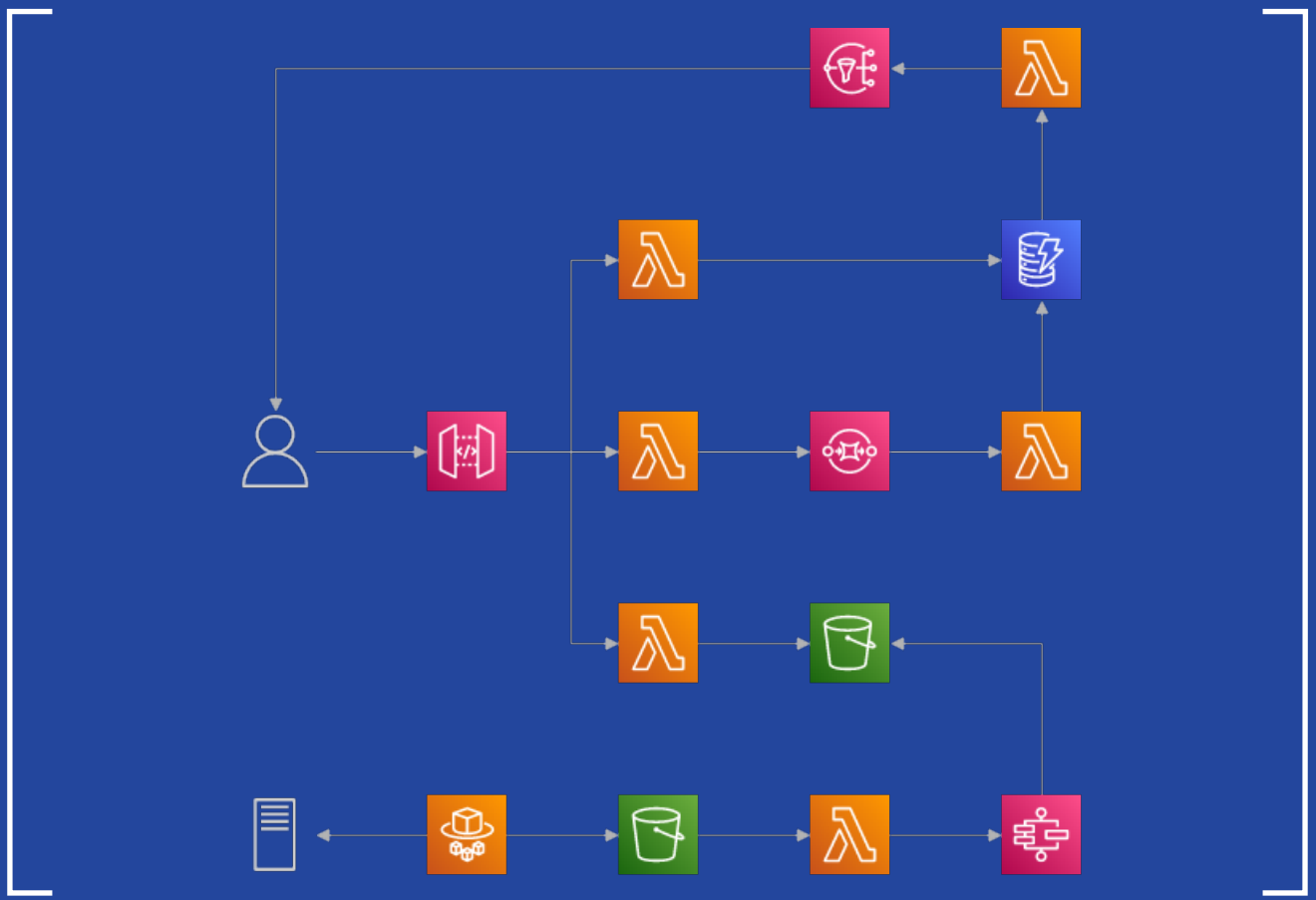
A Starter's Guide To

Serverless on AWS



Maciej Radzikowski

Better Dev .blog

Hi, I'm Maciej 👋

I'm a Software Developer and Architect, member of the AWS Community Builders.

I build serverless solutions on AWS and share my learnings on my blog and Twitter (@radzikowski_m).

While serverless is easy enough to get started and have your first application running in minutes, in the beginning, it can be overwhelming with the number of different tools, services, and approaches. **That is why I wrote this ebook** - to give you a basic orientation and best practices for doing serverless on AWS. I hope you will find it helpful.

Constellation background from svgbackgrounds.com.

v2022-05-22

TABLE OF CONTENT

What is serverless

In short, serverless is a cloud service model where you care only about your business logic. Infrastructure management and scaling are done for you.

Serverless is a broader term than Function as a Service (FaaS). FaaS relates strictly to computing, while serverless includes all other managed services, like databases.

Benefits of serverless

- **focus on the business logic** - no servers, operating systems, and applications to manage, which results in quicker time to market as well as lower operational and development costs
- **pay-per-use pricing** - billing for the actual usage in terms of requests count and execution time in most cases significantly reduces costs
- **scalability** - underlying infrastructure automatically scales up and down to handle the incoming traffic
- **high availability** - just like the scaling, also the resiliency lies on the vendor
- **no language lock-ins** - individual parts of the system can be created in the most appropriate languages and technologies
- **agility** - individual components are small and can be replaced or refactored easily

Part I
Infrastructure as Code

With serverless, even a not-so-big application is a compound of multiple resources. There may be several functions, API with numerous endpoints, tables, queues, and more.

Creating all those resources by hand in the web Console is a path to failure. It's not only time-consuming but also error-prone and hard to reproduce.

With IaC tools, all the **resources are defined with code and then created on AWS**. Thus, creating a separate environment with a full copy of the architecture for development or testing purposes is a matter of running a single command.

This part is not about either using IaC or not. It's about choosing the IaC tool.

CloudFormation

CloudFormation is AWS built-in IaC service. You define resources in YAML or JSON file called template (please, use YAML) and pass it to CloudFormation. Then you wait until everything is provisioned for you.

While capable of great things, **using CloudFormation directly is painful**. Some resources are extremely verbose to define. There are no proper variables. The list could go on.

However, there are several tools built on top of CloudFormation. They allow you to define infrastructure more conveniently while still using CloudFormation under the hood.

Serverless Framework

Serverless Framework is an AWS-independent project that solves most problems of pure CloudFormation. You still write the YAML template but with a **higher-level syntax**. You can declare Lambda functions, events that trigger them, API endpoints, and more in a **short, readable form**. Additionally, you have a **rich variables system**. Then everything is translated to a proper CloudFormation template and deployed.

The true power of the Serverless Framework are **plugins**. Community-driven, they simplify and automate generating resources.

For any resources not supported by the Framework or the plugins, you can use the standard CloudFormation syntax. But even this is much better with the use of SF variables.

CDK

Cloud Development Kit (CDK) is an AWS library to generate CloudFormation templates. While the idea is similar to how Serverless Framework works, declaring resources here is entirely different. **With CDK, you write code** in one of the supported languages: TypeScript, JavaScript, Python, Java, C#, or Go.

Writing infrastructure using a programming language enables a few interesting capabilities. For one, you can use logic you are well familiar with, like IF conditions. This allows easily adjusting created infrastructure components to the situation, like the environment you are deploying to. Another is syntax support giving you **parameters autocompletion and validation** straight in the editor.

While CDK allows declaring pure CloudFormation resources, it encourages the use of **higher-level Constructs**. Constructs define a resource or several, with sensible defaults. As a result, it improves your infrastructure and reduces the code length. You can also easily create your own custom Constructs and reuse them multiple times.

SAM

SAM is short for Serverless Application Model, and it's an AWS framework dedicated to provisioning serverless applications, as you could guess. It's also built on top of the CloudFormation.

SAM makes defining a few serverless resource types easier than with pure CloudFormation. However, you still need to use the standard CloudFormation resource definitions for everything else. Apart from that, a benefit of SAM is the tooling which simplifies simplifying local development and testing.

For any application more complicated than a simple API and one or two Lambda functions, the SAM is too close to raw CloudFormation to be convenient.

Amplify

Amplify is yet another AWS framework that builds on top of CloudFormation. Or, at least the **Amplify CLI**, which is the part of the framework responsible for IaC.

Amplify provisions cloud services almost automagically. You select a resource to create, interactively choose options, and have everything needed generated. However, **customizing it can be hard or even impossible**.

In terms of IaC, Amplify may be suitable for small apps, proof of concepts, or for the frontend developers needing a working backend without spending too much time on it. Besides that, all simplifications and auto-generated resources may turn out to be more of an obstacle than a help.

Amplify also provides a set of frontend libraries to work with AWS services. You can use those libraries without having the infrastructure generated by Amplify.

Terraform

Terraform is a third-party IaC tool, not relying on the CloudFormation at all. Instead, it has its own logic for tracking the state and changes of the infrastructure.

Terraform has some advantages over CloudFormation, with the speed of execution being one of them. However, I prefer to rely on AWS to maintain and modify the infrastructure.

Part II
Services

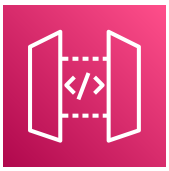
AWS offers over 200 services. Here I present the most useful serverless ones in 4 categories: API, Computation, Storage & Database, and Messaging.

With few paragraphs for each service, I only scratch the surface of their capabilities. It's supposed to help you choose the ones you want to use in your next application. Then, you will find far more details in the AWS documentation.

API

Most services expose HTTP API that clients (frontend, mobile, or other back-end applications) will use. For that, in the serverless world, you need an API service. Such service will receive requests and pass them to other services based on the request path, method, and/or content.

API Gateway



API Gateway comes in three flavors: **REST**, **HTTP**, and **WebSocket**.

Don't be misled by the naming - both REST and HTTP types serve the same purpose. They let you expose HTTP endpoints that trigger your integration, most often a Lambda function. **The HTTP API is a newer, cheaper solution with lower latency.** The REST API has a few more advanced features. If you don't need any of them - **always choose HTTP API**.

See: HTTP API vs. REST API features comparison

WebSocket API is self-explanatory. It accepts WebSocket connections and triggers Lambda function(s) on user connection, messages, and disconnection.

Pricing: per request.

AppSync

AppSync, as the name absolutely does not suggest, is a **GraphQL API service**. If you only did REST APIs in the past, on the internet you can find a list of good reasons to try GraphQL.

In AppSync, you provide a **GraphQL schema** and **resolvers** that define how to obtain particular objects and fields from the schema. Resolvers can trigger a Lambda function to process the request and return results, but they can also call services like DynamoDB directly.

The only drawback of AppSync is that resolvers are written using Velocity Template Language (VTL). Unfortunately, VTL is far from being the easiest or most-readable language. But the support of JavaScript resolvers is announced.

One of the advanced features of AppSync is real-time data synchronization using GraphQL Subscriptions.

Pricing: per request.

Computation

With serverless, you don't manage servers but still need a place to run your code. You have several options depending on the characteristic of the operations you want to do.

Lambda

Lambda, for many, is **the** serverless service. You put your code, set up an event that will trigger it, and do not care about anything more.

Of course, that's not entirely true. As with every technology, there is a number of things to understand and think about. They are just not related to infrastructure and provisioning. Those aspects include performance optimization, logging, and more.

What can trigger a Lambda? The most common examples are API Gateway and AppSync, with Lambda handling HTTP requests. But it can also be used

to process messages from an SQS queue, changes from a DynamoDB table, or almost any event from any AWS service. You can also trigger it on schedule with CRON expression.

The most commonly used Lambda runtimes are Node.js and Python. I personally prefer Node with TypeScript. If you choose JavaScript or TypeScript, you may find my AWS SDK Client mock library useful.

Pricing: per request + execution duration (with millisecond precision)

Fargate

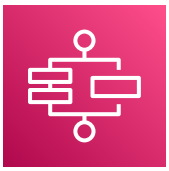
Fargate allows running Docker containers without thinking about servers. But how is it different from Lambda, which supports containers as well?

Fargate is for **long-running jobs**, not processing individual events. It takes a minute or two to launch a container, but then it can run for as long as needed. Basically, there are three types of applications for which you must choose Fargate over Lambda:

1. Applications that continuously poll for messages from external systems
2. Stateful applications, like WebSocket clients
3. Traditional backend applications that cannot be converted to Lambda functions

Pricing: per assigned vCPU and memory GB per second (for the whole duration the container is running)

Step Functions



Lambda functions are, well, functions. And as any function, they should be limited to a single responsibility. If some process consists of multiple steps, that's the place for Step Functions.

Step Functions allow **orchestrating workflows** consisting of various AWS calls and logic operations. Process steps can be executing Lambda functions, making Athena queries, sending messages to SQS, and anything else you can do with AWS SDK. Those actions' results can be passed to the next steps and affect what steps will be executed.

You can even wait for external events to happen to continue the workflow. For example, you can send an email and wait for the recipient to click the link to proceed to the next step.

Step Functions are also perfect whenever you start some job and have to wait minutes or even hours for its end before proceeding - think Machine Learning model training, Athena Big Data query, etc. This can be implemented in a simple workflow with a periodic check for completion before moving to the next step.

Step Functions are a **low-code solution** with visual execution overview and graphic editor. If serverless is about writing less code to build business processes, Step Functions are the next *step* forward after Lambda.

Pricing: per number of executed steps

Storage & Database

Every application needs to keep the data somewhere. Object storage to put files and a database to put data is a good start.

S3

S3 stands for Simple Storage Service. Whatever files you need to store, the S3 is the place.

You create **Buckets** and put **Objects** in them. You can put a virtually unlimited amount of data into a Bucket. You can upload and download files using AWS SDK and make files public to be accessible with a regular URL. This way, you can display images from your Bucket on your website or mobile application.

S3 also has a number of advanced options. For example, you can generate **presigned URLs** that allow secure upload or download of objects for a limited time. You can **automatically remove** old Objects or archive them to reduce storage costs. Or you can **host a static website** directly from the Bucket.

Pricing: per GB stored + per request

DynamoDB

DynamoDB is a NoSQL database. It gives you what is best in the serverless world - **low latency**, **high availability**, **unlimited scalability**, **pay-per-use pricing**, and requires **no administration**. It takes a moment to learn how to use it effectively, especially if you have only worked with SQL so far, but it's worth it.

DynamoDB tables keep **items**. Items do not require a predefined schema and are **represented as JSON objects**.

DynamoDB items consist of the Partition Key and attributes. Unlike SQL databases, where you can query data by any attribute, items from DynamoDB tables can only be retrieved using the Partition Key. While it sounds pretty limiting, there are patterns for designing DynamoDB tables to fit almost every OLTP use case. Moreover, you can define Global Secondary Indexes (GSI) that enable querying by selected attributes.

The tricky point of DynamoDB (and similar NoSQL databases) is that, while it does not require you to define items schema upfront, you must **start by listing your data access scenarios**. Then you **model the items schema** based on it.

Advanced topics:

- Adjacency List Design
- Single Table Design
- The DynamoDB Book

Pricing: per GB stored + per read and write request

Aurora Serverless v2

If you need an SQL database, the serverless option is Aurora Serverless v2. It's a relational database compatible with either MySQL or PostgreSQL.

The Serverless flavor of the Aurora **automatically scales up and down** depending on the usage. However, the v2 does not scale down to zero. That means, even with no queries, you will still pay for the minimal provisioned capacity.

Pricing: per GB stored + per I/O operations + per provisioned capacity per second

Messaging

Messaging services allow decoupling parts of the architecture - separate message producers from the consumers. Introducing messaging instead of directly invoking resources makes the processing asynchronous and our application event-driven. That approach fits nicely in the serverless architectures, where individual components can scale up and down separately, depending on the needs.

SQS

SQS is a Simple Queue Service. Multiple producers can push messages to a queue, and the consumers will take them off from it. There may be many consumers, although **only one consumer will read an individual message**. For that reason, the usual approach is to have a single application reading from the queue. However, you can parallelize processing with multiple application instances, like several Lambda function environments.

In SQS, **the consumer pulls the messages.** There is a long-polling option, where the consumer calls the SQS and waits until there are messages available or it reaches the max polling time. With Lambda as a consumer, this is done automatically under the hood.

Pricing: per request (push and pull)

SNS

While SQS is a queue of messages processed by a single consumer, the SNS - Simple Notification Service - allows sending messages to multiple receivers. Or, to be precise, subscribers because it's a **Publisher-Subscriber model**.

Contrary to the SQS, the SNS sends messages to the receiver without polling from the consumer.

With SNS, you can send messages to:

- Lambda function
- SQS
- HTTP endpoint
- mobile phone – via SMS
- mobile app – via push notification
- email
- Kinesis Firehose

You can optionally **assign filters to individual subscribers** to limit messages delivered to them based on the message attributes.

Pricing: per request + per delivery

EventBridge

EventBridge is an **event bus** for messages that you want to propagate across your (micro)services. Those events can come from state changes of **AWS services**, **other AWS accounts**, or **external applications** like Auth0 or Shopify. You can, of course, also send your custom messages.

Similar to SNS, also here we have a **publisher-subscriber model**. Each **subscriber sets filtering rules** to select what kind of messages he wants to receive. To give some examples, you may want to trigger a Lambda function whenever a specific Step Function execution finishes (event from AWS service) or a new user creates an account (custom event).

Apart from triggering Lambda functions, you can also send messages to several AWS services. That includes other messaging solutions discussed above – SQS and SNS.

EventBridge allows for **true systems decoupling**. With all other solutions, we usually create a single-purpose resource – for example, an SQS queue for specific messages that a single consumer would receive. In the case of EventBridge, on the other hand, we have a **single central event bus**, to which all the producers write, and all the consumers subscribe.

Pricing: per custom event + per subscriber invocation

Other services

As noted before, AWS offers much more services. While not all listed below are strictly serverless, they are general-use or Software as a Service (SaaS) solutions billed in the pay-per-use model. That makes them a perfect fit for serverless.

IAM (Identity and Access Management)
Core AWS service for access management. Every Lambda function you create must have an IAM Role assigned with proper permissions to make requests and use other AWS services.

CloudWatch
Application monitoring. It's the place where you will find logs and metrics from your Lambda function executions and other services' operations. It also includes alerting, advanced log analysis, and more.

Cognito
End-user authentication service. It handles user sign-up and sign-in for you, supporting using social media accounts like Google or Facebook. It integrates with API Gateway and AppSync to require user login to access APIs.

CloudFront
Content Delivery Network (CDN). You put it in front of an S3 bucket or API, and it can cache responses in geographic locations worldwide, close to the users, reducing the latency.

Route53
Domains and DNS management. Integrates with API Gateway, AppSync, and CloudFront to host your services under your domain names.

CodeBuild and **CodePipeline**
Continuous Integration and Delivery (CI/CD). Not the best in existence, but good enough if you want to have everything in one place.

Athena
SQL on files stored in S3. You create tables and just run queries. Perfect for Big Data analysis.

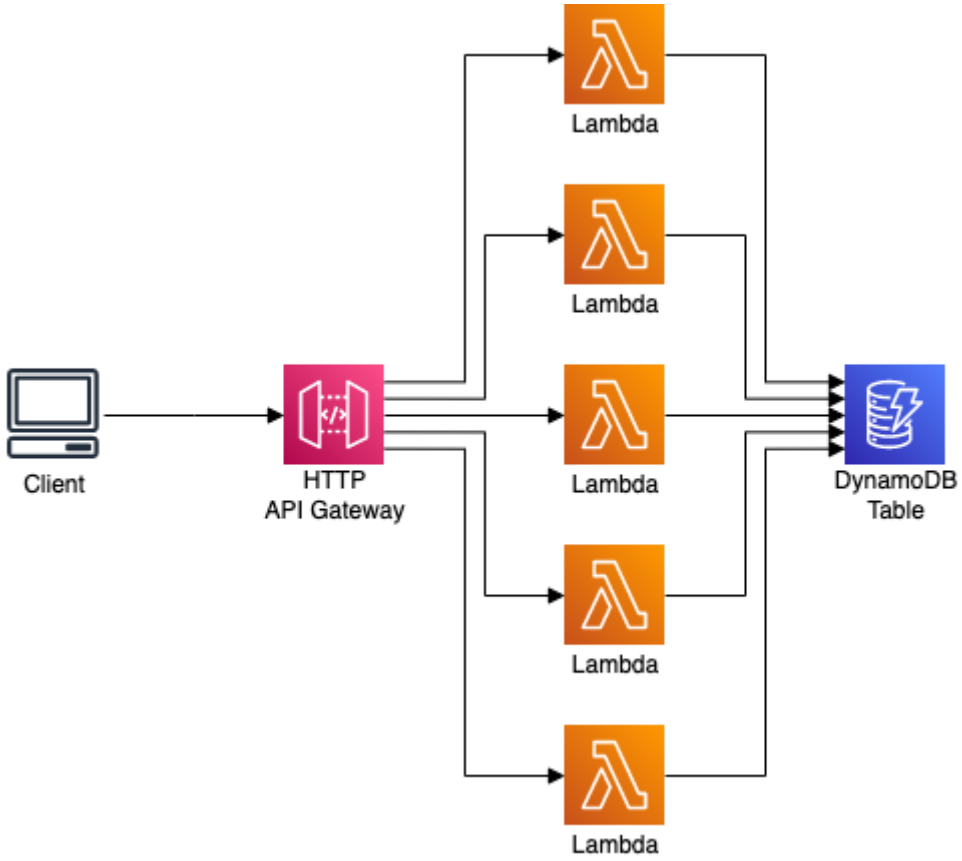
Machine Learning
Various machine learning services provide off-the-shelf solutions:

- **Forecast** - predict values based on historical data
- **Rekognition** - analyze images and videos to detect and analyze faces, text, objects, scenes, celebrities, and others
- **Comprehend** - analyze text to extract key points, recognize and categorize entities, define the sentiment, and more
- **Polly** - turn text to speech
- **Transcribe** - convert speech to text
- **Textract** - extract text from images
- **Translate** - translate text between over 70 languages

Part III
Architecture patterns

As with any other technology, also in serverless, you will find standard solutions for repeatable problems and tasks. Several architecture examples I present on the next pages should give you a good idea of how to build serverless services. Then you can mix and match those patterns according to your needs.

REST API



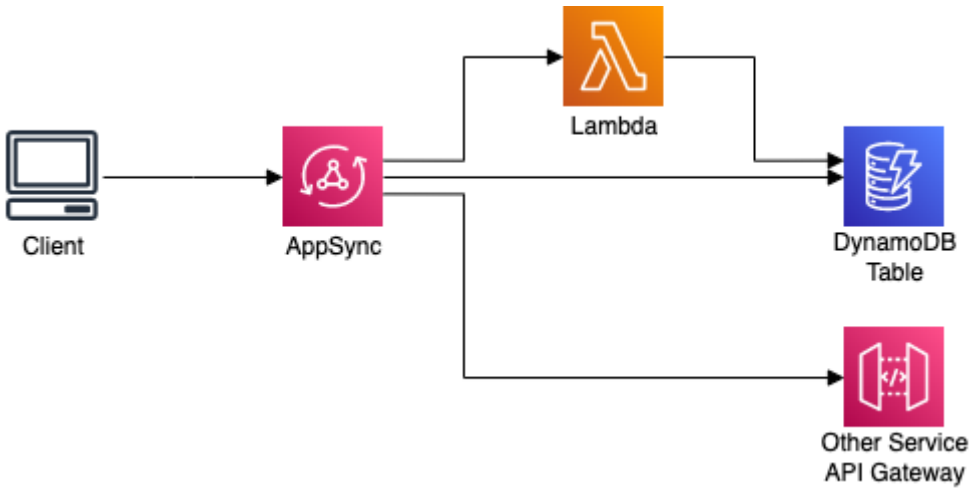
The most basic architecture example is an API with a database behind it. In serverless, you need three services: API Gateway, Lambda, and DynamoDB.

API Gateway exposes an URL and invokes **Lambda** functions to handle calls to different endpoints. Best practice is to create separate Lambda functions per endpoint (`/`, `/tasks`, `/tasks/1`, `/search`) and even HTTP methods (GET, POST, PUT, DELETE). That doesn't mean the functions can't share the code - they can reuse the common logic from the same codebase.

In most straightforward cases, Lambda functions are small, few lines long, and just make the requests to the database to get or update the data. But they can include any other required logic, make multiple calls to various data sources, etc.

Typically, the data is stored in a **DynamoDB** table. However, nothing prevents the Lambda functions from accessing other databases or APIs to fulfill the request.

GraphQL API



Setting up GraphQL API is a bit more complicated than REST AP.

Firstly, you create AppSync and provide it GraphQL schema. Then you define **data sources** and **resolvers** to describe how to obtain schema elements.

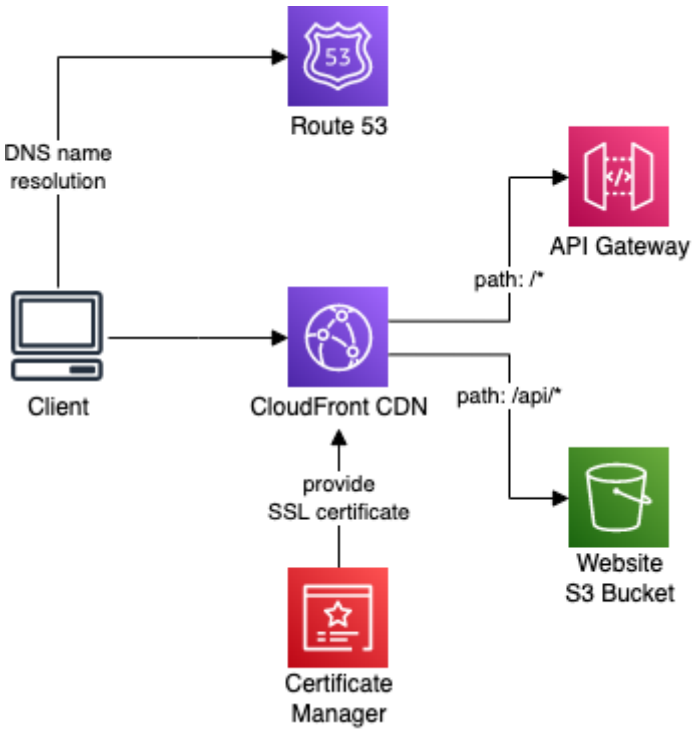
Resolvers parse the request, invoke the data source, and parse the response before returning it to the client. The simplest resolver can just pass everything as is to the data source and then return the response to the client without modifications. If you connect a Lambda function as a data source, you can have all the request processing logic in code in your favorite language.

However, not only Lambda can be a data source. For example, AppSync can make calls directly to:

- DynamoDB
- RDS (including Aurora)
- OpenSearch / Elasticsearch
- other HTTP endpoints

If possible, prefer making resolvers with those data sources instead of using Lambda everywhere. Writing resolver templates for DynamoDB calls can be more difficult than doing the same in TypeScript or Python but brings several benefits. Firstly, **no unnecessary Lambda invocations** that can increase costs. Secondly, **less code that can fail**. And finally, **reduced latency**.

Website with API



To host a static website or single-page application (React, Angular, etc.), all you need is to put it in an **S3 bucket**. Then, you enable **website hosting** on the bucket, and your website is online.

This, however, will give you only an HTTP address that you can't customize or upgrade to HTTPS.

Thus, you need something more. For more customization, you put a **CloudFront** in front of it and configure your S3 bucket website URL as an **origin** to fetch content from. This way, you will get an URL to your website supporting HTTPS. Additionally, since CloudFront is a CDN (Content Delivery Network), your website will load fast worldwide.

If you want to set your own custom domain, you can register one in **Route 53** and create an **alias record** pointing to your CloudFront. Then, to have your website served through HTTPS, you make an **SSL certificate** in **Certificate Manager (ACM)** and provide it to CloudFront. Interestingly, this certificate is one of the few things you can get in AWS entirely for free.

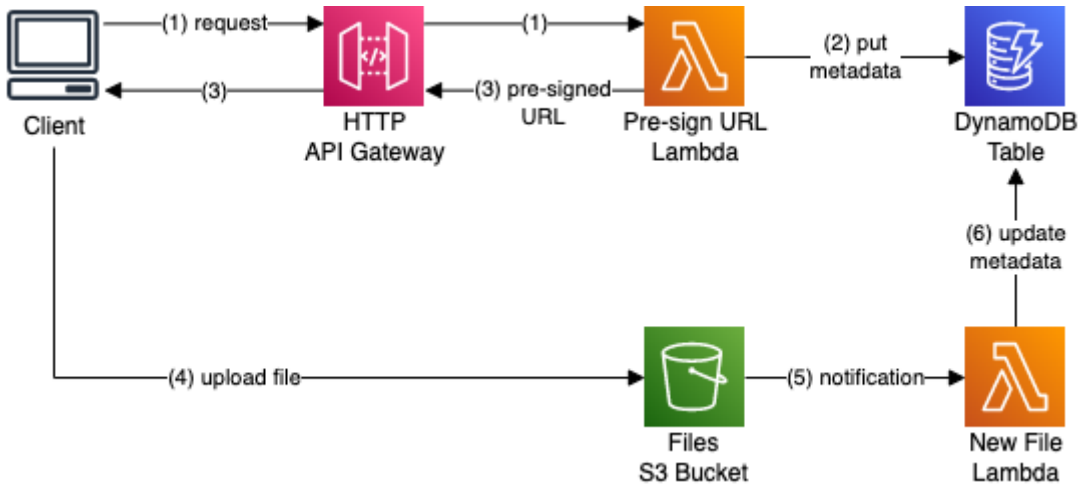
Now, most websites require an API to interact with. While you can put the API URL in the website code, there is a better way.

You can put your API Gateway or AppSync URL as another origin in CloudFront. You then create CloudFront **behavior** with path pattern `/api` to use this origin. Importantly, you must put it in order before the default behavior pointing to the website bucket.

As a result, calls to `/api/*` will be forwarded to your API, and everything else will serve your website. Instead of a hardcoded API URL, the website can have just `/api/...` as a request URL. Both the website and the API will be under the same domain.

As a bonus, you can have another S3 bucket(s) with resources like user-uploaded images, also attached to the CloudFront under another path pattern, for example `/images`, to load them on the website with URL like `/images/01.png`.

File upload



As you already know, the S3 is the service to store any objects and possibly share them on the internet. And those include files uploaded by the users.

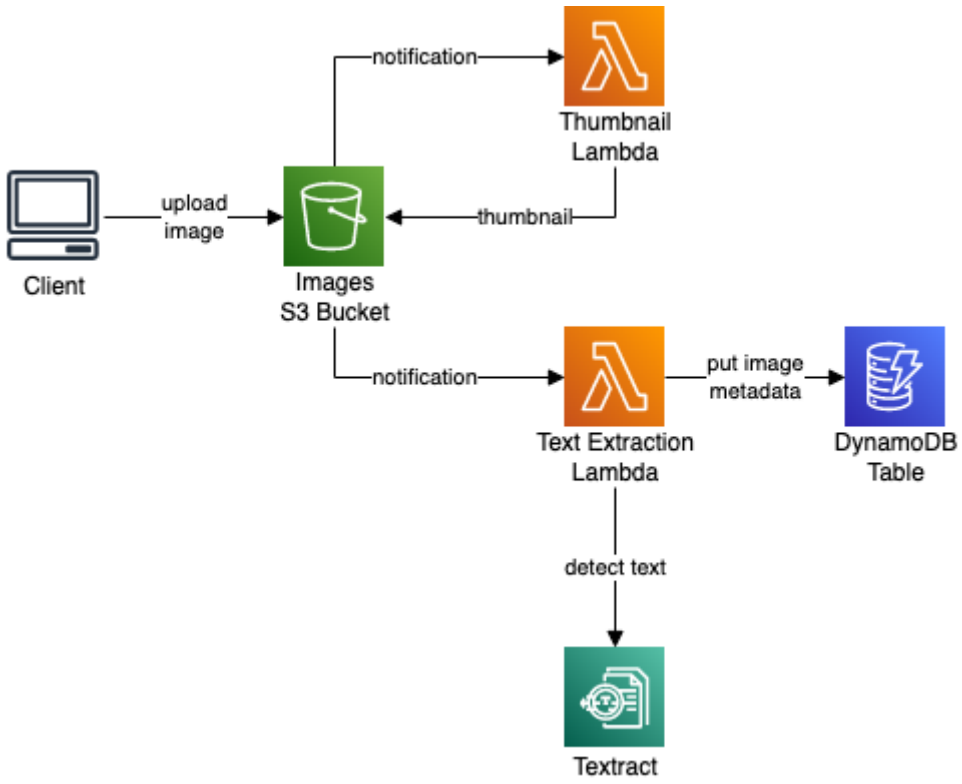
Unlike with traditional applications, to upload a file to S3, you don't push it to the API endpoint. Instead, you firstly make a request to the API to get a **pre-signed URL (1)**. A Lambda function then puts metadata in a DynamoDB **(2)**, saving information like user ID, file name, and upload status "in progress". It then generates the upload URL and returns it to the client **(3)**.

The pre-signed URL is an address with an access token included, and it allows uploading a file to an S3 bucket. It will work only for uploading a file under a specified name to the selected bucket. The token is short-living, so it will be possible only for a defined time.

Having the URL, the client uses it to upload the file **(4)**.

In the S3 bucket, you set up an event notification to trigger a second Lambda function when a new file is uploaded **(5)**. There, you can extract additional metadata from the file and update the item in the DynamoDB **(6)**, marking its status as "uploaded".

Asynchronous processing



Extending the previous file upload example, an example of a simple asynchronous process can be creating an image thumbnail. After the image is put in S3, the bucket sends an **event notification** to a Lambda function. The function fetches the image and creates a smaller version of it to be used as a thumbnail. It then uploads it back to the same or different bucket.

If you are putting thumbnails to the same bucket, **be careful to not create an infinite loop** with Lambda putting a new file in the bucket and being triggered by it back, thus creating another file, and so on. This can be very costly. To avoid that, define the object name **prefix or suffix filter** in the S3 event notification configuration.

Nothing stops you from making this more complex and adding other asynchronous operations. For instance, you can create another S3 bucket event notification to trigger a second Lambda. It can extract text from the image using the Amazon Textract service and then add it to image metadata in the DynamoDB table.

Part IV
Various advice

You will find many good practices on my and other blogs. However, if you are beginning with serverless, this section contains the essential tips on building serverless applications.

Account security

Every other week someone asks on r/aws what to do after having the account compromised and receiving a bill for thousands, dozens of thousands, or tens of thousands dollars.

While the Reddit community is helpful, it's best to not having to ask at all.

Most importantly, setup **multi-factor authentication (MFA)** for your account root user and all IAM Users.

Secondly, **never commit your access credentials** (access key ID and secret access key) to repository. In fact, you shouldn't use those credentials in any place other than your computer to access AWS from the CLI. For all programmatic purposes, use IAM Roles.

Those are the most basic rules of AWS account security. While securing your account could be a topic for another not-so-short ebook, those two are critical to follow.

Budget alerts

An unexpectedly high bill at the end of the month can mean multiple things. For example, your application became popular, your account got hacked, you forgot to shut down a costly service you were exploring, or you just screwed out in one of the many ways possible.

A high monthly bill can mean something generates unnecessary costs for even 30 days. You can reduce this threat significantly by setting budget alerts.

AWS does not allow setting a monthly cap for your spending. The only thing you can do is set up your **budget** in the **Billing Console**. Then, you set thresholds for which exceeding you want to receive **email alerts**.

While alerts are not immediate and your costs can still exceed the budget, it's far better to get this info the next day, not after a week or two.

Region selection

When selecting the AWS region to host your application, the first thing that comes to mind is **latency**. You want to keep your services near the users. If your target audience is in Europe, you should select one of the European regions.

Users from other continents will have higher latency accessing your application. You can mitigate it with CloudFront or, if the management overhead is justified, launch your service in multiple regions.

But latency is not the only factor. All regions are equal, but some regions are more equal than others. And for sure cheaper and with more features.

AWS **pricing** differs between regions. USA regions have the same or similar, low prices. But the pricing can be almost 2x higher in Sao Paulo, currently the most expensive region. Usually, older regions are cheaper.

Before deciding on the region, compare prices for services you plan to use between a few regions. Latency differences between close regions can be minimal, but cost differences can accumulate exponentially.

And lastly, not all regions have the same offerings. New **features**, especially those depending on the hardware, tend to be available first in, again, the oldest regions. Some services are not available in some regions for a long time.

Free Tier

Most of the services described have a Free Tier offer. For 12 months after creating an account, and in some cases indefinitely, you have some resources available for free up to defined limits.

For example, the Free Tier includes:

- 5 GB of storage, 20.000 Get Requests, and 2.000 Put Requests monthly on **S3** for the first year
- 1 million requests and 400.000 GB-seconds of compute time for **Lambda** per month - forever

Tip: if 12 months period comes to an end, nothing stops you from creating a new account. Just make sure you remove all the resources from the old account not to get a surprise bill for them!

Testing

There are several solutions to emulate AWS services locally. Don't use them. You will spend time configuring them to work with your code and each other, but they will never properly imitate the cloud. Also, it can take months until new features released by AWS come to your emulator, and only if those features are popular enough.

Instead, **test in the cloud**. With serverless, it's easy and cheap for every developer to have his own environment deployed. Testing in the actual cloud gives you the closest possible simulation of how the services will act on production.

But deploying changes always takes time. Waiting even 30 seconds after every change breaks "the zone". That's why tests are so crucial in serverless.

Write **unit tests** to test your Lambda functions logic. Lambda functions should be relatively short and simple, so should be unit tests. Unit tests can

run even under a second, providing fast feedback. Thus, writing proper unit tests greatly minimizes the number of re-deployments needed.

To simplify unit testing in JavaScript/TypeScript, see my AWS SDK v3 Client mock library.

Write **integration/system/e2e tests** to test your solutions. Serverless applications are compounds of multiple small pieces that need to work properly with each other. Integration tests should run in the cloud on the solution deployed to the dev/test/staging environment. By invoking the API and expecting valid results, they test the correctness of system logic and the configuration that holds all the parts together.

Architecture diagrams

A picture is worth a thousand words. This is especially true for the architecture of serverless solutions built from dozens of resources.

Use whatever tool suits you, as long as you keep your diagrams in sync with reality. I use diagrams.net - it's free, simple, and added to almost every corporate Confluence. For the latest resource icons, check out an up-to-date AWS icons library I maintain.

Takeaways

1. Use CDK or Serverless Framework for smaller projects
2. Consider switching from REST to GraphQL with AppSync
3. For computation:
   a. Process events with Lambda
   b. Orchestrate with Step Functions
   c. Use Fargate if none of the above suits the case
4. Get to know DynamoDB
5. Decouple services with asynchronous messaging
6. Enable multi-factor authentication
7. Set up budget alerts
8. Test in the cloud - don't bring cloud to you, bring you to cloud
9. Document your architecture with diagrams

Thank you for reading.

If you found this ebook useful, let your colleagues know they can get it for free from BetterDev.blog.

If you have any thoughts or comments, don't hesitate to message me at betterdev.blog/contact/

Since you have this ebook, you have already subscribed to my newsletter. So why not follow me on Twitter too?