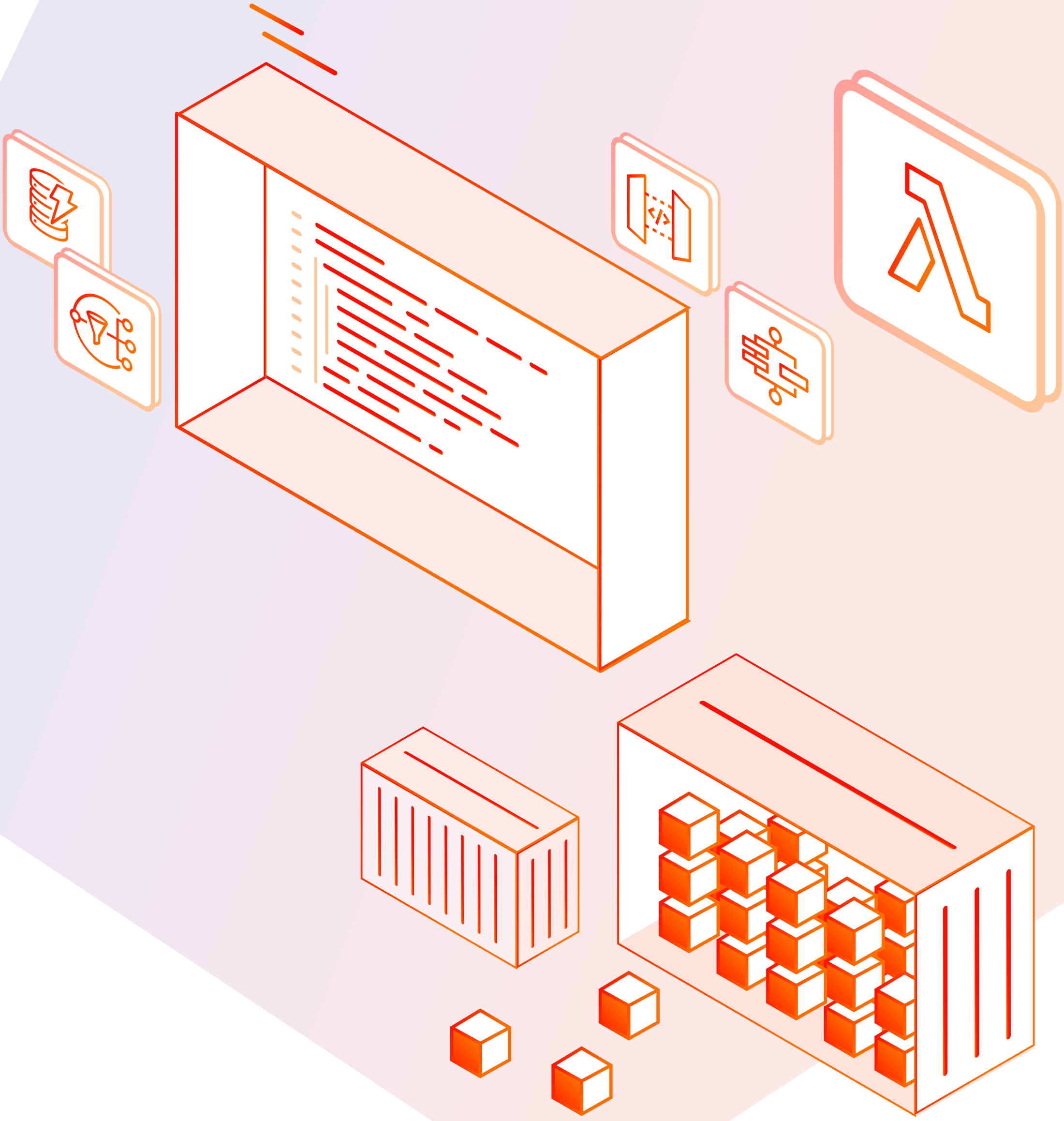


How to choose a **computing platform for microservices**

lumigo

Choosing the best computing platform for your microservices

AUTHORS

Orr Weinstein

VP of Product, Lumigo

Michele Mancioppi

Product Manager, Lumigo

Thorsten Hoeger

AWS DevTools Hero and Cloud Automation Evangelist, Taimos

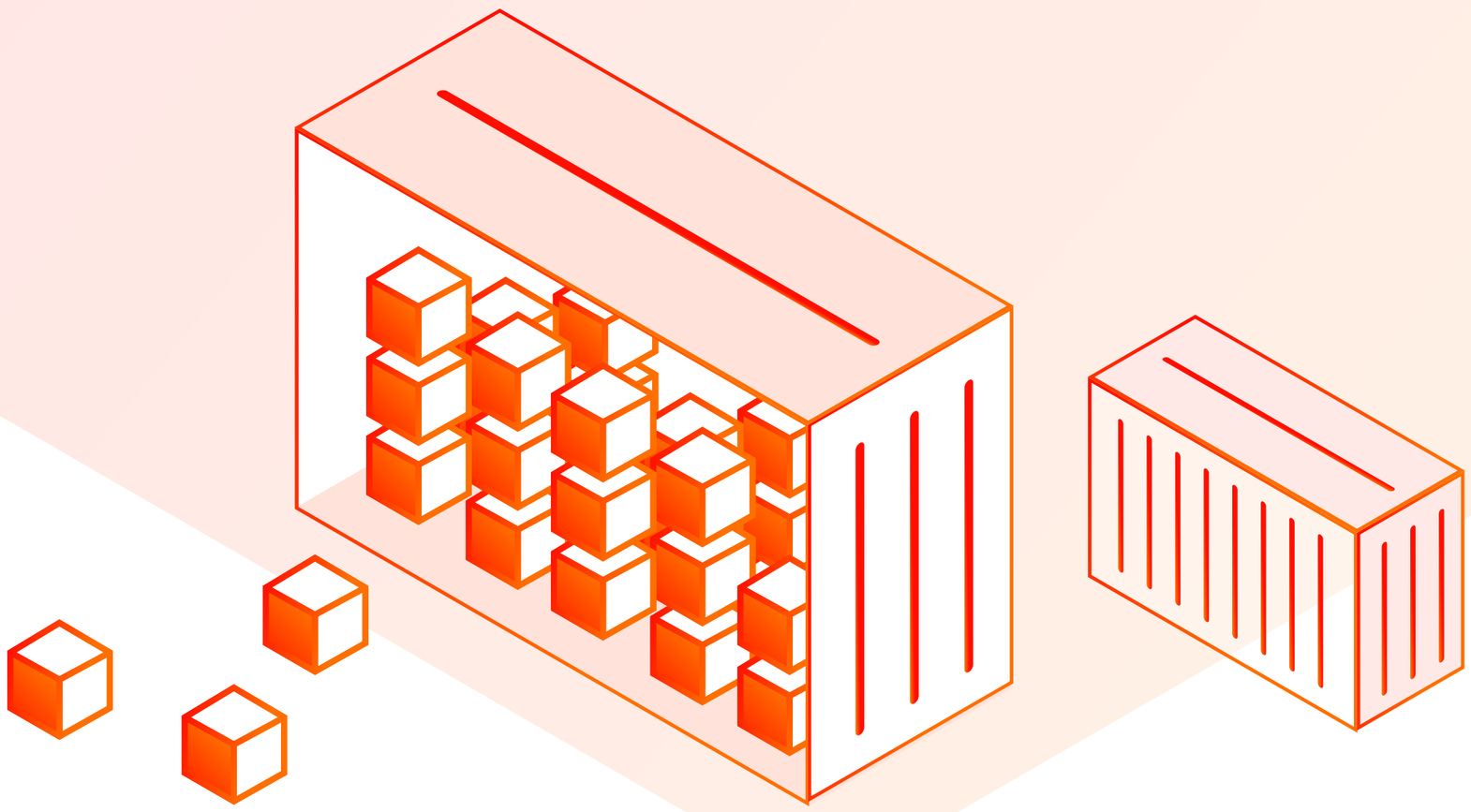
Introduction

Both containers and serverless computing let developers ship production-ready applications on the cloud faster and with less operational overhead than other means, like virtual machines or bare metal.

Given the application you want to build, determining the suitable architecture and where to run it is a complex task that must consider a myriad of aspects and tradeoffs. From a very high-level perspective, serverless offerings have proven suitable to build and operate scalable and resilient applications at a lower cost than containers.

Serverless, however, also has cost considerations at scale and is harder (or sometimes unfeasible) to repurpose existing software. Additionally, developer skills acquired on more traditional platforms are less portable to serverless than to containers.

The following will provide insights into containers and serverless computing to help you select the right computing platform for your next microservice distributed application.



What is a Container?

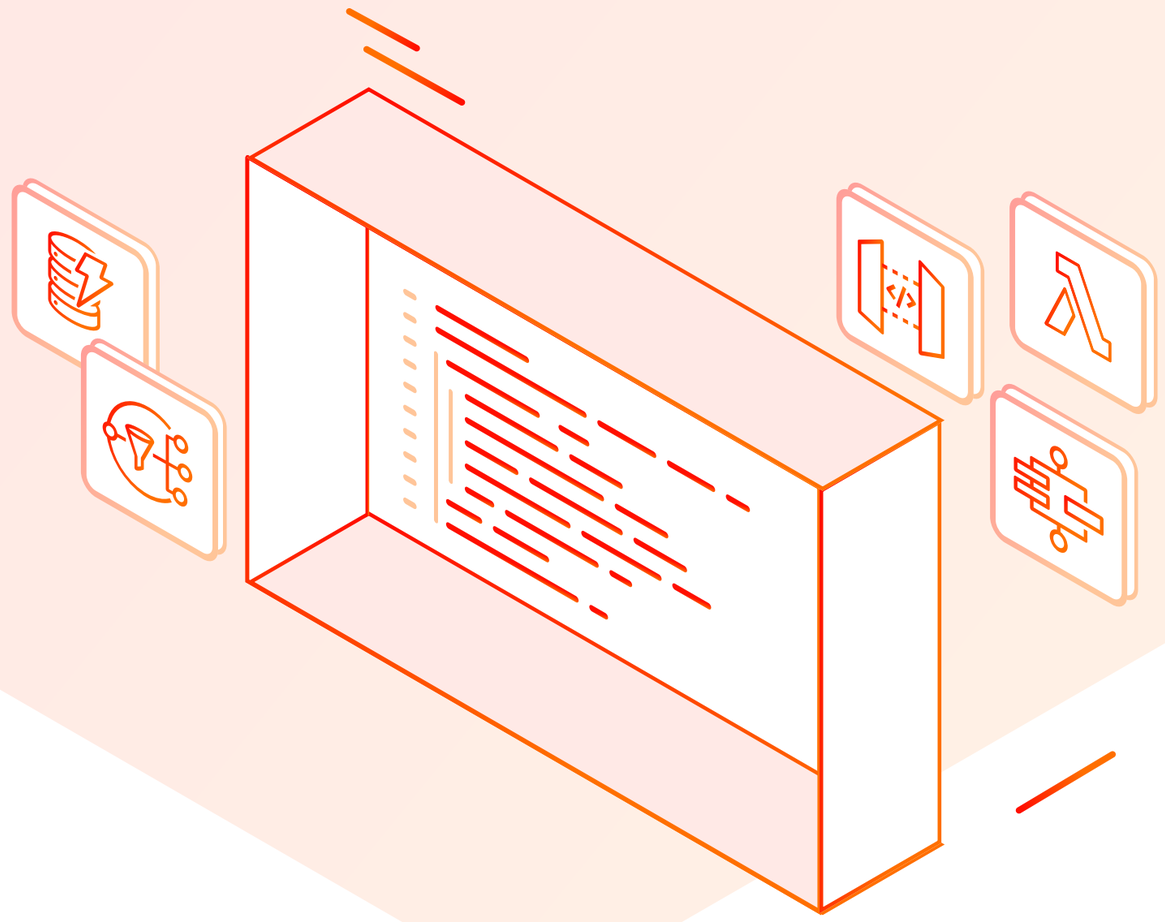
A container is a lightweight package comprising the application, all third-party dependencies, and configuration settings (versions of the language runtime) that the application needs to run successfully. In other words, the main goal of a container is to be almost entirely self-contained. Developers can then deploy these containers in a variety of environments with a high likelihood that the application will behave there the same as on the developer's own machine.

Over the years, managing containers in the cloud has become dramatically simplified. Until a few years ago, developers had to configure everything from scratch. This included setting up and maintaining virtual machines—EC2 for instance—and the orchestration tools such as Docker Swarm or single-Docker engines.

Nowadays, a variety of cloud vendors provide managed container services like Amazon Elastic Container Service (ECS) and Amazon Elastic Kubernetes Service (EKS), which offer out-of-the-box orchestration with different APIs and capabilities.

Some of the container orchestration services still require developers to deal with the underlying VMs in terms of sizing and maintenance, and details of the networking setup like security groups, VPC configurations, and more. To further reduce the overhead of running containerized applications, cloud vendors have been introducing container orchestration offerings.

Services such as AWS Fargate (which is a launch type of ECS) and AWS AppRunner present a "serverless" approach to managing containers without worrying about the underlying infrastructure (VMs and a considerable amount of networking), while providing largely the same capabilities as Amazon ECS. Similarly, Amazon EKS offers a mode where the containers it manages are running on top of AWS Fargate, rather than on an Amazon EC2 host managed by the end user.



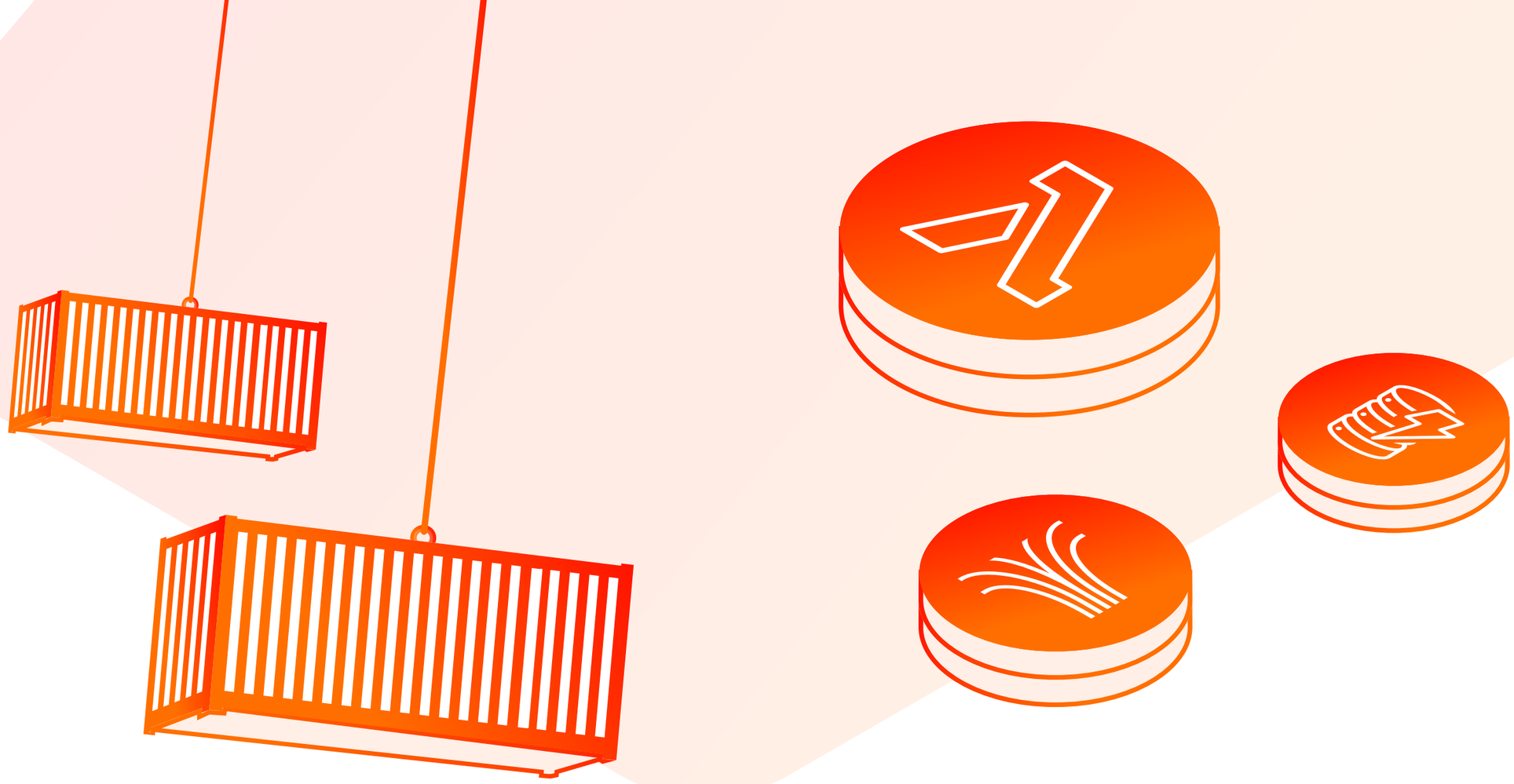
What is Serverless Computing?

Serverless computing strikes a different balance than container orchestration, one even more skewed towards “think only about your code”. Serverless offerings like AWS Lambda enable developers to build scalable, event-driven applications in the cloud without worrying (almost at all) about the underlying infrastructure. Instead, the cloud provider manages all the infrastructure and its scaling while the developer focuses on writing code for business problems.

Serverless computing is becoming more and more general-purpose. In the realm of image/data manipulation, AWS Lambda was suitable only for small tasks due to limits in maximum amount of memory (1 GB), no guaranteed CPU amount, disk space (500MB), and running time (1 minute). Also, it supported a limited number of programming languages, starting with just Node.js.

As time went by, limitations disappeared or significantly lessened. As of June 2022, AWS Lambda supports up to 10GB of memory, 6 vCPUs, 10 GB of storage, and a maximum running time of 15 minutes, meaning that developers can perform much more resource-intensive operations than before. It can run code in a variety of programming languages like Python, Java, Ruby, .NET, and even entire Docker images, where the runtimes and applications are entirely up to the developers.

Developers can deploy containerized applications directly on AWS Lambda and obtain the same pay-as-you-use, fully-managed and scalable service, thus bringing serverless computing and containers closer.



Containers vs. Serverless: The Differences

Though containers and serverless are more closely related to each other than they used to be, they have critical differences.

01 Cold starts vs. Scale-up time

Serverless compute services are prone to cold starts. A cold start occurs when an entirely new instance of the serverless function needs to be created to serve an incoming request, and that leads to increased latency in serving the request. For workloads that are sensitive to latency, cold starts can be an issue and, as such, there has been significant analysis of cold start times.

The issue was significant enough for AWS Lambda to offer the Provision Concurrency capability, where one can sacrifice the “scale-to-zero” nature of AWS Lambda—where nothing is paid if no requests are being processed. This reduced cold starts by having a fixed amount of “idle” instances around to handle bursts of requests.

In the world of containers, rather than cold starts, developers must think in terms of “time to scale”, meaning how long it will take to spin up more containers. Unlike AWS Lambda functions, one container can usually serve multiple requests in parallel, with cases of tens of thousands and more when using refined frameworks like Netty. Nevertheless, bursts of activity may require more running containers—and fast. The speed of scaling up containers has been improving over the years, but AWS Lambda is still the gold standard when it comes to scaling up and down AWS services.

02 Timeouts

AWS Lambda comes with very precise expectations in terms of the maximum amount of time it takes for a request to be answered. By default, it is three seconds and can be changed to a maximum of 15 minutes. This means that if answering requests is going to take more than 15 minutes, the software must be re-architected or sped up, or a different compute service must be selected.

Containers, on the other hand, leave it to the developers to decide how to spend CPU resources. Most container orchestrators will be perfectly happy with your containers sitting there idle for as long as you want them to and will take however much they need to serve requests.

03 Opinionation

Serverless, by design, is a more opinionated platform than container orchestrations. AWS Lambda has a very specific set of restrictions for how the code will be run: Each instance serves one request at a time and within a certain timeframe (see the Timeouts section). A well-defined lifecycle and facilities to extend available data to the application with layers—which effectively makes more files available to a Lambda function in a way that is reusable across many functions—can also be provided by 3rd-parties.

Layers can enhance the capabilities of the runtime with extensions that provide capabilities like running daemons (i.e., background processes) side-by-side with your application to, for example, collect additional monitoring information like metrics, logs, and distributed traces.

Applications running inside AWS Lambda, despite generally being in a Linux environment, have access restricted to several low-level facilities, like networking stacks and other mechanisms that are sometimes useful for very advanced scenarios. These restrictions, on the other hand, are the foundation for AWS Lambda to take away from you almost all the work of managing the scaling of your workload to handle incoming traffic.

Containers, again, leave much more freedom to developers. Of course, some restrictions change from container orchestrator to container orchestrator, but there are generally fewer than AWS Lambda, as the container is almost entirely the responsibility of the developers.

04 Event orientation

The perhaps most powerful aspect of AWS Lambda is its integration within the AWS ecosystem. There is a truly impressive number of AWS services that can invoke AWS Lambda functions, or at least have first-class integrations with AWS Lambda. At Lumigo, we use plenty of AWS Lambda functions in conjunction with Kinesis streams and DynamoDB streams. Among the many other use cases, some of these authors' favorites (and maybe less conventional) are rotating secrets and transforming data to and from an S3 object.

The event orientation, by the way, is not limited to the AWS ecosystem of services. Recently we came across a project that has GitHub Action runners on top of AWS Lambda that, when the limitations fit the needs, seems an incredibly useful way to scale out one's build infrastructure elastically and effortlessly.

Most of the integrations available for AWS Lambda can be replicated in containers, but with significantly more programming effort, and without the effortless elasticity that AWS Lambda provides.

05 Stateless vs. Stateful

In the realm of software engineering, an application is considered to be "stateful if it is designed to remember preceding events or user interactions" [Wikipedia]. For example, imagine storing the list of tasks to perform only in the memory of your application: that application holds "state" (which tasks to execute). Having state in your applications that you can't lose (e.g., the orders posted by a customer) and that are not kept also in some database or other type of reliable persistence is overall a bad idea, as important data is lost upon crashes and other mishaps.

Stateless applications are not natively designed to hold state in them. You may keep a copy of the data in the memory of your processes for optimization reasons, but those are effectively caches, and the normative data (the "source of truth") lies elsewhere in your persisted data. Serverless embraces statelessness, in that your AWS Lambda functions offer you no guarantee to keep state. Instances, and the state they contain, can be discarded at the whims of AWS Lambda.

This statelessness of Lambda has the interesting implication that, in reality, most AWS Lambda functions start their operations by retrieving state from persistent services, but overall, like a programming model, it seems to lead to far fewer data losses. Bringing this further, serverless experts advocate for the "storage-first" model, in which "data is stored persistently before any business logic is applied".

This greatly reduces the risk of losing or corrupting data. Containers, on the other hand, can have a stateful programming model. You could store important state in your containers and nowhere else. (Which, for the avoidance of doubt, is generally not a good idea!) The ease of achieving statelessness in containers is strongly related to the availability of integrations for particular frameworks and libraries, and how much development work is necessary to leverage them.

How to deal with state in containerized applications is a very nuanced topic where details make a lot of difference. But generally, there is a consensus in the industry that, if your application can be made in such a way, statelessness is better than statefulness.

06 Language support & Runtimes

In terms of language support and runtimes, AWS Lambda used to be more restrictive than containers. However, with the addition of custom runtimes, followed by the introduction of support for functions packaged as container images, developers can ship pretty much any runtime they want on AWS Lambda as they can in containers.

Granted, creating a custom runtime or packaging a function in a container image for AWS Lambda requires non-trivial, bespoke work to integrate with the opinionated AWS Lambda runtime (see the Opinionation section), but it is feasible overall.

07 Access to hardware capabilities

Despite the leaps and bounds achieved by AWS Lambda in terms of computing resources made available to functions (see the What is Serverless Computing? section), the choices of hardware capabilities to underpin AWS Lambda functions are limited. In terms of CPU architectures, AWS Lambda currently supports the x86 and arm64 instruction sets, and it has some optimizations for x86 to better support CPU-intensive tasks. However, when running a container orchestration with control on the underpinning host – say, Amazon ECS with EC2 launch type –, there is a very large variety of virtual machine images available to best power workloads that, for example, need access to GPU resources, which are often necessary for image processing, machine learning, and other computation-intensive workloads.

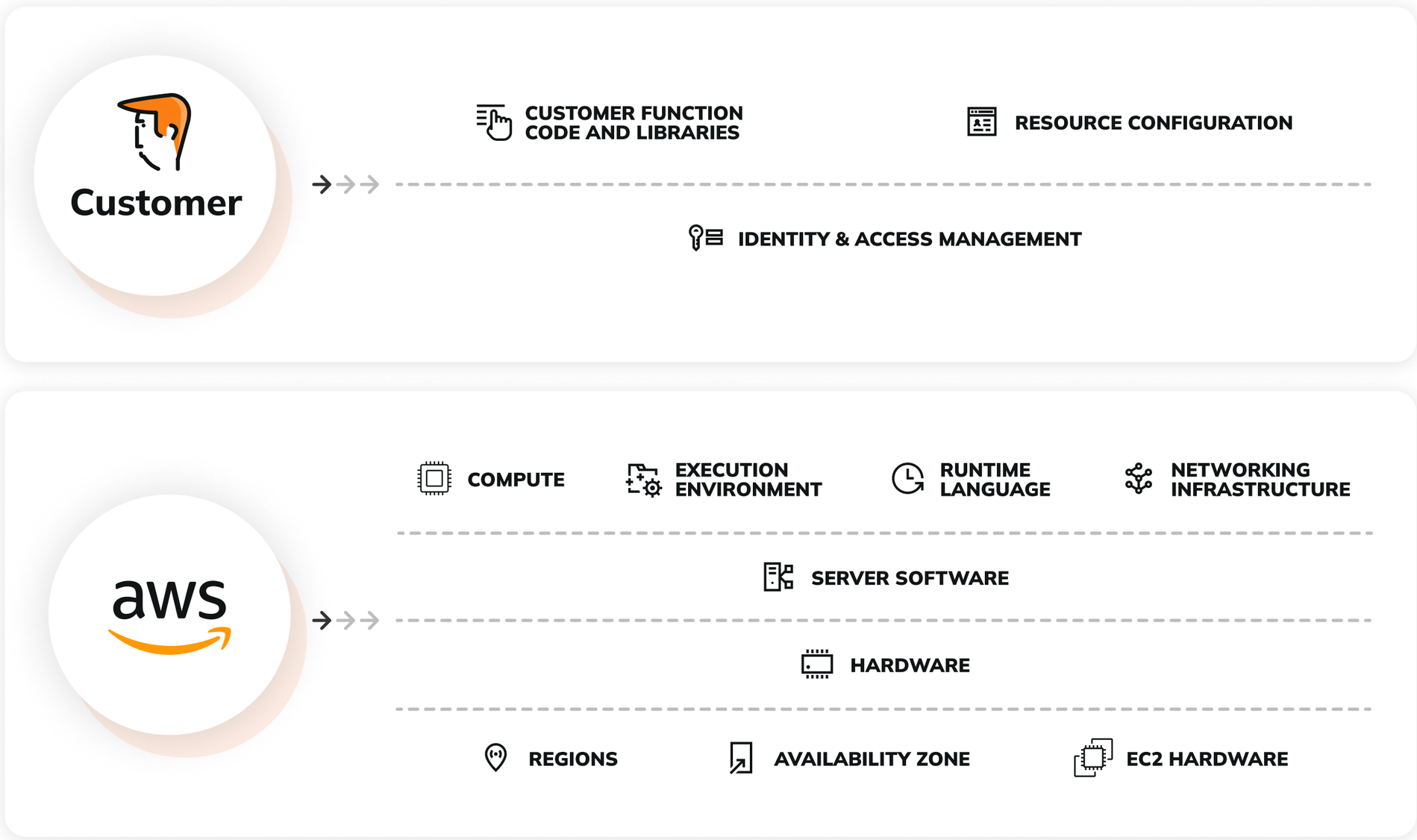
Container orchestrators without control of the underpinning host – like Amazon ECS with Fargate launch type – are also considerably more constrained than their counterparts, although how much depends on the orchestration service and availability of hardware capabilities. Increased availability is to be expected as the services mature and want to attract new use-cases.

08 Shared responsibility model

The main reason to embrace serverless is to delegate as much responsibility to the cloud provider as possible. This imposes restrictions in terms of programming model (see the Opinionation, Stateless vs. Stateful, Event orientation, and Timeouts sections), and access to hardware capabilities (see the Access to hardware capabilities section), but it comes with the reward of having far less work to do besides specifying the logic of the applications.

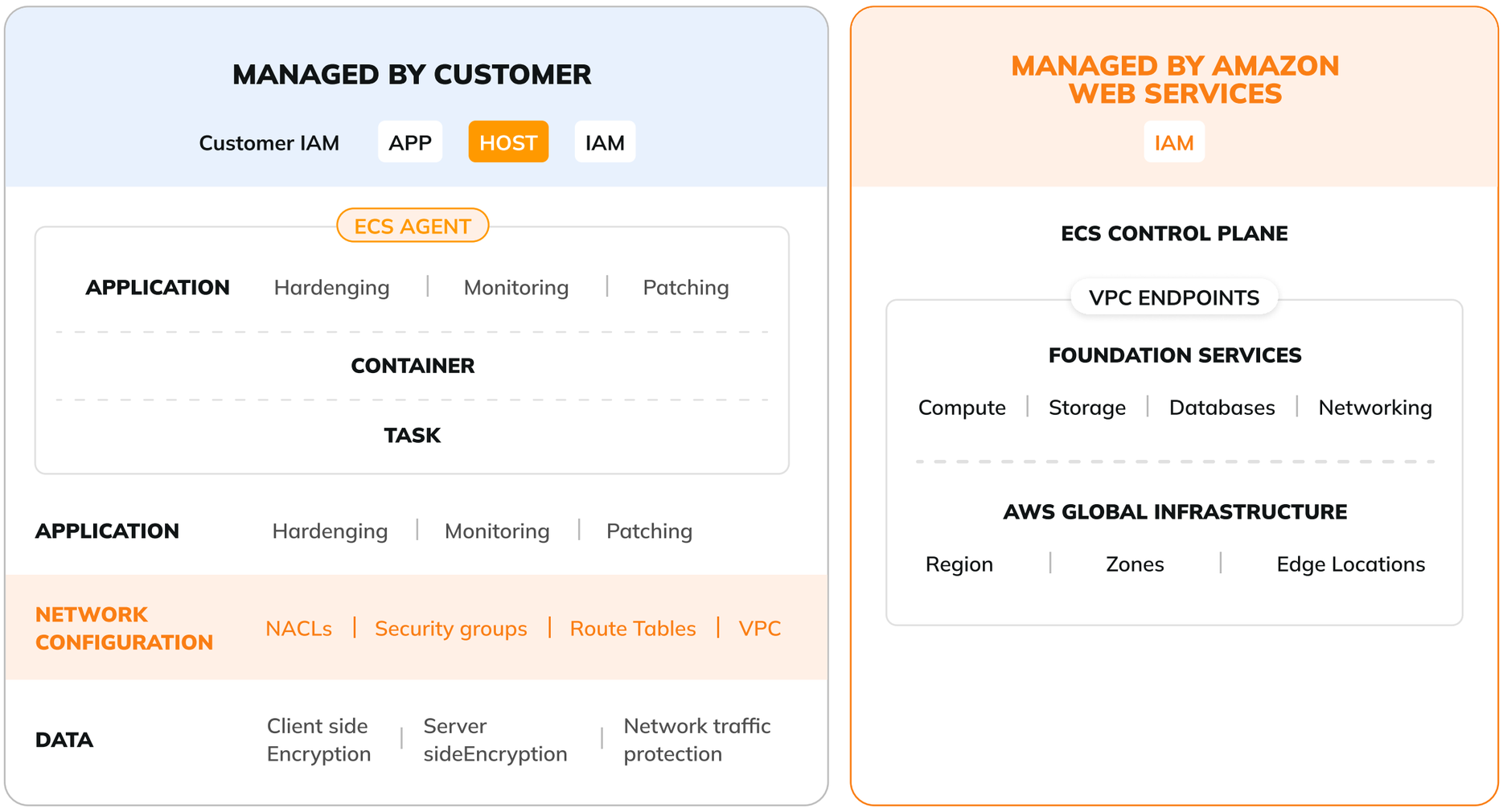
On AWS, the partition of responsibilities between a customer and the cloud provider is called the Shared Responsibility Model. While keeping in mind that “security is job zero” (that is, there is nothing more important than security), and that customers and AWS need to collaborate to keep applications secure, the more of the computing stack AWS controls and has exclusive access to, the less is expected from the customer.

Compare the partitioning of responsibilities between customer and cloud provider for AWS Lambda, Amazon ECS with Fargate Launch-type, and Amazon ECS with EC2 launch-type:

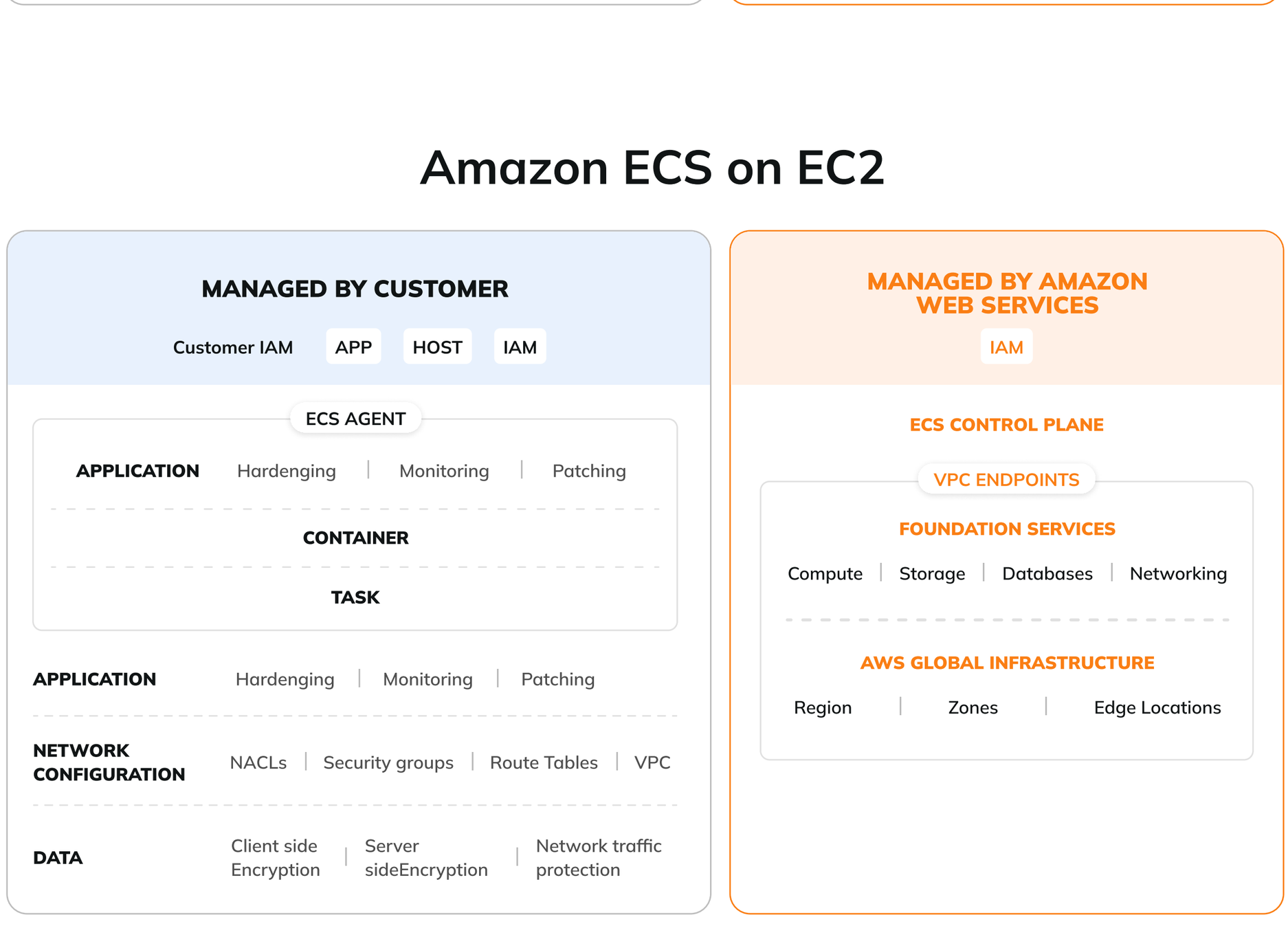


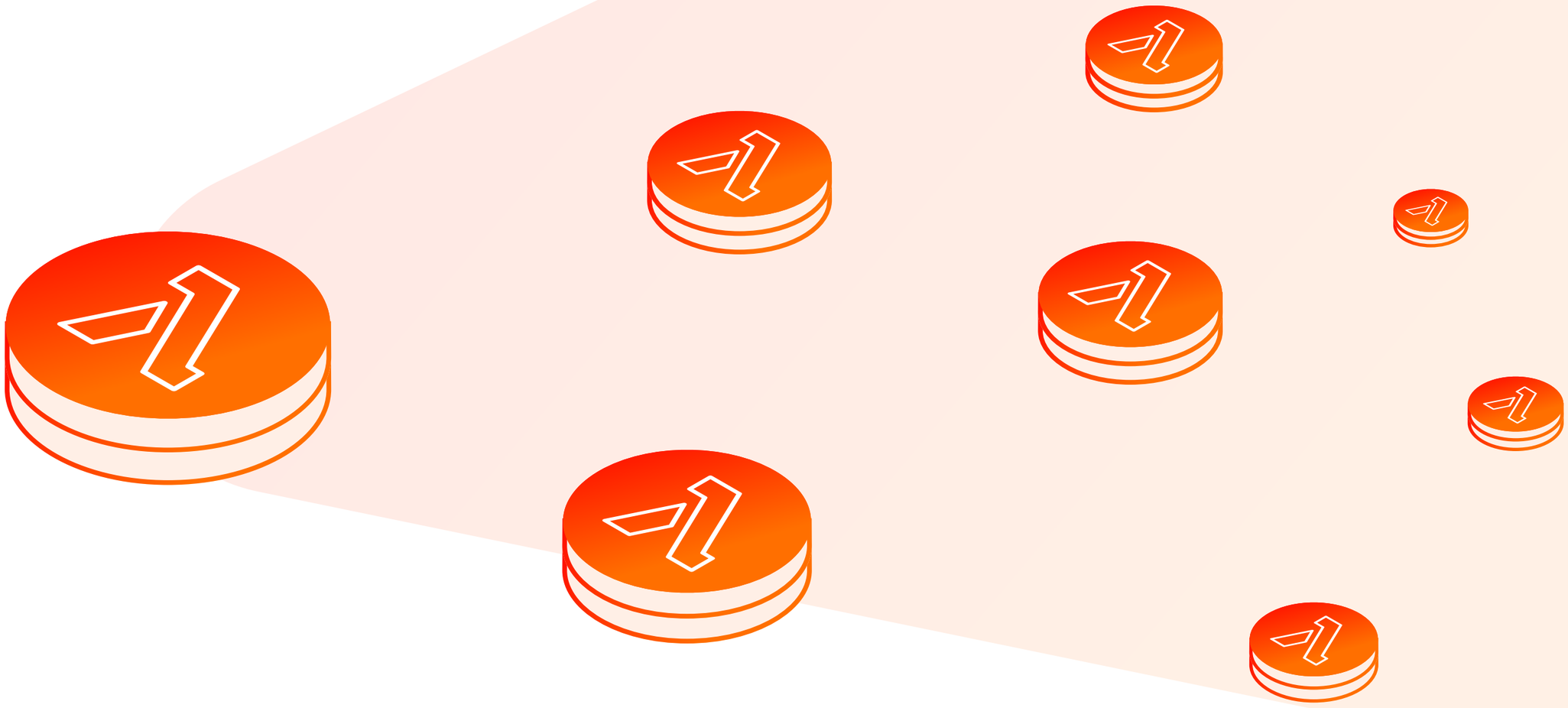
There is a stark progression in terms of when there is more customer responsibility the less serverless the service is. And make no mistake, every bit of additional responsibility you accept along with the choice of compute service, you pay for in developer resources to do more tasks, and take on more risk in terms of governance and security.

Amazon ECS on Fagate



Amazon ECS on EC2





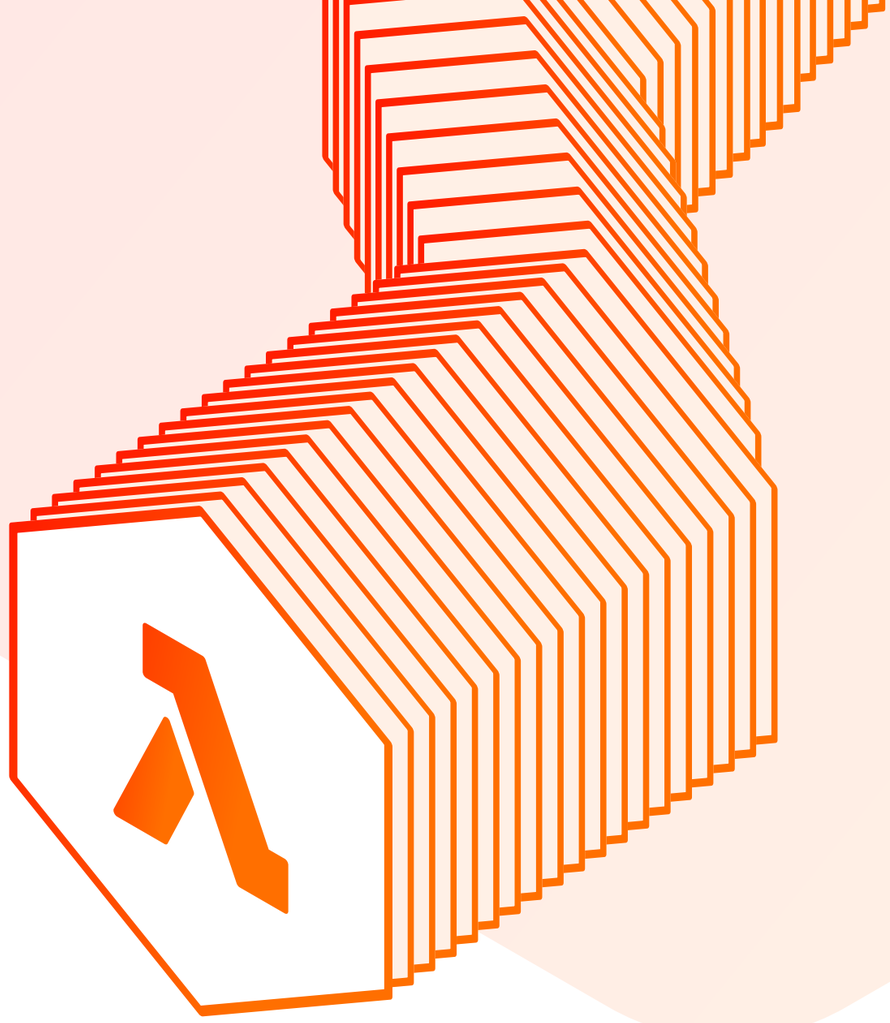
PRICING

From the point of view of pricing, it is not generally possible to draw a meaningful comparison between AWS Lambda and the many container orchestration offerings. The fact that a container usually handles multiple requests in parallel, as opposed to the one request at a time of an AWS Lambda function, effectively voids most generic chances at a direct comparison.

In our experience, one needs to reach very large volumes of served requests to make AWS Lambda significantly more expensive than containers, to the extent that one may bring it up with AWS for a negotiation.

One should keep in mind that the most significant expense for an engineering organization is often the developers, and therefore optimizing their productivity, freeing them from a lot of operational concerns the way that AWS Lambda supports, is usually an excellent approach.

Nevertheless, there may be use-cases where AWS Lambda may grow anti-economic: at Lumigo, we are considering migrating some extremely heavy-duty AWS Lambda functions (running thousands of times an hour) to containers to get better usage out of the computing resources for which we pay AWS. When you are torn between AWS Lambda and a container orchestration due to pricing, the best option would be to talk to specialists in cloud cost management.



Containers vs. Serverless - How to Choose?

So far, we have outlined some fundamental differences between containers and serverless. But how do you go about selecting one approach over the other? The rule of thumb, in our eyes, is simple:

You should use AWS Lambda for your next codebase unless you have a good reason not to.

In this section, we give an overview of some reasons that may steer you away from AWS Lambda in your use cases. It bears to note that one of the usually-touted reasons to avoid AWS Lambda—pricing— in our experience is extremely unlikely to be a deciding factor early on in your development process, if ever. For more information, refer to the Pricing section.

01 AWS Service Quotas

No matter where compute processes run, they consume resources (e.g. memory and CPU). When you run containers on self-managed infrastructure, resource consumption is only limited by the size and strength of your infrastructure. On AWS Lambda, that is not the case. Resources are capped by the AWS service's hard quotas and limits (for more information, refer to the What is Serverless Computing? section).

One of your first qualifying factors for using Lambda should be: does my use case fit within Lambda's limits? If the answer is yes, Lambda can be considered. However, if you have a processing job that consistently takes more than 15 minutes (Lambda's function duration limit), or if you have specialized infrastructure needs (e.g. GPUs), containers might be a better fit for your workload.

02 Need for specialized hardware capabilities

Some workloads, like those related to image processing, natural language processing, and machine learning, greatly benefit from access to specific hardware capabilities like GPUs. Lambda does not offer access to such capabilities; refer to the Access to hardware capabilities section for more information.

03 Traffic predictability and variance

The amount of resources your application consumes is directly related to the number of requests your application needs to serve. When your application serves a consistent and/or predictable number of requests, it is easy to anticipate and provision the required resources beforehand.

But what happens when your traffic varies widely and quickly? What happens when you are a booming e-commerce site on the eve of Black Friday, and you simply do not know how many requests your application will need to serve tomorrow? In this case, provisioning just the right amount of resources is hard. If you provision too little, you run the risk of throttling customer requests. If you provision too much, you could end up with unutilized resources that substantially increase your costs.

This is a common dilemma for customers running containers on their own provisioned infrastructure. With AWS Lambda, traffic forecasts are not required. The service automatically scales up to serve any incoming number of requests, and you only pay for the requests your application serves. For more information about how Lambda and containers scale up and down, refer to the Cold starts vs. Scale-up time section.

04 Required level of control over your tech stack

When running containers, you can decide the infrastructure and operating system you run on, and exactly which libraries are included in your production environment. With Lambda, AWS manages the infrastructure, including the OS and the production environment.

Most customers are better off focusing on their business logic, and offloading ongoing infrastructure management to AWS, which seamlessly handles things like security patching and OS upgrades for you. However, if you have a proprietary OS or infrastructure that directly contributes to your business advantage, or need full control over your production environment, then containers might be a better fit. For more information about the limitations that the Lambda runtime environment applies to your functions, refer to the Opinionation section.

05 Reducing lock-in

Containers are easy to migrate, either from on-prem to the cloud or across cloud providers. Container images, in and of themselves, are largely not vendor-specific (container runtimes, however, have differences in terms of networking and other capabilities that may make migration less than straightforward). The relative ease at which you can port container definitions across environments is one of the key benefits of using containers.

AWS Lambda, on the other hand, is a native AWS service, and as such, includes definitions, syntax, and deployment processes that are specific to AWS. While it is certainly possible to port an AWS Lambda function to a different type of compute (either on AWS or on a different provider), it involves a considerable amount of overhead. This is particularly true if you take advantage (and, in our opinion, you definitely should!) of the deep integration of AWS Lambda with other AWS services. For more information, refer to the event orientation section.

06 Team size and project lifecycle stage

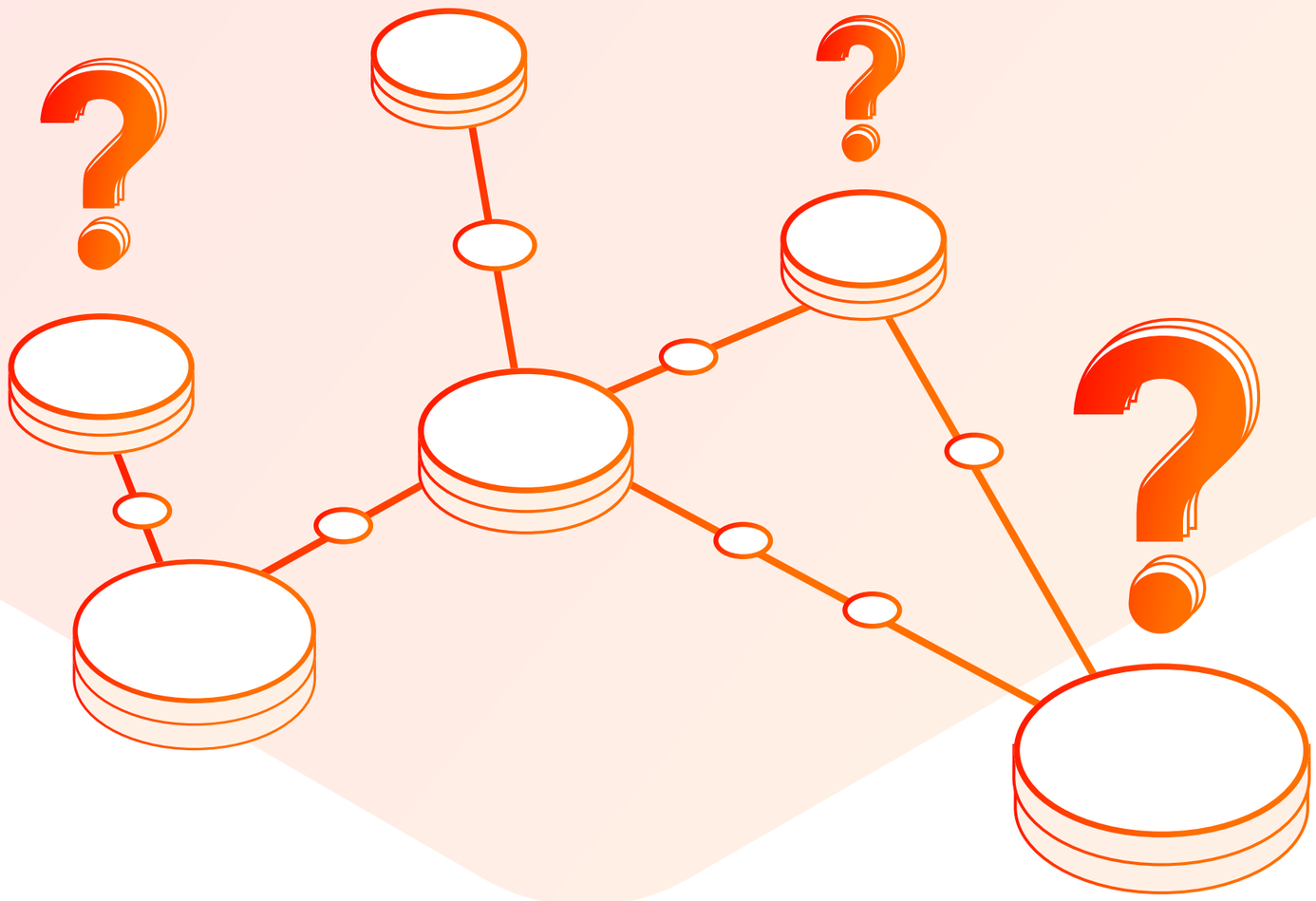
AWS Lambda is easy to get started with. All you need to do is write your code in one of the supported runtimes, and deploy. With AWS Lambda, much of the overhead related to provisioning, managing, and scaling your application is done by AWS. In some cases, small teams who are just getting started and are low on resources prefer to start with AWS Lambda, so they can focus on their code and leave infrastructure management to AWS.

07 Team expertise

Containers are broadly more understood than AWS Lambda, so your team might be more proficient with the former than the latter. Since expertise with a platform is key to the developers' productivity and velocity, your team may feel more productive with one of the various container orchestration options out there..

08 Pre-existing software

AWS Lambda offers a very opinionated programming model, which may not fit the existing software you want to move to the cloud (e.g., with a "lift and shift"). Indeed, AWS Lambda tends to be the compute of choice for new software, although the authors have ported to AWS Lambda with relatively little effort from applications that were offering HTTP APIs. For more information on the programming model of AWS Lambda, refer to the Opinionation section.



Observability and Debugging Challenges

The observability of cloud applications is a challenge, irrespective of whether they are serverless or containerized. Especially when one embraces the microservices architectural pattern, the resulting applications are made of highly distributed services that communicate with one another synchronously and asynchronously. Unfortunately, this leads to multiple points of failure, within the services themselves, but also “in between” them, e.g., in the network infrastructure or the service mesh connecting the various services.

In distributed services, failures spread. A database failure may cause another failure in the application querying it. So, when failure in distributed systems occurs, the root cause is often upstream of where failures are observed. In a distributed environment, being able to “navigate” upstream to find the root cause is often challenging, and can be an utterly daunting task with complex architectures, under the duress of an outage, for operators that are not intimately familiar with the “big picture” of their applications.

01 The basics: **Metrics and logs**

Traditionally, applications are monitored through logging and metrics. There are a variety of monitoring services that provide support for logs and metrics, including native services such as CloudWatch. However, logs and metrics that represent a “single component” view and provide neither a holistic understanding of how failure spread across the architecture, nor enough context in terms of how specific requests are served, where their latency is coming from, and how they fail.

02 End-to-end overview through distributed tracing

The “big picture” is precisely what the observability approach called “distributed tracing” provides in a nutshell: distributed tracing consists of collecting telemetry about which requests are being processed by which components of the application, and how those components interact with one another. For example, when your API-providing application receives a request for details for a specific product, specialized instrumentation, added to your application manually or automatically, extracts span data.

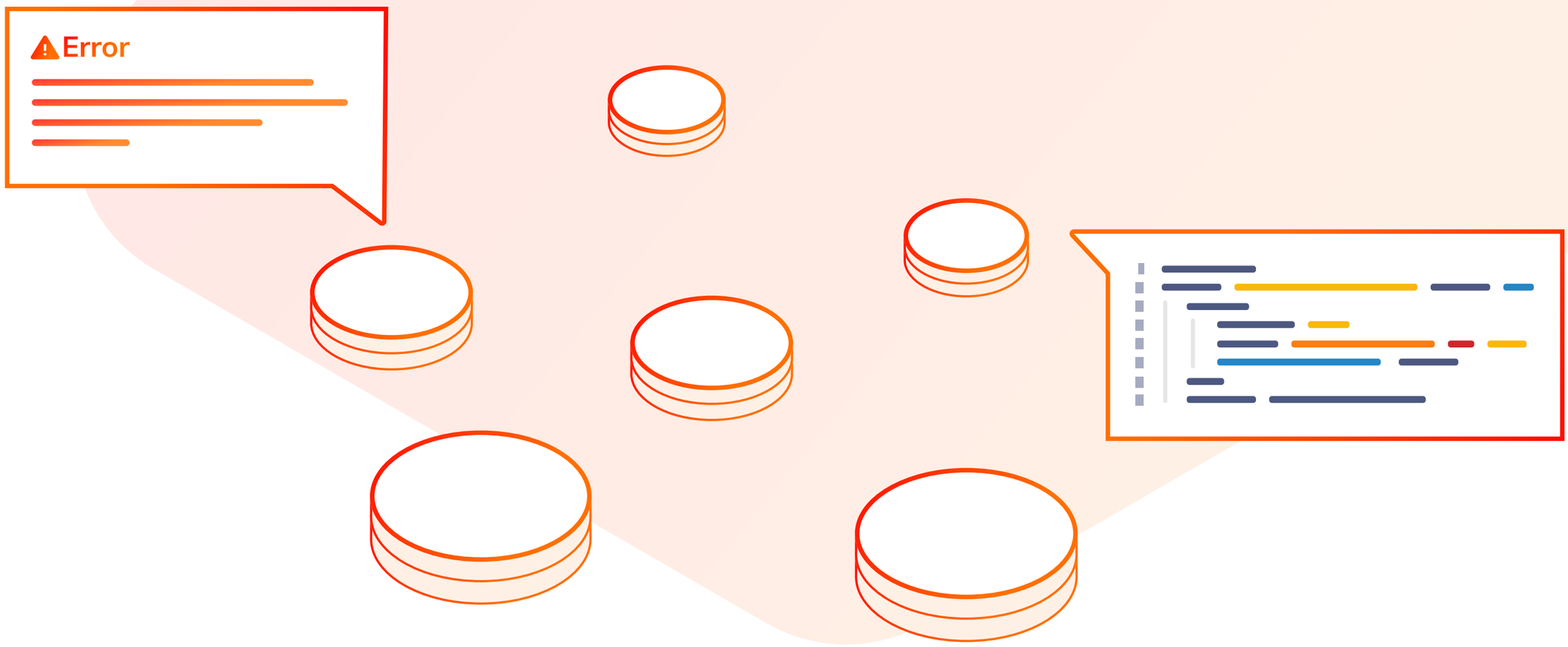
Each span describes, for example, the reception of a request, or a query executed against a database. Spans are grouped in traces, and related to one another in terms of causality through relations called parent-child relations in virtually all current distributed tracing systems: When your application queries a database, the span representing that query is the child of the span that represents the reception of the API request that needs the data from the database.

The parent-child relations between spans are possible due to the concept of trace concept propagation, namely the conveying, within one application (e.g., via GoLang Context API or Java ThreadLocal variables) and over direct and indirect interactions (e.g., via HTTP or Kafka headers), which trace is being recorded and what is the identifier of the last span created within this trace so far (which will be the parent of the next span).

When distributed tracing is performed correctly, it documents:

- Where latency (i.e., “response time”) originates and how it adds up for the end-user;
- How failure originates, propagates, and spreads across applications;
- The impact on the end-users of failures deep down in your architecture.

Recognizing the value of distributed tracing, AWS X-Ray provides AWS-specific distributed tracing capabilities, but it is unfortunately very limited in terms of coverage of AWS services and, above all, the various technologies that make part of your application and that must be specifically instrumented to be able to record spans and propagate trace context correctly.



A Better Tool for Debugging Distributed Applications

Using the native debugging and monitoring services will leave some parts of a distributed application unobserved. In such situations, developers can integrate third-party observability tools into their workflow. Lumigo is the go-to platform for monitoring, observing, and debugging cloud applications on AWS.

Lumigo offers a more efficient and practical approach to debugging distributed applications. It offers:

01 Visual System Maps

Lumigo provides a comprehensive System Map that provides an up-to-date view of the entire application architecture. it helps developers view the way services integrate from a high level. The main challenge with distributed applications, especially those that are rich with managed cloud services, is fully understanding your application infrastructure, and how services talk with each other. Lumigo automatically maps request paths to create a visual system map that provides a comprehensive view of your application architecture.

02 Distributed Tracing

Distributed tracing—correlating the executed services in a request so that a single continuous path can be followed—is critical to understanding application behavior, troubleshooting issues and improving performance.

With Lumigo, developers get out of the box automated distributed tracing that tracks every service in every request, with faster insights, better correlations, and reduced alert noise. Additionally, Lumigo highlights all data transmitted across the transaction so even if developers don't log specific data, they still have access to the data passed in and out of each service for debugging purposes.

03 Tracing Managed Services

Highly complex and distributed, cloud native applications are rarely built from scratch, and instead commonly leverage the ready-made software of managed and third-party services. However, developers are blind to these services with monitoring that can only see into owned code (i.e. a Lambda function or container).

Lumigo can see through these black boxes and trace the data sent and returned to the managed services. For example, Lumigo traces requests sent to DynamoDB along with the responses that it returns. Therefore, this helps to improve the debugging capability of managed services significantly. When a request or transaction fails, you not only see the impacted service but the entire transaction in one visual map so you can easily understand the root cause, limit the impact and prevent future failures.

04 Useful Metrics

Lumigo generates more valuable metrics than the standard CloudWatch metrics. For example, Lumigo compiles a list of all the AWS Lambda functions with Cold Starts and the Cold Start latency. It helps developers identify performance issues quickly and fix these issues quickly.

05 Latency Breakdown Timelines

Lumigo provides a latency breakdown timeline for each transaction, including asynchronous invocations. Hence, understanding the time taken for each synchronous and asynchronous invocation allows developers to fix performance bottlenecks. Using Lumigo to monitor, observe and debug distributed applications helps developers improve observability and debugging capability, ensuring that errors get resolved quicker than usual.

Concluding Thoughts

In this white paper, we have endeavored to provide an overview of the most important aspects driving the choice of "Containers vs. Serverless", and concluded that

1. There is no generally correct answer, it depends on a variety of factors like the nature of your software, the skills of your team, whether you want to write entirely new software or reuse an existing one, and more.

2. The same application may very well consist of a mix of serverless and containers, with the decision taken case-by-case for the various components.

3. Going serverless is likely the best default choice for new software, and in some cases, it is practical to port existing software, not designed for serverless, to it.

We also noted that, irrespective of containers or serverless as a foundation for the implementation, the inherent distribution of most modern cloud applications ups the stakes on observability, bringing distributed tracing to the forefront of the indispensable observability capabilities for modern software.

Having accurate, contextualized information about where latency originates and accumulates, where failures originate, how they spread to end-users, and how badly the latter is impacted, is key to making and keeping your application capable of serving millions of clients better and faster with little to no downtime.