SERVERLESS ARCHITECTURES: OUR EASY WAY TO COST OPTIMIZATION.

Amazon API Gateway | Kibana | Serverless

beSharp | 12 June 2020

We are in the middle of the **serverless age**, in a moment where the advantages of this approach become more and more evident.

One of the advantages that contributed the most to the rise of the serverless paradigm is certainly the promise of significant **economic advantages**.

But are you really saving money? And: are you taking full advantage of the serverless paradigm?

What is serverless?

Serverless is a native cloud architecture that enables you to shift more and more of your operational responsibilities to the cloud provider, in this case, AWS, increasing your **agility** and **innovation**. Serverless allows you to build and run applications and services **without caring about servers** and thus eliminating infrastructures management tasks such as servers or cluster provisioning, patching, operating system maintenance, and capacity provisioning. Serverless helps you run and scale your application while **high availability** is provided transparently by the services you use as building blocks.

While Designing the architecture of a serverless application, you need to think about different topics, including **storage**, **database**, **computing**, **messaging**, and many more.

AWS offers plenty of services you can use to make a valid serverless application. As you need to pick the right tool for each scope, the knowledge of serverless services is vital to make wise choices.

Not only Lambda

When thinking about serverless the first AWS service that comes to mind is for sure AWS Lambda, indeed the **computing** part of a serverless application is probably the most iconic and important

one. However, a serverless architecture is not composed only by lambdas, but also by different **AWS managed services** which in turn must be serverless.

Here is a brief and not exhaustive list of serverless and fully managed services for each tier of an application.

Compute

Of course, we have, AWS Lambda and Lambda@Edge, which are usually associated with serverless and attract the majority of frameworks and thirty parties solutions, but also

AWS Fargate, that is a purpose-built serverless compute engine for containers.

Storage

Amazon Simple Storage Service (Amazon S3) provides a secure, durable, highly-scalable object storage.

Database

Amazon DynamoDB is a fast and flexible NoSQL database service for all the applications that need consistent, single-digit millisecond latency at any scale.

Amazon Aurora Serverless is an on-demand, auto-scaling configuration for Amazon Aurora. The database will automatically start-up, shut-down, and scale capacity up or down based on your application's needs.

API Proxy

Amazon API Gateway is a fully managed service and offers a comprehensive platform for API management with support for authorization, access control, monitoring, and API version management.

Application integration

Amazon SNS is a fully managed pub/sub messaging service that makes it easy to decouple and scale microservices, distributed systems, and serverless applications.

Amazon SQS is a fully managed message queuing service that makes it easy to decouple and scale microservices, distributed systems, and serverless applications.

AWS AppSync simplifies application development by letting you create a flexible GraphQL API to securely access, manipulate, and combine data from one or more sources.

Orchestration

AWS Step Functions makes it easy to coordinate the components of distributed applications and microservices using visual workflows. Step Functions is a reliable way to coordinate components and step through the functions of your application.

Analytics

Amazon Kinesis is a platform for streaming data on AWS, offering powerful services to make it easy to load and analyze streaming data, and also providing the ability for you to build custom streaming data applications for specialized needs.

Amazon Athena is an interactive query service that makes it easy to analyze data in Amazon S3 using standard SQL. Athena is serverless, so there is no infrastructure to manage, and you pay only for the queries that you run.

AWS Lambda: pricing model and pitfalls

When creating a serverless application, it is important to keep in mind its **pricing model** in order to structure the services and triggers in a way that allows you to achieve **the best performances at the lowest price.**

AWS Lambda has a really interesting pricing model if used in the right way; also, you can leverage a generous **free tier** of 1M free requests per month and 400,000 GB-seconds of computing time per month. This free tier does not expire after the first year.

The Lambda pricing model is all about **invocations and execution duration**.

Lambda counts a request each time it starts executing in response to an event notification or invoke call. The duration is calculated from the time your code execution begins to the moment it ends, rounded up to the nearest 100ms by excess. This means that the execution time is paid “in steps” of 100ms. Having a function whose average execution time is very close to the threshold can thus considerably increase the overall execution cost.

The price of each 100ms “time step” depends on the amount of memory you allocate to your function. An increase in memory size triggers an equivalent increase in the computational power available to your function.

Requests: $0.20 per 1M requests

Memoria (MB)	Prezzo per 100 ms
128	$0.0000002083
256	$0.0000008333
1024	$0.0000016667
1536	$0.0000025000
2048	$0.0000033333
3008	$0.0000048958

The following facts can thus be deduced from the pricing model:

- Optimizing the **execution time** is crucial, the code must make the best use of resources and terminate as soon as possible.
- The **Lambda size** affects response time and overall costs. We need to balance the cost with the response time, keeping in mind that the price is for a step of 100ms.
- A lambda function should never be waiting for something: **triggers, queues, step functions** should be used to decouple the services and invoke the Lambda only when necessary.

Another common pitfall is to process messages from SQS at batches of size 1. If the queue is full you want to use the same invocation to process as many messages as possible because each invocation has a start-up time and a loading time which add-up pretty quickly if you process just 1 message at a time.

AWS API Gateway

API Gateway offers many features, which can be invaluable to **speed up the development** of complex applications and it is often regarded as the natural solution for Lambda-based serverless back-ends. However, the price of each API call is 5 to 15 times the price of a lambda invocation and while this price is adequate for the features of the service (e.g. authentication management, caching, request/response transformation, proxying), it is easy to understand that if these functionalities are not used ApiGateway is probably not the best choice.

There are several situations where API Gateway is used just as a transparent proxy to pass the entire HTTP request to a Lambda function which then manages authentication, authorization, parsing, and response generation. In these cases however, there are often cheaper solutions: for example, an Elastic Load Balancer can also be used to forward HTTP requests to Lambda, and can be more affordable than API Gateway at high volumes.

When your front-end is the only consumer of the backend APIs, you can leverage Cognito to obtain IAM credentials for your users. They can then invoke your lambda functions safely using the AWS Javascript SDK and the temporary credentials. This solution allows you to achieve both authentication and authorization via IAM, avoiding the need for an API Gateway at the cost of reduced functionalities (you have to prepare your request on the frontend side, and also manage responses and errors in your lambda code). Also, you can't attach a WAF to your backend, so this might not be always possible.

In essence either take advantage of the high-level features of the ApiGateway or try not to use it in order to avoid wasting money.

Distributed logging system

Logging is another area where you can save money without losing functionality.

By default, **CloudWatch Logs** is used to collect the logs of all the invocations of the lambda functions.

The solution is perfect for getting started, and the cost increases proportionally with the number of GB of logs generated, starting from 0$.

However, the cost per GB is quite high, and you may be surprised to see how many GB of logs are generated by a typical serverless web application. There are thus cases where the cost of this logging system can become excessive and difficult to justify.

So how can you optimize the cost of logging?

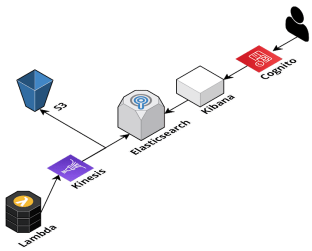
First, you need to reduce the number of logs generated as much as possible, it is almost always convenient to summarize the same information in a more compact format.

It is always best to avoid long, verbose messages and log only checkpoints or important events during execution, preferably just a few times during a single invocation.

It is also advisable to save the log messages in a structured format, such as JSON, and to include in a single message all the information necessary to reconstruct the event that caused it.

There are also cases where CloudWatch Logs is simply not advanced enough to allow adequate analysis of problems or exceptional situations. In these cases, you can benefit from a logging infrastructure created with Kinesis Firehose for log streaming, ElasticSearch, and Kibana for consultation.

This solution has a rather high fixed cost, but which tends to remain constant and in some cases (when the amount of log messages is stable and high) can be much lower than the cost generated by CloudWatch Logs.



These are only some of the tricks which can be used to **optimize the costs of a serverless application**, obviously, there are many more particular cases and specific scenarios that can benefit from the most disparate infrastructure customizations.

In general, on AWS it is important to take into consideration the pricing model of each service and build both the infrastructure and the application in order to take advantage of them rather than pay more to use the services in a non-optimal way.

Do not miss the next articles for further tips on the subject!

beSharp

Dal 2011 beSharp guida le aziende italiane sul Cloud. Dalla piccola impresa alla grande multinazionale, dal manifatturiero al terziario avanzato, aiutiamo le realtà più all'avanguardia a realizzare progetti innovativi in campo IT.

Get in touch

beSharp.it
proud2becloud@besharp.it