AWS Serverless Multi-Tier Architectures

With Amazon API Gateway and AWS Lambda

First Published November 2015
Updated October 20, 2021

Notices

Contents

Abstract

This whitepaper illustrates how innovations from Amazon Web Services (AWS) can be used to change the way you design multi-tier architectures and implement popular patterns such as microservices, mobile backends, and single-page applications. Architects and developers can use Amazon API Gateway, AWS Lambda, and other services to reduce the development and operations cycles required to create and manage multi-tiered applications.

Introduction

The multi-tier application (three-tier, *n*-tier, and so forth) has been a cornerstone architecture pattern for decades, and remains a popular pattern for user-facing applications. Although the language used to describe a multi-tier architecture varies, a multi-tier application generally consists of the following components:

- **Presentation tier** – Component that the user directly interacts with (for example, webpages and mobile app UIs).
- **Logic tier** – Code required to translate user actions to application functionality (for example, CRUD database operations and data processing).
- **Data tier** – Storage media (for example, databases, object stores, caches, and file systems) that hold the data relevant to the application.

The multi-tier architecture pattern provides a general framework to ensure decoupled and independently scalable application components can be separately developed, managed, and maintained (often by distinct teams).

As a consequence of this pattern in which the network (a tier must make a network call to interact with another tier) acts as the boundary between tiers, developing a multi-tier application often requires creating many undifferentiated application components. Some of these components include:

- Code that defines a message queue for communication between tiers
- Code that defines an application programming interface (API) and a data model
- Security-related code that ensures appropriate access to the application

All of these examples can be considered “boilerplate” components that, while necessary in multi-tier applications, do not vary greatly in their implementation from one application to the next.

AWS offers a number of services that enable the creation of serverless multi-tier applications—greatly simplifying the process of deploying such applications to production and removing the overhead associated with traditional server management. Amazon API Gateway, a service for creating and managing APIs, and AWS Lambda, a service for running arbitrary code functions, can be used together to simplify the creation of robust multi-tier applications.

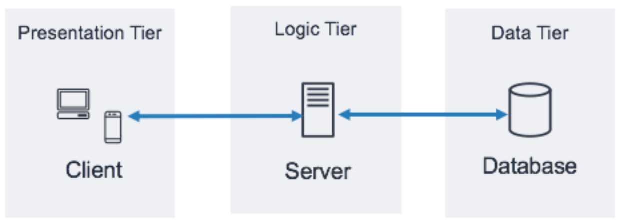
API Gateway's integration with AWS Lambda enables user-defined code functions to be initiated directly through HTTPS requests. Regardless of the request volume, both API Gateway and Lambda scale automatically to support exactly the needs of your application (refer to Amazon API Gateway quotas and important notes for scalability information). By combining these two services, you can create a tier that enables you to write only the code that matters to your application and not focus on various other undifferentiating aspects of implementing a multi-tiered architecture such as architecting for high availability, writing client SDKs, server and operating system (OS) management, scaling, and implementing a client authorization mechanism.

API Gateway and Lambda enable the creation of a serverless logic tier. Depending on your application requirements, AWS also provides options to create a serverless presentation tier (for example, with Amazon CloudFront and Amazon Simple Storage Service (Amazon S3) and data tier (for example, Amazon Aurora and Amazon DynamoDB).

This whitepaper focuses on the most popular example of a multi-tiered architecture, the **three-tier** web application. However, you can apply this multi-tier pattern well beyond a typical three-tier web application.

Three-tier architecture overview

The three-tier architecture is the most popular implementation of a multi-tier architecture, and consists of a single presentation tier, a logic tier, and a data tier. The following illustration shows an example of a simple, generic three-tier application.



Architectural pattern for a three-tier application

There are many great online resources where you can learn more about the *general* three-tier architecture pattern. This whitepaper focuses on a specific implementation pattern for this architecture using API Gateway and Lambda.

Serverless logic tier

The logic tier of the three-tier architecture represents the brains of the application. This is where using API Gateway and AWS Lambda can have the most impact compared to a traditional, server-based implementation. The features of these two services enable you to build a serverless application that is highly available, scalable, and secure. In a traditional model, your application could require thousands of servers; however, by using Amazon API Gateway and AWS Lambda you are not responsible for server management in any capacity. In addition, by using these managed services together, you gain the following benefits:

- **Lambda**
  - No OS to choose, secure, patch, or manage
  - No servers to right size, monitor, or scale
  - Reduced risk to your cost from over-provisioning
  - Reduced risk to your performance from under-provisioning
- **API Gateway**
  - Simplified mechanisms to deploy, monitor, and secure APIs
  - Improved API performance through caching and content delivery

AWS Lambda

AWS Lambda is a compute service that enables you to run arbitrary code functions in any of the supported languages (Node.js, Python, Ruby, Java, Go, .NET. For more information, refer to Lambda FAQs) without provisioning, managing, or scaling servers. Lambda functions are run in a managed, isolated container, and are launched in response to an event which can be one of several programmatic triggers that AWS makes available, called an *event source.* Refer to Lambda FAQs for all event sources.

Many popular use cases for Lambda revolve around event-driven data processing workflows, such as processing files stored in Amazon S3 or streaming data records from Amazon Kinesis. When used in conjunction with API Gateway, a Lambda function performs the functionality of a typical web service: it initiates code in response to a client HTTPS request; API Gateway acts as the front door for your logic tier, and Lambda invokes the application code.

Your business logic goes here, no servers necessary

Lambda requires that you to write code functions, called *handlers*, which will run when initiated by an event. To use Lambda with API Gateway, you can configure API Gateway to launch handler functions when an HTTPS request to your API occurs. In a serverless multi-tier architecture, each of the APIs you create in API Gateway will integrate with a Lambda function (and the handler within) that invokes the business logic required.

Using AWS Lambda functions to compose the logic tier enables you to define a desired level of granularity for exposing the application functionality (one Lambda function per API or one Lambda function per API method). Inside the Lambda function, the handler can reach out to any other dependencies (for example, other methods you've uploaded with your code, libraries, native binaries, and external web services), or even other Lambda functions.

Creating or updating a Lambda function requires either uploading code as a Lambda deployment package in a zip file to an Amazon S3 bucket, or packaging code as a container image along with all the dependencies. The functions can use different deployment methods, such as AWS Management Console, running AWS Command Line Interface (CLI), or running infrastructure as code templates or frameworks such as AWS CloudFormation, AWS Serverless Application Model (AWS SAM), or AWS Cloud Development Kit (AWS CDK). When you create your function using any of these methods, you specify which method inside your deployment package will act as the request handler. You can reuse the same deployment package for multiple Lambda function definitions, where each Lambda function might have a unique handler within the same deployment package.

Lambda security

To run a Lambda function, it must be invoked by an event or service that is permitted by an AWS Identity and Access Management (IAM) policy. Using IAM policies, you can create a Lambda function that cannot be initiated at all unless it is invoked by an API Gateway resource that you define. Such policy can be defined using resource-based policy across various AWS services.

Each Lambda function assumes an IAM role that is assigned when the Lambda function is deployed. This IAM role defines the other AWS services and resources your Lambda function can interact with (for example, Amazon DynamoDB table and Amazon S3). In context of Lambda function, this is called an execution role.

Do not store sensitive information inside a Lambda function. IAM handles access to AWS services through the Lambda execution role; if you need to access other credentials (for example, database credentials and API keys) from inside your Lambda function, you can use AWS Key Management Service (AWS KMS) with environment variables, or use a service such as AWS Secrets Manager to keep this information safe when not in use.

Performance at scale

Code pulled in as a container image from Amazon Elastic Container Registry (Amazon ECR), or from a zip file uploaded to Amazon S3, runs in an isolated environment managed by AWS. You do not have to scale your Lambda functions—each time an event notification is received by your function, AWS Lambda locates available capacity within its compute fleet and runs your code with runtime, memory, disk, and timeout configurations that you define. With this pattern, AWS can start as many copies of your function as needed.

A Lambda-based logic tier is always right sized for your customer needs. The ability to quickly absorb surges in traffic through managed scaling and concurrent code initiation, combined with Lambda pay-per-use pricing, enables you to always meet customer requests while simultaneously not paying for idle compute capacity.

Serverless deployment and management

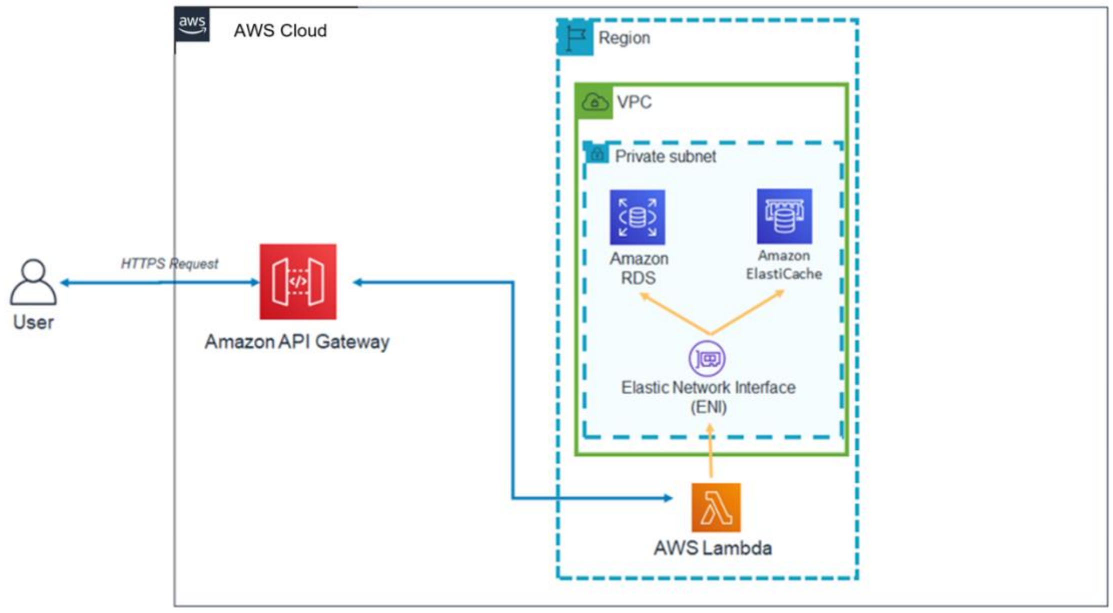
To help you deploy and manage your Lambda functions, use AWS Serverless Application Model (AWS SAM), an open-source framework that includes:

- **AWS SAM template specification** – Syntax used to define your functions and describe their environments, permissions, configurations, and events for simplified upload and deployment.
- **AWS SAM CLI** – Commands that enable you to verify AWS SAM template syntax, invoke functions locally, debug Lambda functions, and deployment package functions.

You can also use AWS CDK, which is a software development framework for defining cloud infrastructure using programming languages and provisioning it through CloudFormation. AWS CDK provides an imperative way to define AWS resources, whereas AWS SAM provides a declarative way.

Typically, when you deploy a Lambda function, it is invoked with permissions defined by its assigned IAM role, and is able to reach internet-facing endpoints. As the core of your logic tier, AWS Lambda is the component directly integrating with the data tier. If your data tier contains sensitive business or user information, it is important to ensure that this data tier is appropriately isolated (in a private subnet).

You can configure a Lambda function to connect to private subnets in a virtual private cloud (VPC) in your AWS account if you want the Lambda function to access resources that you cannot expose publicly, like a private database instance. When you connect a function to a VPC, Lambda creates an elastic network interface for each subnet in your function's VPC configuration and elastic network interface is used to access your internal resources privately.



Lambda architecture pattern inside a VPC

The use of Lambda with VPC means that databases and other storage media that your business logic depends on can be made inaccessible from the internet. The VPC also ensures that the only way to interact with your data from the internet is through the APIs that you've defined and the Lambda code functions that you have written.

API Gateway

API Gateway is a fully managed service that enables developers to create, publish, maintain, monitor, and secure APIs at any scale.

Clients (that is, presentation tiers) integrate with the APIs exposed through API Gateway using standard HTTPS requests. The applicability of APIs exposed through API Gateway to a service-oriented multi-tier architecture is the ability to separate individual pieces of application functionality and expose this functionality through REST endpoints. API Gateway has specific features and qualities that can add powerful capabilities to your logic tier.

Integration with Lambda

Amazon API Gateway supports both REST and HTTP types of APIs. An API Gateway API is made up of resources and methods. A resource is a logical entity that an app can access through a resource path (for example, `/tickets`). A method corresponds to an API request that is submitted to an API resource (for example, `GET /tickets`). API Gateway enables you to back each method with a Lambda function, that is, when you call the API through the HTTPS endpoint exposed in API Gateway, API Gateway invokes the Lambda function.

You can connect API Gateway and Lambda functions using proxy integrations and non-proxy integrations.

Proxy integrations

In a proxy integration, the entire client HTTPS request is sent as-is to the Lambda function. API Gateway passes the entire client request as the event parameter of the Lambda handler function, and the output of the Lambda function is returned directly to the client (including status code, headers, and so forth).

Non-proxy integrations

In a non-proxy integration, you configure how the parameters, headers, and body of the client request are passed to the event parameter of the Lambda handler function. Additionally, you configure how the Lambda output is translated back to the user.

Note: API Gateway can also proxy to additional serverless resources outside of AWS Lambda, such as mock integrations (useful for initial application development) and direct proxy to S3 objects.

Stable API performance across regions

Each deployment of API Gateway includes a Amazon CloudFront distribution under the hood. CloudFront is a content delivery service that uses Amazon's global network of edge locations as connection points for clients using your API. This helps decrease the response latency of your API. By using multiple edge locations across the world, CloudFront also provides capabilities to combat distributed denial of service (DDoS) attack scenarios. For more information, review the AWS Best Practices for DDoS Resiliency whitepaper.

You can improve the performance of specific API requests by using API Gateway to store responses in an optional in-memory cache. This approach not only provides performance benefits for repeated API requests, but it also reduces the number of times your Lambda functions are invoked, which can reduce your overall cost.

Encourage innovation and reduce overhead with built-in features

The development cost to build any new application is an investment. Using API Gateway can reduce the amount of time required for certain development tasks and lower the total development cost, enabling organizations to more freely experiment and innovate.

During initial application development phases, implementation of logging and metrics gathering are often neglected to deliver a new application more quickly. This can lead to technical debt and operational risk when deploying these features to an application running in production. API Gateway integrates seamlessly with Amazon CloudWatch, which collects and processes raw data from API Gateway into readable, near real-time metrics for monitoring API implementation. API Gateway also supports access logging with configurable reports, and AWS X-Ray tracing for debugging. Each of these features requires no code to be written, and can be adjusted in applications running in production without risk to the core business logic.

The overall lifetime of an application might be unknown, or it might be known to be short-lived. Creating a business case for building such applications can be made easier if your starting point already includes the managed features that API Gateway provides, and if you only incur infrastructure costs after your APIs begin receiving requests. For more information, refer to Amazon API Gateway pricing.

Iterate rapidly, stay agile

Using API Gateway and AWS Lambda to build the logic tier of your API enables you to quickly adapt to the changing demands of your user base by simplifying API deployment and version management.

Stage deployment

When you deploy an API in API Gateway, you must associate the deployment with an API Gateway stage—each stage is a snapshot of the API and is made available for client apps to call. Using this convention, you can easily deploy apps to *dev*, *test*, *stage*, or *prod* stages, and move deployments between stages. Each time you deploy your API to a stage, you create a different version of the API which can be reverted if necessary. These features enable existing functionality and client dependencies to continue undisturbed while new functionality is released as a separate API version.

Decoupled integration with Lambda

The integration between API in API Gateway and Lambda function can be decoupled using API Gateway stage variables and a Lambda function alias. This simplifies and speeds up the API deployment. Instead of configuring the Lambda function name or alias in the API directly, you can configure stage variable in API which can point to a particular alias in the Lambda function. During deployment, change the stage variable value to point to a Lambda function alias and API will run the Lambda function version behind the Lambda alias for a particular stage.

Canary release deployment

Canary release is a software development strategy in which a new version of an API is deployed for testing purposes, and the base version remains deployed as a production release for normal operations on the same stage. In a canary release deployment, total API traffic is separated at random into a production release and a canary release with a preconfigured ratio. APIs in API Gateway can be configured for the canary release deployment to test new features with a limited set of users.

Custom domain names

You can provide an intuitive business-friendly URL name to API instead of the URL provided by API Gateway. API Gateway provides features to configure custom domain for the APIs. With custom domain names, you can set up your API's hostname, and choose a multi-level base path (for example, `myservice`, `myservice/cat/v1`, or `myservice/dog/v2`) to map the alternative URL to your API.

Prioritize API security

All applications must ensure that only authorized clients have access to their API resources. When designing a multi-tier application, you can take advantage of several different ways in which API Gateway contributes to securing your logic tier:

Transit security

All requests to your APIs can be made through HTTPS to enable encryption in transit.

API Gateway provides built-in SSL/TLS Certificates—if using the custom domain name option for public APIs, you can provide your own SSL/TLS certificate using AWS Certificate Manager. API Gateway also supports mutual TLS (mTLS) authentication. Mutual TLS enhances the security of your API and helps protect your data from attacks such as client spoofing or man-in-the middle attacks.

API authorization

Each resource and method combination that you create as part of your API is granted a unique Amazon Resource Name (ARN) that can be referenced in AWS Identity and Access Management (IAM) policies.

There are three general methods to add authorization to an API in API Gateway:

- **IAM roles and policies.** Clients use AWS Signature Version 4 (SigV4) authorization and IAM policies for API access. The same credentials can restrict or permit access to other AWS services and resources as needed (for example, S3 buckets or Amazon DynamoDB tables).
- **Amazon Cognito user pools.** Clients sign in through an Amazon Cognito user pool and obtain tokens which are included in the authorization header of a request.
- **Lambda authorizer.** Define a Lambda function that implements a custom authorization scheme that uses a bearer token strategy (for example, OAuth and SAML) or uses request parameters to identify users.

Access restrictions

API Gateway supports the generation of API keys and association of these keys with a configurable usage plan. You can monitor API key usage with CloudWatch.

API Gateway supports throttling, rate limits, and burst rate limits for each method in your API.

Private APIs

Using API Gateway, you can create private REST APIs that can only be accessed from your virtual private cloud in Amazon VPC by using an interface VPC endpoint. This is an endpoint network interface that you create in your VPC.

Using resource policies, you can enable or deny access to your API from selected VPCs and VPC endpoints, including across AWS accounts. Each endpoint can be used to access multiple private APIs. You can also use AWS Direct Connect to establish a connection from an on-premises network to Amazon VPC and access your private API over that connection.

In all cases, traffic to your private API uses secure connections and does not leave the Amazon network—it is isolated from the public internet.

Firewall protection using AWS WAF

Internet-facing APIs are vulnerable to malicious attacks. AWS WAF is a web application firewall which helps protect APIs from such attacks. It protects APIs from common web exploits such as SQL injection and cross-site scripting attacks. You can use AWS WAF with API Gateway to help protect APIs.

Data tier

Using AWS Lambda as your logic tier does not limit the data storage options available in your data tier. Lambda functions connect to any data storage option by including the appropriate database driver in the Lambda deployment package, and use IAM role-based access or encrypted credentials (through AWS KMS or Secrets Manager).

Choosing a data store for your application is highly dependent on your application requirements. AWS offers a number of serverless and non-serverless data stores that you can use to compose the data tier of your application.

Serverless data storage options

- Amazon S3 is an object storage service that offers industry-leading scalability, data availability, security, and performance.

- Amazon Aurora is a MySQL-compatible and PostgreSQL-compatible relational database built for the cloud, that combines the performance and availability of traditional enterprise databases with the simplicity and cost-effectiveness of open-source databases. Aurora offers both serverless and traditional usage models.
- Amazon DynamoDB is a key-value and document database that delivers single-digit millisecond performance at any scale. It is a fully managed, serverless, multi-region, durable database with built-in security, backup and restore, and in-memory caching for internet-scale applications.
- Amazon Timestream is a fast, scalable, fully managed time series database service for IoT and operational applications that makes it simple to store and analyze trillions of events per day at 1/10th the cost of relational databases. Driven by the rise of IoT devices, IT systems, and smart industrial machines, time series data—data that measures how things change over time—is one of the fastest growing data types.
- Amazon Quantum Ledger Database (Amazon QLDB) is a fully managed ledger database that provides a transparent, immutable, and cryptographically verifiable transaction log owned by a central trusted authority. Amazon QLDB tracks each and every application data change and maintains a complete and verifiable history of changes over time.
- Amazon Keyspaces (for Apache Cassandra) is a scalable, highly available, and managed Apache Cassandra–compatible database service. With Amazon Keyspaces, you can run your Cassandra workloads on AWS using the same Cassandra application code and developer tools that you use today. You don't have to provision, patch, or manage servers, and you don't have to install, maintain, or operate software. Amazon Keyspaces is serverless, so you pay for only the resources you use and the service can automatically scale tables up and down in response to application traffic.

- Amazon Elastic File System (Amazon EFS) provides a simple, serverless, set-and-forget, elastic file system that lets you share file data without provisioning or managing storage. It can be used with AWS Cloud services and on-premises resources, and is built to scale on demand to petabytes without disrupting applications. With Amazon EFS, you can grow and shrink your file systems automatically as you add and remove files, eliminating the need to provision and manage capacity to accommodate growth. Amazon EFS can be mounted with Lambda function which makes it a viable file storage option for APIs.

Non-serverless data storage options

- Amazon Relational Database Service (Amazon RDS) is a managed web service that enables you to set up, operate, and scale a relational database using several engines (Aurora, PostgreSQL, MySQL, MariaDB, Oracle, and Microsoft SQL Server) and running on several different database instance types that are optimized for memory, performance, or I/O.
- Amazon Redshift is a fully managed, petabyte-scale data warehouse service in the cloud.
- Amazon ElastiCache is a fully managed deployment of Redis or Memcached. Seamlessly deploy, run, and scale popular open source compatible in-memory data stores.
- Amazon Neptune is a fast, reliable, fully managed graph database service that makes it simple to build and run applications that work with highly connected datasets. Neptune supports popular graph models—property graphs and W3C Resource Description Framework (RDF)—and their respective query languages, enabling you to easily build queries that efficiently navigate highly connected datasets.
- Amazon DocumentDB (with MongoDB compatibility) is a fast, scalable, highly available, and fully managed document database service that supports MongoDB workloads.
- Finally, you can also use data stores running independently on Amazon EC2 as the data tier of a multi-tier application.

Presentation tier

The presentation tier is responsible for interacting with the logic tier through the API Gateway REST endpoints exposed over the internet. Any HTTPS capable client or device can communicate with these endpoints, giving your presentation tier the flexibility to take many forms (desktop applications, mobile apps, webpages, IoT devices, and so forth). Depending on your requirements, your presentation tier can use the following AWS serverless offerings:

- **Amazon Cognito –** A serverless user identity and data synchronization service that enables you to add user sign-up, sign-in, and access control to your web and mobile apps quickly and efficiently. Amazon Cognito scales to millions of users and supports sign-in with social identity providers, such as Facebook, Google, and Amazon, and enterprise identity providers through SAML 2.0.
- **Amazon S3 with CloudFront** – Enables you to serve static websites, such as single-page applications, directly from an S3 bucket without requiring provision of a web server. You can use CloudFront as a managed content delivery network (CDN) to improve performance and enable SSL/TL using managed or custom certificates.

AWS Amplify is a set of tools and services that can be used together or on their own, to help front-end web and mobile developers build scalable full stack applications, powered by AWS. Amplify offers a fully managed service for deploying and hosting static web applications globally, served by Amazon's reliable CDN with hundreds of points of presence globally and with built-in CI/CD workflows that accelerate your application release cycle. Amplify supports popular web frameworks including JavaScript, React, Angular, Vue, Next.js, and mobile platforms including Android, iOS, React Native, Ionic, and Flutter. Depending on your networking configurations and application requirements, you might need to enable your API Gateway APIs to be cross-origin resource sharing (CORS) – compliant. CORS compliance allows web browsers to directly invoke your APIs from within static webpages.

When you deploy a website with CloudFront, you are provided a CloudFront domain name to reach your application (for example, `d2d47p2vcczkh2.cloudfront.net`). You can use Amazon Route 53 to register domain names and direct them to your CloudFront distribution, or direct already-owned domain names to your CloudFront distribution. This enables users to access your site using a familiar domain name. Note that you can also assign a custom domain name using Route 53 to your API Gateway distribution, which enables users to invoke APIs using familiar domain names.

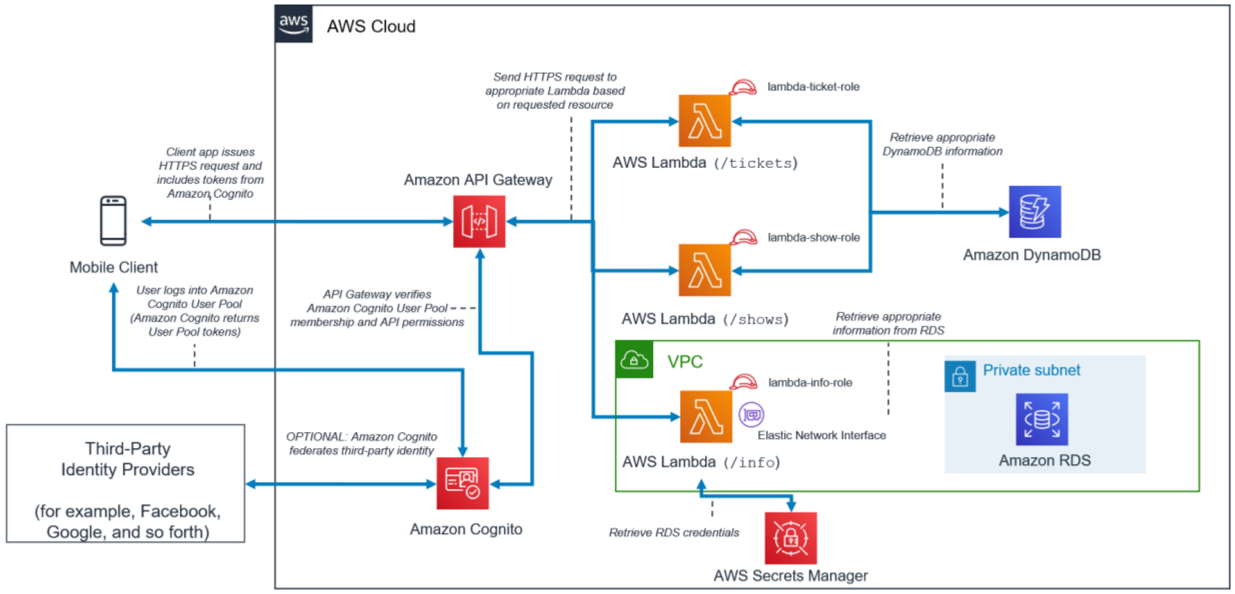
Sample architecture patterns

You can implement popular architecture patterns using API Gateway and AWS Lambda as your logic tier. This whitepaper includes the most popular architecture patterns that use AWS Lambda-based logic tiers:

- **Mobile backend** – A mobile application communicates with API Gateway and Lambda to access application data. This pattern can be extended to generic HTTPS clients that don’t use serverless AWS resources to host presentation tier resources (such as desktop clients, web server running on EC2, and so forth).
- **Single-page application** – A single-page application hosted in Amazon S3 and CloudFront communicates with API Gateway and AWS Lambda to access application data.
- **Web application** – The web application is a general-purpose, event-driven, web application back-end that uses AWS Lambda with API Gateway for its business logic. It also uses DynamoDB as its database and Amazon Cognito for user management. All static content is hosted using Amplify.

In addition to these two patterns, this whitepaper discusses the applicability of AWS Lambda and API Gateway to a general microservice architecture. A microservice architecture is a popular pattern that, although not a standard three-tier architecture, involves decoupling application components and deploying them as stateless, individual units of functionality that communicate with each other.

Mobile backend

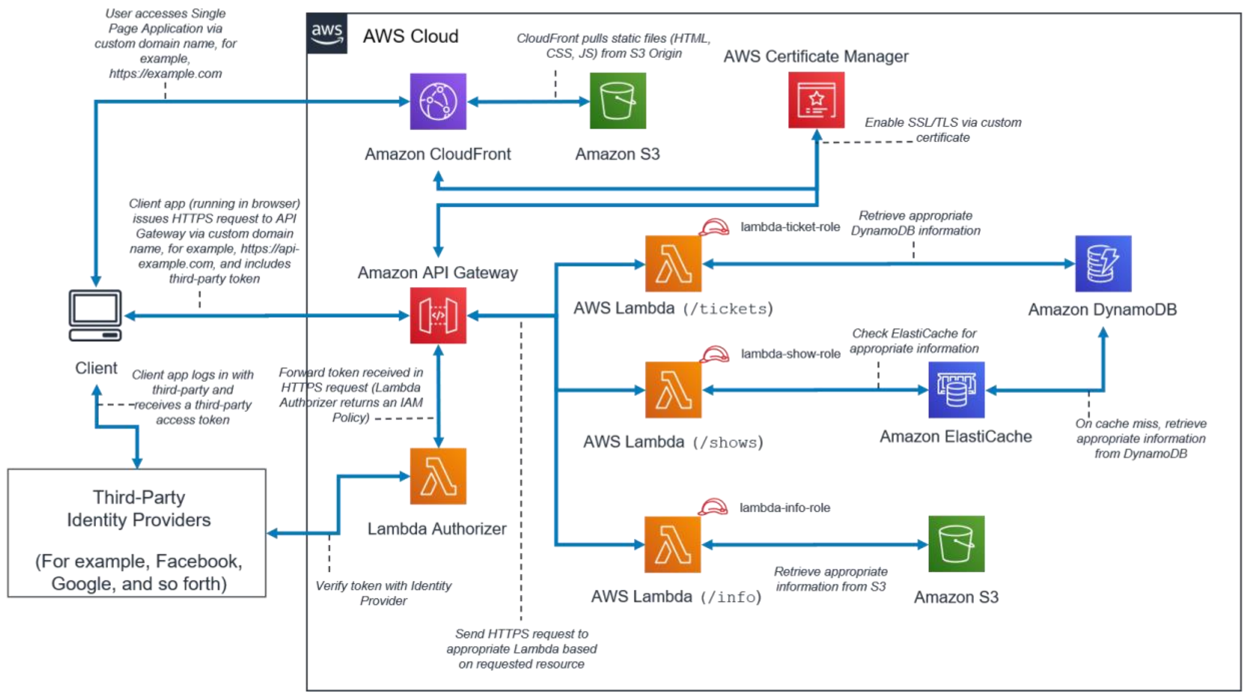

AWS Cloud
Send HTTPS request to appropriate Lambda based on requested resource
lambda-ticket-role
AWS Lambda (/tickets)
Retrieve appropriate DynamoDB information
Amazon DynamoDB
Client app issues HTTPS request and includes tokens from Amazon Cognito
Amazon API Gateway
Mobile Client
lambda-show-role
AWS Lambda (/shows)
User logs into Amazon Cognito User Pool (Amazon Cognito returns User Pool tokens)
API Gateway verifies Amazon Cognito User Pool membership and API permissions
Retrieve appropriate information from RDS
VPC
lambda-info-role
Private subnet
Elastic Network Interface
Amazon RDS
AWS Lambda (/info)
Third-Party Identity Providers
(for example, Facebook, Google, and so forth)
OPTIONAL: Amazon Cognito federates third-party identity
Amazon Cognito
Retrieve RDS credentials
AWS Secrets Manager

Architectural pattern for serverless mobile backend

Table 1 - Mobile backend tier components

Tier	Components
Presentation	Mobile application running on a user device.
Logic	API Gateway with AWS Lambda. This architecture shows three exposed services (`/tickets`, `/shows`, and `/info`). API Gateway endpoints are secured by Amazon Cognito user pools. In this method, users sign in to Amazon Cognito user pools (using a federated third party if necessary), and receive access and ID tokens that are used to authorize API Gateway calls. Each Lambda function is assigned its own Identity and Access Management (IAM) role to provide access to the appropriate data source.
Data	DynamoDB is used for the `/tickets` and `/shows` services. Amazon RDS is used for the `/info` service. This Lambda function retrieves Amazon RDS credentials from Secrets Manager and uses an elastic network interface to access the private subnet.

Single-page application

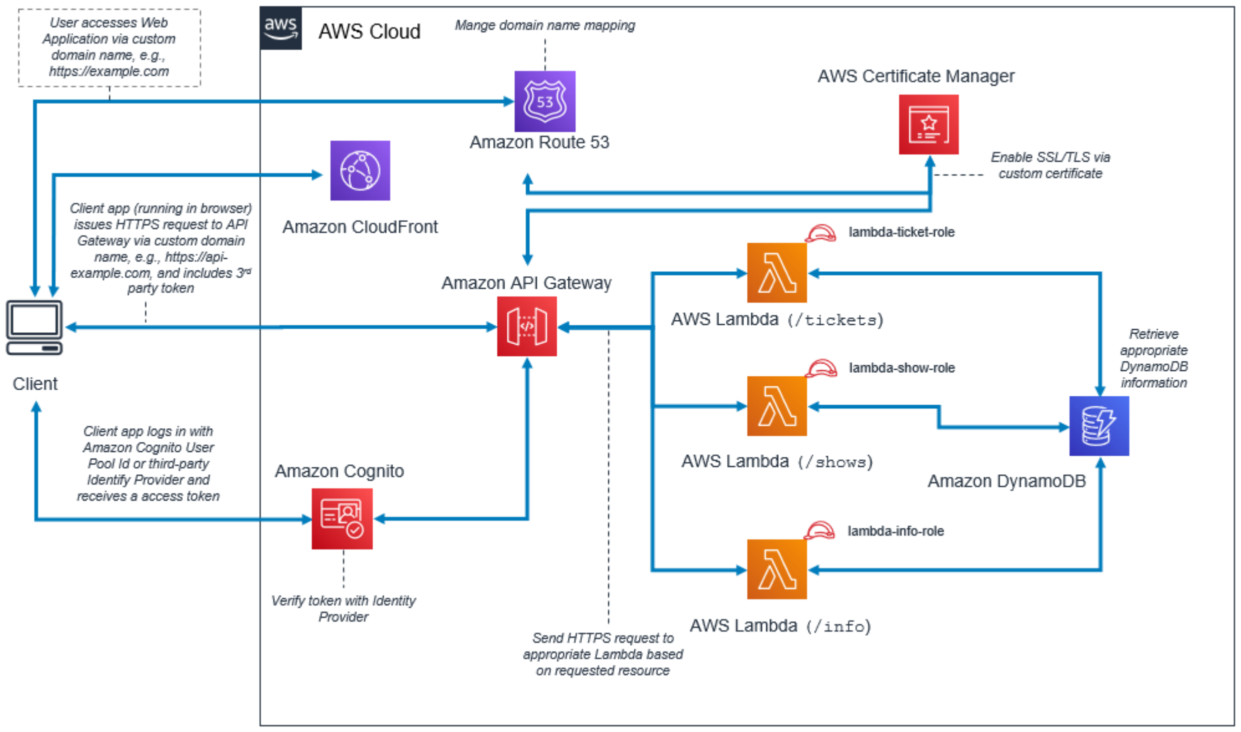

User accesses Single Page Application via custom domain name, for example, https://example.com
AWS Cloud
CloudFront pulls static files (HTML, CSS, JS) from S3 Origin
AWS Certificate Manager
Amazon CloudFront
Amazon S3
Enable SSL/TLS via custom certificate
Client app (running in browser) issues HTTPS request to API Gateway via custom domain name, for example, https://api-example.com, and includes third-party token
lambda-ticket-role
Retrieve appropriate DynamoDB information
Amazon API Gateway
AWS Lambda (/tickets)
Amazon DynamoDB
Client
Client app logs in with third-party and receives a third-party access token
Forward token received in HTTPS request (Lambda Authorizer returns an IAM Policy)
lambda-show-role
Check ElastiCache for appropriate information
AWS Lambda (/shows)
Amazon ElastiCache
On cache miss, retrieve appropriate information from DynamoDB
Third-Party Identity Providers
(For example, Facebook, Google, and so forth)
Lambda Authorizer
lambda-info-role
Verify token with Identity Provider
Retrieve appropriate information from S3
AWS Lambda (/info)
Amazon S3
Send HTTPS request to appropriate Lambda based on requested resource

Architectural pattern for serverless single-page application

Table 2 - Single-page application components

Tier	Components
Presentation	Static website content is hosted in Amazon S3 and distributed by CloudFront. AWS Certificate Manager allows a custom SSL/TLS certificate to be used.
Logic	API Gateway with AWS Lambda. This architecture shows three exposed services (`/tickets`, `/shows`, and `/info`). API Gateway endpoints are secured by a Lambda authorizer. In this method, users sign in through a third-party identity provider and obtain access and ID tokens. These tokens are included in API Gateway calls, and the Lambda authorizer validates these tokens and generates an IAM policy containing API initiation permissions. Each Lambda function is assigned its own IAM role to provide access to the appropriate data source.
Data	DynamoDB is used for the `/tickets` and `/shows` services. ElastiCache is used by the `/shows` service to improve database performance. Cache misses are sent to DynamoDB. Amazon S3 is used to host static content used by the `/info` service.

Web application


User accesses Web Application via custom domain name, e.g., https://example.com
AWS Cloud
Mange domain name mapping
Amazon Route 53
AWS Certificate Manager
Enable SSL/TLS via custom certificate
Amazon CloudFront
Client app (running in browser) issues HTTPS request to API Gateway via custom domain name, e.g., https://api-example.com, and includes 3rd party token
Amazon API Gateway
lambda-ticket-role
AWS Lambda (/tickets)
lambda-show-role
AWS Lambda (/shows)
Retrieve appropriate DynamoDB information
Amazon DynamoDB
Client
Client app logs in with Amazon Cognito User Pool Id or third-party Identify Provider and receives a access token
Amazon Cognito
Verify token with Identity Provider
lambda-info-role
AWS Lambda (/info)
Send HTTPS request to appropriate Lambda based on requested resource

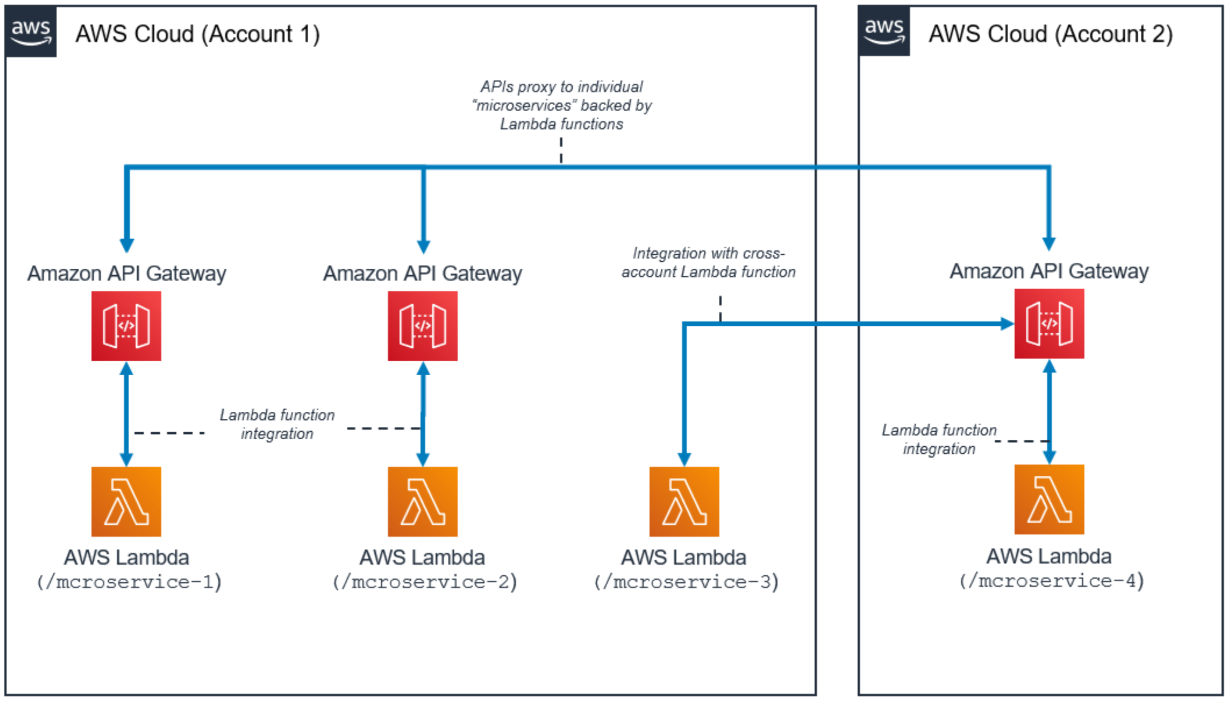
Architectural pattern for web application

Table 3 - Web application components

Tier	Components
Presentation	The front-end application is all static content (HTML, CSS, JavaScript and images) which are generated by React utilities like create-react-app. Amazon CloudFront hosts all these objects. The web application, when used, downloads all the resources to the browser and starts to run from there. The web application connects to the backend calling the APIs.
Logic	Logic layer is built using Lambda functions fronted by API Gateway REST APIs. This architecture shows multiple exposed services. There are multiple different Lambda functions each handling a different aspect of the application. The Lambda functions are behind API Gateway and accessible using API URL paths.

Tier	Components
	The user authentication is handled using Amazon Cognito User Pools or federated user providers. API Gateway uses out of box integration with Amazon Cognito. Only after a user is authenticated, the client will receive a JSON Web Token (JWT) which it should then use when making the API calls. Each Lambda function is assigned its own IAM role to provide access to the appropriate data source.
Data	In this particular example, DynamoDB is used for the data storage but other purpose-built Amazon database or storage services can be used depending on the use case and usage scenario.

Microservices with Lambda


AWS Cloud (Account 1)
AWS Cloud (Account 2)
APIs proxy to individual "microservices" backed by Lambda functions
Amazon API Gateway
Amazon API Gateway
Integration with cross-account Lambda function
Amazon API Gateway
Lambda function integration
Lambda function integration
AWS Lambda (/mcroservice-1)
AWS Lambda (/mcroservice-2)
AWS Lambda (/mcroservice-3)
AWS Lambda (/mcroservice-4)

Architectural pattern for microservices with Lambda

The microservice architecture pattern is not bound to the typical three-tier architecture; however, this popular pattern can realize significant benefits from the use of serverless resources.

In this architecture, each of the application components are decoupled and independently deployed and operated. An API created with API Gateway, and functions subsequently launched by AWS Lambda, is all that you need to build a microservice. Your team can use these services to decouple and fragment your environment to the level of granularity desired.

In general, a microservices environment can introduce the following difficulties: repeated overhead for creating each new microservice, issues with optimizing server density and utilization, complexity of running multiple versions of multiple microservices simultaneously, and proliferation of client-side code requirements to integrate with many separate services.

When you create microservices using serverless resources, these problems become less difficult to solve and, in some cases, simply disappear. The serverless microservices pattern lowers the barrier for the creation of each subsequent microservice (API Gateway even allows for the cloning of existing APIs, and use of Lambda functions in other accounts). Optimizing server utilization is no longer relevant with this pattern. Finally, API Gateway provides programmatically generated client SDKs in a number of popular languages to reduce integration overhead.

Conclusion

The multi-tier architecture pattern encourages the best practice of creating application components that are simple to maintain, decouple, and scale. When you create a logic tier where integration occurs by API Gateway and computation occurs within AWS Lambda, you realize these goals while reducing the amount of effort to achieve them. Together, these services provide an HTTPS API front end for your clients and a secure environment to apply your business logic while removing the overhead involved with managing typical server-based infrastructure.

Contributors

Contributors to this document include:

- Andrew Baird, AWS Solutions Architect
- Bryant Bost, AWS ProServe Consultant
- Stefano Buliani, Senior Product Manager, Tech, AWS Mobile
- Vyom Nagrani, Senior Product Manager, AWS Mobile

- Ajay Nair, Senior Product Manager, AWS Mobile
- Rahul Popat, Global Solutions Architect
- Brajendra Singh, Senior Solutions Architect

Further reading

For additional information, refer to:

- AWS Whitepapers and Guides

Document revisions

Date	Description
October 20, 2021	Updated for new service features and patterns.
June 1, 2021	Updated for new service features and patterns.
September 25, 2019	Updated for new service features.
November 1, 2015	First publication.