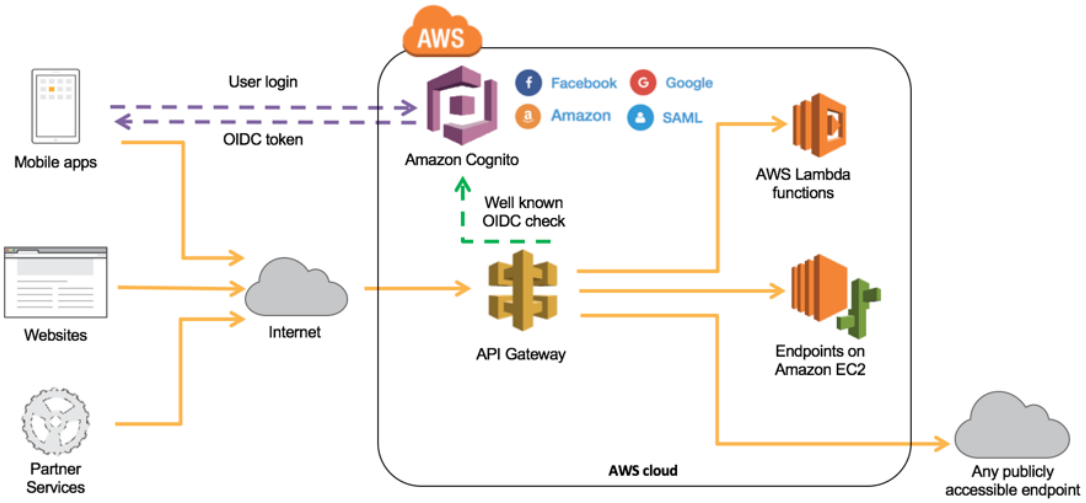

Figure 7: Amazon Cognito user pools

SERVSEC 2: How are you enforcing boundaries as to what AWS services your Lambda functions can access?

With regards to what a Lambda function can access, it is recommended to follow least-privileged access and strictly allow only what's necessary to perform a given operation. Attaching a role with more permissions than necessary can open your systems up for abuse.

Therefore, having smaller functions that perform scoped activities contribute to a more well-architected serverless application within the security context.

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

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

With respect to IAM roles, sharing an IAM role within more than one Lambda function will likely violate least-privileged access.

Detective Controls

> SERVSEC 3: How are you analyzing serverless application logs?

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

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

Leverage CloudWatch Logs metric filters to transform your serverless application standard output into custom metrics through regex pattern matching. In addition, create CloudWatch alarms based on your application custom metrics to quickly gain insight into how your application is behaving.

Similarly, AWS CloudTrail logs should also be used both for auditing and for API calls.

Consider enabling API Gateway logging for individual methods when troubleshooting as opposed to an entire stage because enabling logging at the stage level will record all API activities. Depending on your serverless application design it may contain sensitive data.

For this reason, we recommend that you encrypt any sensitive data traversing your serverless application. For more information on that, see the Data Protection sub-section.

Lambda functions are designed to do one task and complete it as fast as possible. Therefore, making API calls to CloudWatch within your code may cause the observer effect and excessive printing statements may ingest

unnecessary data into your logs. This will cause both an increase in signal-to-noise ratio as well as CloudWatch Logs ingestion charges.

API Gateway may log entire request/response payloads and that may violate your compliance requirements. Make certain to verify with your compliance team before enabling logging.

> SERVSEC 4: How do you monitor dependency vulnerabilities within your serverless application?

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

Infrastructure Protection

Although there is no infrastructure to manage in serverless applications, there could be scenarios where your serverless application needs to interact with other components deployed in a virtual private cloud (VPC) or applications residing on-premises. Consequently, it is important to ensure networking boundaries are considered under this assumption.

> SERVSEC 5: For VPC access, how are you enforcing networking boundaries as to what AWS Lambda functions can access?

Lambda functions can be configured to access resources within a VPC including resources sitting outside of AWS through VPN connections. You should review security group best practices as covered in the AWS Well-Architected Framework.

Also, similar to a non-serverless application, a security group and Network Access Control Lists (NACLs) should be the basis on which networking boundaries are enforced. For workloads that require outbound traffic filtering due to compliance reasons, proxies can be used in the exact same manner that they are placed in non-serverless architectures.

Enforcing networking boundaries solely at code level given instructions as to what resource one could access is not recommended due to separation of concerns.

Data Protection

> SERVSEC 6: How are you protecting sensitive data within your serverless application?

API Gateway employs the use of TLS across all communications, client and integrations. Although HTTP payloads are encrypted in-transit, request path and query strings that are part of a URL might not be. Also, encrypted HTTP payloads may be unencrypted when received from API Gateway, AWS Lambda in this case. 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 to either gain access to a system or cause it to 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 prior to any processing or data manipulation. This will prevent data leakage in the event that such data gets exposed in a persistent storage chosen or via standard output that is streamed and persisted by CloudWatch Logs. In the scenarios described earlier in this document, Lambda functions would persist encrypted data in either DynamoDB, Amazon ES, or Amazon S3 along with encryption at rest.

We strictly advise against sending, logging, and storing unencrypted sensitive data, be it part of an HTTP request path/query strings or 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 sub-section, you should consult about such an operation with your compliance team before enabling API Gateway logging.

> SERVSEC 7: What is your strategy on input validation?

For input validation, make sure to set up API Gateway 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 parameter 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.

Key AWS Services

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

Resources

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

Documentation & Blogs

- IAM role for Lambda function with Amazon S3 example[24]
- API Gateway Request Validation[25]
- API Gateway Custom Authorizers[26]
- Securing API Access with Amazon Cognito Federated Identities, Amazon Cognito User Pools, and Amazon API Gateway[27]
- Configuring VPC Access for AWS Lambda[28]
- Filtering VPC outbound traffic with Squid Proxies[29]

Whitepapers

- OWASP Secure Coding Best Practices[30]

- AWS Security Best Practices[31]

Partner Solutions

- Twistlock Serverless Security[32]

Third-Party Tools

- OWASP Vulnerability Dependency Check[33]
- Snyk – Commercial Vulnerability DB and Dependency Check[34]

Reliability Pillar

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

Definition

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

- Foundations
- Change management
- Failure management

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

Best Practices

Foundations

> SERVREL 1: Have you considered serverless limits for peak workloads?

AWS enforces default service limits to protect customers for unauthorized use of services. Limits that are not properly monitored may result in a degradation or

throttling of service. Many limits are soft limits and can be raised if you anticipate exceeding them.

You can use AWS Trusted Advisor within the AWS Management Console and APIs to detect whether a service exceeds 80% of a limit.[35]

You can also proactively raise limits if you anticipate exceeding them for workloads. When raising these limits, ensure there is a sufficient gap between your service limit and your max usage to accommodate scale or absorb a denial of service attack. You should consider these limits across all relevant accounts and Regions.

Additionally, the Lambda concurrent execution limit feature should be explicitly used to isolate non-business-critical to business-critical paths so that an unexpected event load doesn't take the remaining concurrency available within the rest of the resources competing for them. It is also a recommended practice to separate workloads in different accounts depending on their profile, threat model, and organizational structure.

Regardless of whether your serverless application is spiky or not, following asynchronous patterns when designing communications between services and transactions makes for a more resilient serverless application.

SERVREL 2: How are you regulating access rates to and within your serverless application?

Throttling

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

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

Throttling should be enabled at the API level to enforce access patterns established by a service contract. Defining a request access pattern strategy is fundamental to establish how a consumer should use a service be that at the resource level 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 access patterns in unexpected behavior. API keys also simplifies the process for administrators to cut off access if an individual consumer is making suspicious requests.

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

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

SERVREL 3: What is your strategy on asynchronous calls and events within your serverless architecture?

Asynchronous Calls and Events

Asynchronous calls reduce the latency on HTTP responses. Multiple synchronous calls, as well as long-running wait cycles, may result in timeouts and “locked” code that prevents retry logic. Event-driven architectures enable streamlining asynchronous executions of code, thus limiting consumer wait cycles.

Event-driven architectures that are commonly implemented within serverless applications are asynchronous. State machines, queues, pub/sub, WebHooks, events, and other techniques are commonly applied across multiple components that perform a business functionality.

User experience is decoupled with asynchronous calls. Instead of blocking the entire experience until the overall execution is completed, frontend systems receive a reference/job ID as part of their initial request and an additional API is used 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.

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

Alternatively, when synchronous calls are necessary, it is recommended at minimum to ensure the entire execution doesn't exceed API Gateway max timeout and that coordination is done by an external service (for example, AWS Step Functions) to control both state and exceptions that can occur along the request lifecycle.

SERVREL 4: What's your testing strategy for serverless applications?

Testing

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

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

Integration tests shouldn't mock services you can't control, since they may change and may provide unexpected results. These tests are better performed when using real services because they can provide the exact 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 end users' actions through the external interface available to them. Therefore, there is no unique recommendation to be aware of here.

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

Here are some considerations during performance testing to be aware of:

- Metrics such as invoked max memory are available in CloudWatch Logs. The metrics may indicate optimal memory and proper timeout value. For more information, read the performance pillar section.
- If your Lambda function runs inside a VPC, pay attention to available IP address space inside your subnet. For more information, read the operational excellence pillar section.
- Creating modularized code into separate functions outside of the handler enables more unit-testable functions.
- Establishing externalized connection code (such as a connection pool to a relational database) referenced in the Lambda function's static constructor/initialization code (that is, global scope, outside the handler) will ensure that external connection thresholds aren't reached if the Lambda execution environment is reused.
- Adjust the throughput of the DynamoDB read and write tables accordingly, and make sure to set up Auto Scaling to accommodate throughput changes throughout the performance testing cycle.
- Take into account any other service limits not listed here that may be used within your serverless application under performance testing.

SERVREL 5: How are you building resiliency into your serverless application?

Change Management

Having the ability to revert back to a previous version in the event of a failed change ensures service availability.

First, you need to put monitoring metrics in place. Determine what your environment workload's "normal" state is and define the appropriate metrics and threshold parameters in CloudWatch to determine what is "not normal" based on historical data.

Also monitor deployments and implement automated actions. Features such as Lambda function versioning and API versions can help you increase the cutover time back to a previous state if a deployment fails.

Failure Management

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

For Lambda functions, build retry logic into your Lambda queries to ensure that spiky workloads don't overwhelm your backend. Also, leverage the Lambda logging library within function code to add errors to CloudWatch Logs, which can be captured as a custom metric. For more information, read the operational excellence pillar section.

AWS SDKs provide back-off and retry mechanisms by default when talking to other AWS services that are sufficient in most cases. However, they should be reviewed and possibly tuned in order to suit your needs.

AWS X-Ray and third-party Application Performance Monitoring (APM) solutions can help enable distributed tracing to identify throttling, and how distribution latency is affected when they occur.

For asynchronous calls that may fail, it is a best practice to enable Dead Letter Queues (DLQ) and create dedicated DLQ resources (using Amazon SNS and Amazon Simple Queue Service (Amazon SQS)) for individual Lambda functions.

You also want to develop a plan to poll by a separate mechanism to re-drive these failed events back to their intended service.

Whenever possible, Step Functions should be used to minimize the amount of custom try/catch, back-off, and retries within your serverless applications. For more information, read the cost optimization pillar section.

Moreover, non-atomic operations such as PutRecords (Kinesis) and BatchWriteItem (DynamoDB) can return successful in the event of a partial failure. Therefore, the response should be inspected at all times when using such operations and programmatically dealt with.

For synchronous parts that are transaction-based and depend on certain guarantees and requirements, rolling back failed transactions as described by the Saga pattern[36] can also be achieved by using Step Functions state machines, which will decouple and simplify the logic of your application.

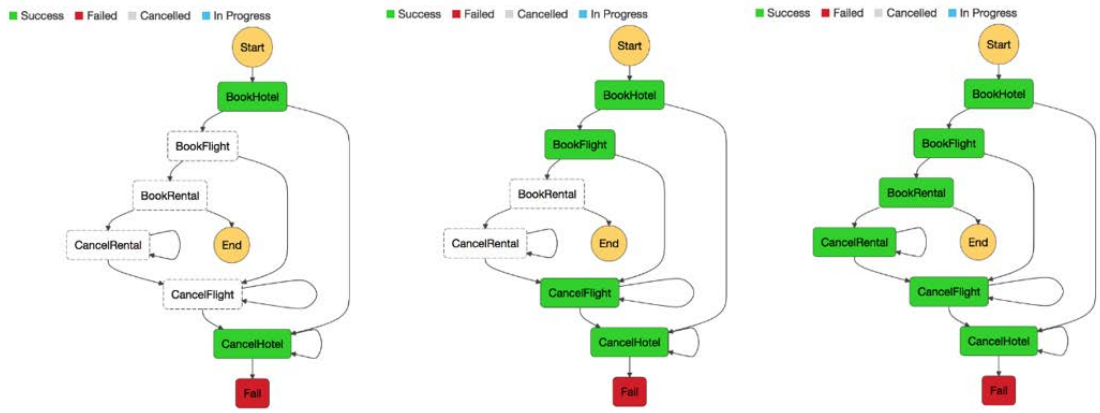

Figure 8: Saga pattern in Step Functions by Yan Cui

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 security.

Documentation & Blogs

- Limits in Lambda[37]
- Limits in API Gateway[38]
- Limits in Kinesis Streams[39]
- Limits in DynamoDB[40]
- Limits in Step Functions[41]
- Error handling patterns[42]
- Serverless testing with Lambda[43]
- Monitoring Lambda Functions Logs[44]
- Versioning Lambda[45]
- Stages in API Gateway[46]
- API Retries in AWS[47]
- Step Functions error handling[48]
- X-Ray[49]
- Lambda DLQ[50]
- Error handling patterns with API Gateway and Lambda[51]
- Step Functions Wait state[52]
- Saga pattern[53]
- Applying Saga pattern via Step Functions[54]

Whitepapers

- Microservices on AWS[55]

Performance Efficiency Pillar

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

Definition

Performance efficiency in the cloud is composed of four areas:

- Selection
- Review
- Monitoring
- Tradeoffs

Take a data-driven approach to selecting a high-performance architecture. Gather data on all aspects of the architecture, from the high-level design to the selection and configuration of resource types. By reviewing your choices on a cyclical basis, you will ensure that you are taking advantage of the continually evolving AWS Cloud. Monitoring will ensure that you are aware of any deviance from expected performance and can take action on it. Finally, you can make tradeoffs in your architecture to improve performance, such as using compression or caching, or relaxing consistency requirements.

Selection

In the AWS Lambda resource model, you choose the amount of memory you want for your function and are allocated proportional CPU power and other resources such as networking and storage IOPS. For example, choosing 256 MB of memory allocates approximately twice as much CPU power to your Lambda function as requesting 128 MB of memory and half as much CPU power as choosing 512 MB of memory.

In the Kinesis resource model, you choose how many shards you may need based on ingestion and consumption rate (reads, writes, data size). In the DynamoDB resource model, you choose how many reads and writes per second you may need based on your requirements.

Make sure to run performance testing on your Lambda functions prior to deciding on memory and timeout settings for your serverless application. Fine-

tuning memory and timeout will have a significant impact on performance, cost, and operational procedures.

It is recommended to set 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 and tasks.

Along with performance testing and data requirements, Kinesis and DynamoDB capacity units will be more closely aligned with your workload profile.

> SERVPER 1: How do you choose the most optimum capacity units (memory, shards, reads/writes per second) within your serverless application?

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

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

> SERVPER 2: How have you optimized the performance of your serverless application?

Optimize

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

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

Also described in Mobile Backend scenario, it is recommended to test the performance of your Lambda functions by using accurately sized sample workflows and varying the memory settings and timeout values.

Leverage global scope within your Lambda function code to take advantage of Lambda container reuse. With that, database connection and AWS services initial connection and configuration will be executed once if the environment in which that Lambda function was executed is still available.

Deployment

Lambda functions don't always need to be deployed in a VPC. Similarly, CPU and network bandwidth are proportionally allocated based on memory settings configured for a Lambda function.

Configure VPC access to your Lambda functions only when necessary, as deploying your function in a VPC will result in additional startup time for your Lambda function since Elastic Network Interfaces (ENI) must be created beforehand.

If your Lambda function needs access to the VPC and to the internet, you'll need a NAT gateway in order to allow traffic from Lambda to any resource available publicly on the internet. We recommend that you place a NAT gateway across multiple Availability Zones for high availability and performance.

SERVPER 3: How do you decide what components of your serverless application should be deployed in a VPC?

The decision tree below can help you decide whether to deploy your Lambda function in a VPC.

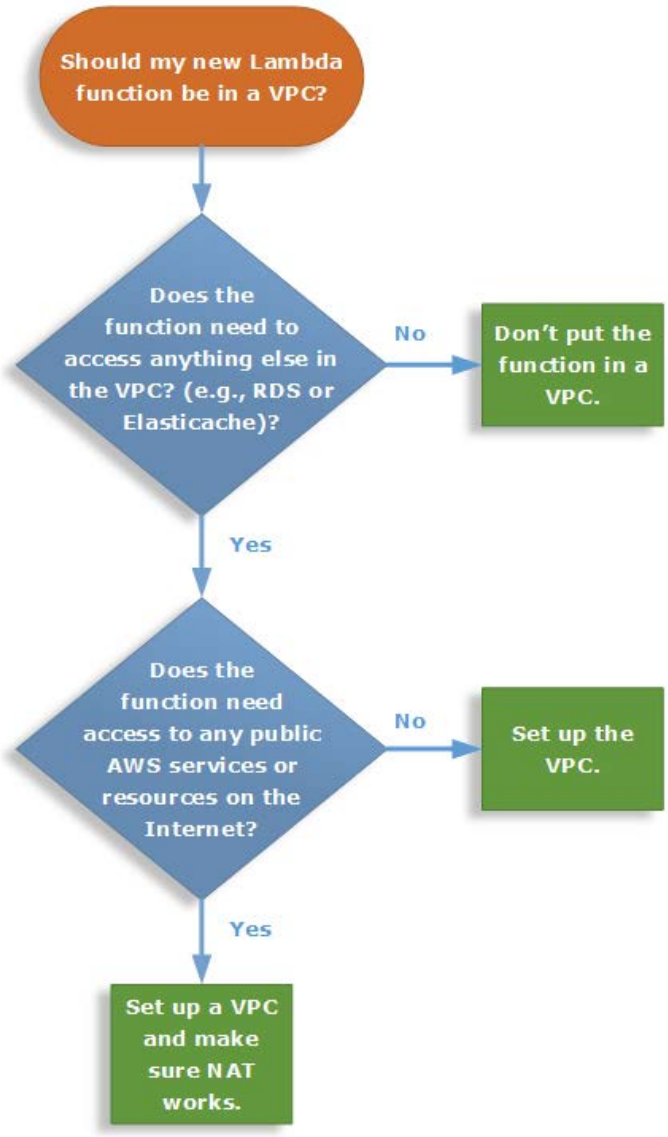

Figure 9: Decision tree for deploying a Lambda function in a VPC

SERVPER 4: How are you optimizing your Lambda code for performance?

Code Optimization

Lambda functions are single purpose and execute as fast as possible. However, there are techniques that can be leveraged to take advantage of the execution environment, as well as general design principles, since existing application dependencies/frameworks in non-serverless applications may not always be the best choice in this context.

On AWS, follow best practices for working with Lambda functions[56] such as container re-use, minimizing deployment package size to its runtime necessities, and minimizing the complexity of your dependencies. Tradeoff here is key when making this decision. 99th percentile (P99) should be always taken into account, as one may not impact application SLA agreed with other teams.

For Lambda functions in VPC, avoid DNS resolution of public host names for your VPC. This may take several seconds to resolve, which adds several seconds of billable time on your request. For example, if your Lambda function accesses an Amazon RDS DB instance in your VPC, launch the instance with the no-publicly-accessible option.

SERVPER 5: How are you initializing database connections?

After a Lambda function has executed, AWS Lambda maintains the runtime container for some time in anticipation of another Lambda function invocation.

Leverage global scope as described previously in the Optimize subsection. For example, if your Lambda function established a database connection, instead of reestablishing the connection, the original connection is used in subsequent invocations. Declare database connections and other objects/variables outside the Lambda function handler code to provide additional optimization when the function is invoked again. You can add logic in your code to check if a connection already exists before creating one.

Review

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

Monitoring

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

Tradeoffs

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

Key AWS Services

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

Resources

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

Documentation & Blogs

- AWS Lambda FAQs[57]
- Best Practices for Working with AWS Lambda Functions[58]
- AWS Lambda: How It Works[59]
- Understanding Container Reuse in AWS Lambda[60]
- Configuring a Lambda Function to Access Resources in an Amazon VPC[61]
- Enable API Caching to Enhance Responsiveness[62]
- DynamoDB: Global Secondary Indexes[63]
- Amazon DynamoDB Accelerator (DAX)[64]
- Developer Guide: Kinesis Streams[65]

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 very first proof of concept to the ongoing operation of production workloads, adopting the practices in this document will enable you to build and operate cost-aware systems that achieve business outcomes and minimize costs, thus allowing your business to maximize its return on investment.

Definition

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

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

As with the other pillars, there are tradeoffs to consider. For example, do you want to optimize for speed to market or for cost? In some cases, it's best to optimize for speed—going to market quickly, shipping new features, or simply meeting a deadline—rather than investing in upfront cost optimization. Design decisions are sometimes guided by haste as opposed to empirical data, as the temptation always exists to overcompensate "just in case" rather than spend time benchmarking for the most cost-optimal deployment. This often leads to drastically over-provisioned and under-optimized deployments. The following sections provide techniques and strategic guidance for the initial and ongoing cost optimization of your deployment.

Generally, serverless architectures tend to reduce costs due to the fact that some of the services (like AWS Lambda) don't cost anything while they're idle. However, following certain best practices and making tradeoffs will help you reduce the cost of these solutions even more.

Best Practices

SERVCOST 1: What is your strategy for deciding the most optimal Lambda memory allocation?

Cost-Effective Resources

Serverless architectures are easier to manage in terms of correct resource allocation. The fact that almost no sizing is required with architecting and the ability to scale based on demand with services such as AWS Lambda reduces the number of decisions to make, such as cluster or instance sizing, storage, etc.

However, as we mentioned in the performance efficiency and operational excellence pillar sections, optimal memory allocation based on testing scenarios is needed to ensure the best cost/performance.

Also as described in the operational excellence pillar section, fine tuning memory and timeout will have a significant impact, not only on performance and operational procedures, but it also might reduce cost.

Better memory allocation of your Lambda functions will reduce the execution time, therefore the cost will be reduced. Also, CPU allocation is directly related to the amount of memory you allocate. The more memory you allocate, the more CPU will be allocated, which will impact performance.

Using the least amount of memory might seem like the perfect strategy for reducing costs, but using less memory means that each Lambda function will require more time to execute. Therefore, it can be more expensive due to the 100-ms incremental billing dimension.

Matching Supply and Demand

The AWS serverless architecture is designed to scale based on demand, so you don't need to worry about overprovisioning or under provisioning.

Expenditure Awareness

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

We recommend reading the AWS Well-Architected Framework whitepaper to dive deep into the topics discussed there. However, some of the questions mentioned there might not fully apply to serverless architectures. An example of

this would be the need to decommission resources such as AWS Lambda since it doesn't cost anything when idle.

For all other scenarios, such as unused APIs, Kinesis shards, or DynamoDB tables, existing questions and recommendations still apply and so there is no unique practice to serverless applications here.

Optimizing Overtime

As AWS releases new services and features, it is a best practice to review your existing architectural decisions to ensure that they continue to be the most cost effective. Once your infrastructure is running on a serverless architecture, you should reiterate in order to optimize costs in topics such as Lambda executions or logs storage.

> SERVCOST 2: What is your strategy for code logging in your Lambda functions?

AWS Lambda uses CloudWatch Logs to store the output of the executions in order 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.

When deploying your functions to AWS Lambda, it is important to remove unnecessary print statements within the code as this will be ingested into CloudWatch and increase the cost per ingestion of the same. A good approach to maintain these prints when needed is the use tools or libraries and best practices such as logging[66] and set the correct logging level whenever it's needed through environment variables. This way, unless specifically activated, logs ingested will be INFO and not DEBUG.

In order to reduce log storage costs, it is recommended to leverage the following features:

- Use log retention periods for Amazon CloudWatch Logs groups for AWS Lambda.
- Export logs to a more cost-effective platform such as Amazon S3 or Amazon ES.

With these two approaches, you will be able to save costs in terms of log storage and, if needed, explore these logs with different tools directly on Amazon S3 (with, for example, Amazon Athena) or upload them to Amazon ES for troubleshooting.

SERVCOST 3: Is your code architecture running unnecessary Lambda functions in order to reduce complexity?

When designing the architecture, avoiding unnecessary Lambda executions might reduce cost as well. Using features such as API Gateway service proxy or direct integrations between IoT and other AWS services will avoid both cost increases for these Lambda functions and operational overhead when managing these resources.

Finally, optimizing your code in order to reduce the execution time will decrease the cost per execution of your Lambda functions.

With regard to unnecessary invocations, integration of both Kinesis and DynamoDB with AWS Lambda makes it ideal to batch requests into a single invocation when latency is an acceptable tradeoff for throughput. This enables you to reduce overall concurrency, invocations, and cost.

Most serverless architectures use API Gateway as the entry point for final users. This is due to the fact that a RESTful API, agnostic of the implementation, is always a good service contract between our infrastructure and the final user. For more information, see the Microservices scenario.

In some cases, this might impact cost but can be avoided.

Three different approaches are considered here:

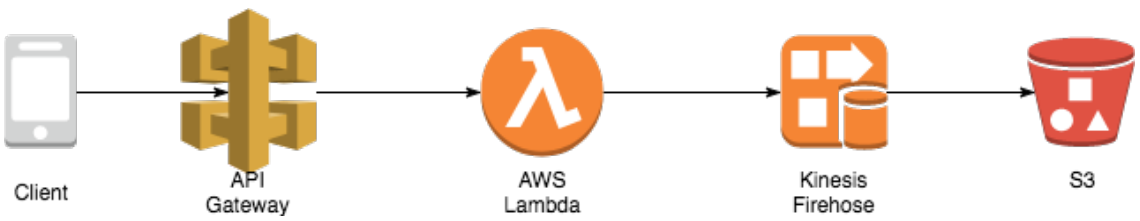

Figure 10: Sending data to Amazon S3 using Kinesis Firehose

In this scenario, API Gateway will execute a Lambda function that will pass the data to Kinesis Firehose and further on to Amazon S3. Here the cost comes from all of these services.

However, a different approach could be:

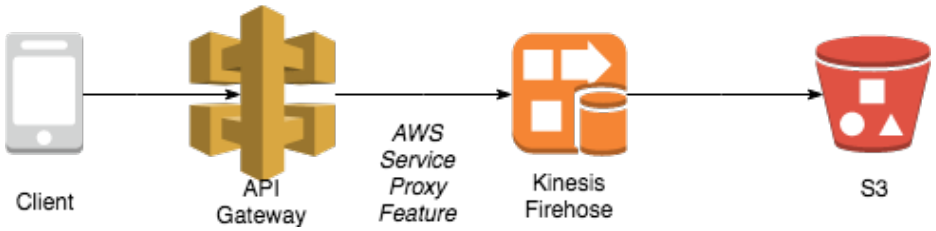

Figure 11: Reducing cost of sending data to Amazon S3 by implementing AWS service proxy

With this approach, we remove the cost of using Lambda and unnecessary invocations by implementing the AWS Service Proxy Feature within API Gateway. However, this might introduce some extra complexity when dealing with other services such as Kinesis since, for example, we need to define shards for ingestion within each call.

We can also stream data directly from the client using the Kinesis Firehose SDK into an S3 bucket, thereby removing the costs associated with API Gateway and AWS Lambda and simplify our architecture altogether.

Figure 12: Reducing cost of sending data to Amazon S3 by streaming directly using the Kinesis Firehose SDK

Of course, with this last implementation we won't benefit from using the RESTful API on our application but will reduce our costs and even the latency of our streams. Depending on the specific use case, one or another of these approaches might fit your workload.

SERVCOST 4: How are you optimizing your code to run in the least amount of time possible?

Some of the workloads our customers implement in their serverless architectures require functions to run for a long time. Within these workloads, some functions might wait for a resource to become available. This wait state can be implemented within the Lambda code waiting for some specific amount of time, however, we recommend implementing the waiting state using Step Functions instead.

With this pattern, instead of having your Lambda function waiting a specific amount of time for a resource to become available, which will incur charges and waste resources when sitting idle, you can reduce costs by implementing this wait with Step Functions. For example, in the image below, we poll an AWS Batch job and review its state every 30 s to see if it has finished. Instead of coding this wait within the Lambda function, we implement a poll (*GetJobStatus*) + wait (*Wait30Seconds*) + decider (*CheckJobStatus*).

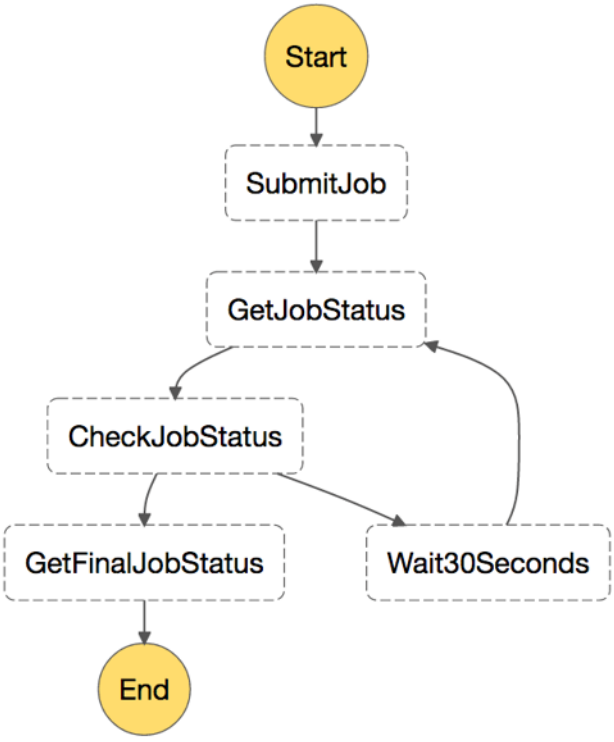

Figure 13: Implementing a wait state with AWS Step Functions

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

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

Key AWS Services

Key AWS services for cost optimization are CloudWatch Logs, DynamoDB, Kinesis, API Gateway, Step Functions, and AWS Batch.

Resources

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

Documentation & Blogs

- CloudWatch Logs Retention[67]
- Exporting CloudWatch Logs to Amazon S3[68]
- Streaming CloudWatch Logs to Amazon ES[69]
- Defining wait states in Step Functions state machines[70]
- Coca-Cola Vending Pass State Machine Powered by Step Functions[71]
- Building high throughput genomics batch workflows on AWS[72]

Whitepaper

- Optimizing Enterprise Economics with Serverless Architectures[73]

Conclusion

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

For reliability, by regularly testing failure pathways you will be more likely to catch errors before they reach production. For performance, starting backward from customer expectation will allow you to design for optimal experience. There are a number of AWS tools to help optimize performance as well. For cost optimization, you can reduce unnecessary waste within your serverless application by sizing resources in accordance with traffic demand. For operations, your architecture should strive toward automation in responding to events. Finally, a secure application will protect your organization's sensitive information assets and meet any compliance requirements at every layer.

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

Contributors

The following individuals and organizations contributed to this document:

- Adam Westrich: Senior Solutions Architect, Amazon Web Services
- Mark Bunch: Enterprise Solutions Architect, Amazon Web Services

- Ignacio Garcia Alonso: Solutions Architect, Amazon Web Services
- Heitor Lessa: Specialist Solutions Architect, Amazon Web Services
- Philip Fitzsimons: Sr. Manager Well-Architected, Amazon Web Services
- Dave Walker: Specialist Solutions Architect, Amazon Web Services

Further Reading

For additional information, see the following:

- AWS Well-Architected Framework[74]
- Serverless Architectures with AWS Lambda[75]

Notes

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[34] https://snyk.io/

[35] https://aws.amazon.com/answers/account-management/limit-monitor/

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[66] https://docs.python.org/2/library/logging.html

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

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

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

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

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

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

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

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

[75] https://d1.awsstatic.com/whitepapers/serverless-architectures-with-aws-lambda.pdf