Migrating your Application to Serverless Architecture: A CTO's Guide

2020

SIMFORM

CONTENTS

Introduction

If you ask a developer or CTO, about the biggest challenge of developing an application, they will often refer to the scaling of the app. Scalability is a dicey scenario for any firm as predicting the scale is not easy when building an app on an idea. For example, Stewart Butterfield, the CEO and co-founder of Slack, once said during a podcast interview with Reid Hoffman that "We didn't know how to ask people to switch from email to Slack because we were not competing against anything as such in the market."

Companies often face the issue of not knowing the exact scale of a web app. So, they are always on the lookout for a development architecture that can help them scale up and down quickly. The emphasis on scaling up is so much that many firms forget the importance of scaling down.

To understand this, let's go a little down the memory lane. The concept of web app development in the ‘90s was divided into front-end and backend. Companies needed to have physical hardware to support the backend, which was left idle post the deployment. The reason being, the scale of the product was not in line with the size of the hardware. In the 2000's, we saw the rise of VMs or Virtual Machines, making the scaling a little easier than its predecessor. The first step towards the serverless concept came in 2008, when AWS came up with EC2(Elastic-Cloud), and then subsequently S3(Simple Storage Service) with SQS(Simple Queue Service).

Coming back to the present day, serverless is a fully manageable service with Lambda as glue. A serverless architecture offers automatic scaling, redundancy of code, and lower provisioning. The asynchronous nature of serverless reduces the need to deploy a service due to dependency on other services. So, companies can concentrate on each function on the atomic level. The best thing about choosing a serverless architecture is that it reduces the cost of idle resources.

The first chapter of this ebook explores the serverless architecture and provides a skeletal insight into the game-changing concept. The chapter discusses different aspects of architecture, with characteristics, and history of the serverless technology.

Chapter 1

Serverless Architecture: A Brief Recap

What is Serverless Architecture?

Serverless technology is a universal architectural pattern, from mobile app development outsourcing to modernization of software, everything becomes comfortable with a serverless architecture.

It is a type of software architecture where large applications are made of small, self-contained units. These functional units work together through APIs(Application Programming Interface) independent from specific syntax. Each function has a limited scope and concentrates on a particular task. It allows Development Teams to build systems in a granular way.

The word 'Serverless' is often misleading because it uses servers but not in a conventional way. Let's take a look at what exactly serverless architecture is-

Serverless is a merger of two distinct services:

BaaS(Backend as a Service): The backend part is outsourced to third-party services/applications to handle database management, remote development, and even cloud storage.

SIMFORM

FaaS(Function as a Service): It is a part of cloud computing service that helps developers by automatically managing physical servers, virtual machines, and software management on cloud services.

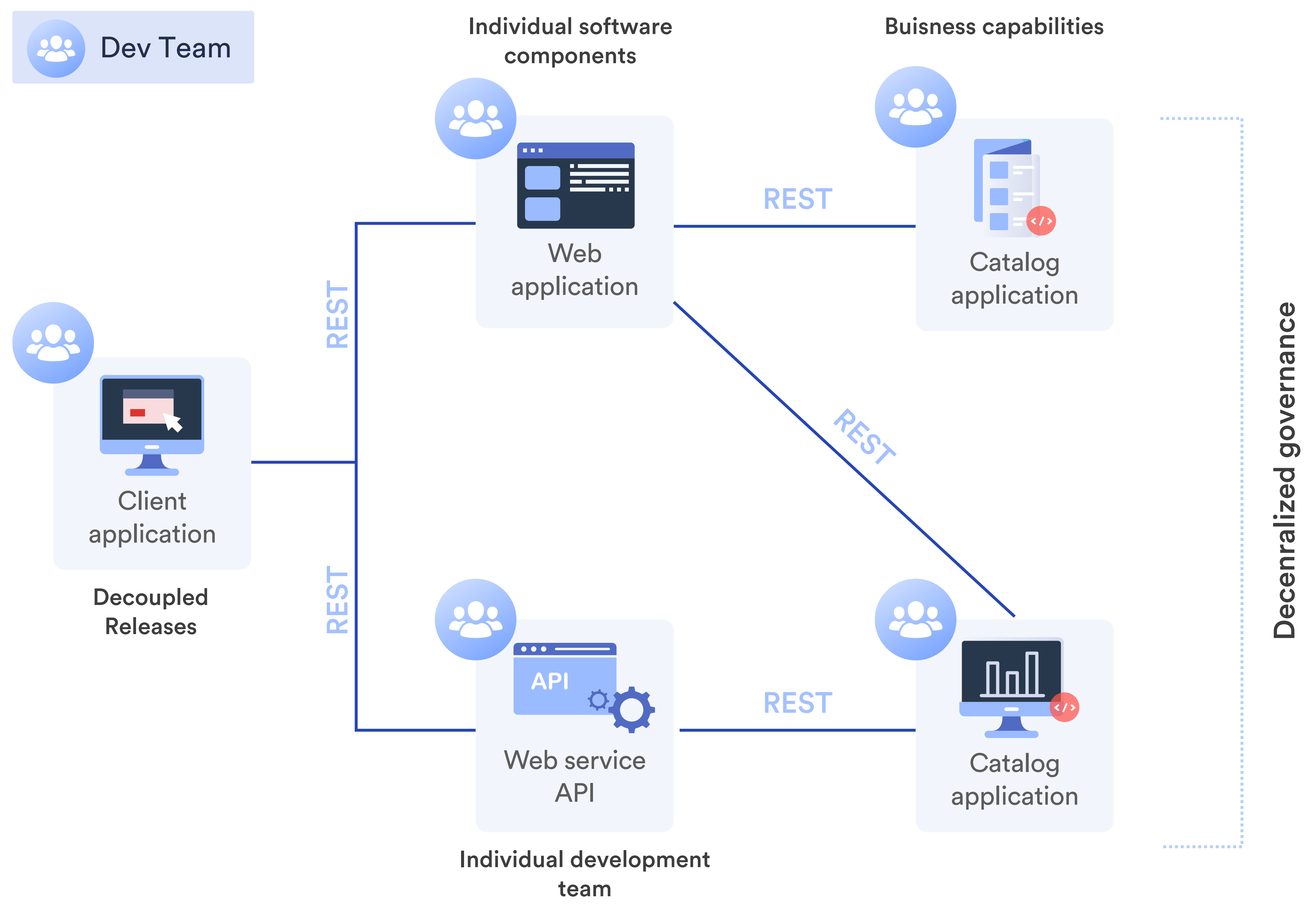
Let’s discuss some key characteristics of Serverless Architecture:

01 **Componentization:** Functions are independent, stateless, and granular units that are easily replaced or upgraded. It is a technique to break down the software into small pieces. Each piece or function is independent, stateless, and micro in size. These microservices can be replaced and updated easily. They can interact with remote procedures through APIs.

02 **Business Driven:** Legacy applications divide development teams into different teams according to technology. The serverless approach builds development teams around business capabilities, with responsibilities for the complete stack of functions.

03 **Product First:** Instead of focusing on the aesthetics of a product and end-user experience only, this approach helps developers work on core services. So, the development teams primarily focus on the functional part of product development.

04 **Decentralized Governance:** Tools are built and shared to handle similar problems faced by other teams. Let's understand the structure with this infographic-



For an in-depth understanding of serverless architecture, refer to A Comprehensive Guide to Serverless Architecture.

History of Serverless Architecture

The first published reference to the term ‘Serverless” appeared on Oct. 15, 2012, in an article by Ken Fromm titled- “Why the Future of Software & Apps is Serverless.” It gained popularity when AWS launched AWS Lambda Functions in 2014 and made its way into the mainstream application development cycles over eight years.

The concept of “Serverless” is an evolved version of microservices which is used by tech giants like-

- Netflix
- Uber
- Codepen
- Coca-Cola
- AutoDesk
- Vogue
- AOL

Problems that Serverless Architecture Solves

Large organizations face many issues when their monolithic legacy systems don’t scale, maintain, or update as they evolve. Serverless architecture provides a solution to that problem. We can call it a software architecture where complex tasks are broken into granular processes (functions) that operate independently and communicate through language-agnostic APIs.

Conventional systems are composed of a user interface on the client-side, an application on the server-side, and a database. The app processes HTTP requests to obtain information from the database and sends it further to the browser. Serverless, however, handles HTTP requests/responses with the help of APIs and messaging.

What’s more, a serverless architecture rebels against the traditionally enforced architecture standards and communicates directly with open formats like HTTP, ATOM, and others.

As conventional applications scale higher, intricate connections grow. With serverless architecture, you can split them up. This helps in horizontal scaling, which makes it easy to manage and maintain separate components.

For example, Netflix uses AWS Lambda to create serverless infrastructure. The idea behind using a serverless approach was to not make the errors of conventional systems. Netflix has a massive user base of more than 60 million customers and services spanning over 60 countries.

Thus, the kind of user base the infrastructure needed to handle was humongous. So, Netflix switched to a serverless approach for better assets management and database rerouting to avoid unhealthy local systems.

Similarly, Bustle also switched to serverless for better database management. According to Tyler Love, ex-CTO at Bustle: “Our previous website was built on monolithic architecture which needed a fair amount of server management, automation, and monitoring involved to keep the website running smoothly.

After migrating to serverless architecture, we have to put zero efforts in scaling high traffic. Plus, nobody is required to work on managing servers; as a result, we all are focused on building new features and innovation.”

To sum it up, a serverless architecture requires -

- 0% Efforts to scale application
- 50% Reduction in Team size
- 84% Cost savings

Let's move ahead to understand how serverless architecture works in coordination with DevOps

Relationship of Serverless Architecture with DevOps

Incorporation of any upcoming technology is just the tip of the tech-berg. Perhaps the most significant bump is fostering a new culture that encourages risk-taking and accountability for the entire project from 'cradle to the crypt.' Development Teams may discover a culture shock when more autonomy is imparted to them than before.

Subsequently, DevOps will play a critical role in determining where and when to use serverless. Trying to combine serverless with bloated, monolithic legacy systems may not always work.

With serverless, services are continuously developed and refined on-the-fly. DevOps engineers need to ensure that changes in components are always deployed. This can be achieved by working closely with the internal stakeholders and suppliers to incorporate updates.

Responding to the Changing Market

The shift to serverless is distinctly evident. The confluence of mobile and in-expensive computing with the lowest time to market pushes businesses to this exciting new approach.

Paul Johnston, Serverless Developer at AWS, in his article titled, “Don’t Dip your Toes in Serverless, you Have to Dive Right In!” stresses on why the IT Industry needs to embrace this new technology saying, “Just go for it, or you’ll find yourself a long way behind the curve when everyone starts to go Serverless.” Companies like Netflix, Paypal, and Airbnb have heeded the alarm and are moving forward with serverless architecture at a rapid pace.

Serverless architecture is more than just an idea. The migration of applications from a monolithic to the serverless architecture needs a business initiative. Every business decision needs the vision to make the most of serverless technology. If you seek the business perspective of choosing serverless over monolithic architecture, you should understand the factors that may affect your decision-making.

Chapter 2

Serverless Architecture as a Business Initiative

There must be a business initiative for serverless architecture to work successfully in an organization.

One of the biggest challenges of adopting serverless technology is having the vision of its capabilities. The reason being, many peers and experts often discuss the risk of data exposure in a serverless architecture. But, the reality is far different from what's dreaded!

There is no doubt that the approach has been successful in scaling businesses better. Even the Cloud Programming Simplified: A Berkeley View on Serverless Computing, which is a report released in 2019, clearly states that " In our definition, for a service to be considered serverless, it must scale automatically with no need for explicit provisioning, and be billed based on usage." But scalability is just one of the many benefits it can offer for firms.

With technologies like 5G making the data transfer more rapid, innovations and speeds take the centerstage. Such designs can make or break your business!

The ideation of businesses to choose a serverless architecture is driven through the constant need for innovating features. The decoupling from the legacy system empowers DevOps with the freedom to experiment with new approaches and continuously iterate changes and business requirements.

Traditional systems cannot scale up rapidly, and chances are, that may cripple companies in the ever-expanding spectrum of technology. Boris Scholl, VP of Development at Oracle, shares his talk about a situation where they had the monolithic system. It had eventually become so complicated that when they added new code to it, the system stopped working, and it took them two whole days to figure out why! Quite fast, eh? No!

Two days can make a big deal when you are on a deadline, and your end-product is time-intensive. This is the reason why many choose serverless over legacy systems approach.

But, many organizations are yet to figure out where exactly does the serverless fit into their traditional systems. Apart from coding, Devs now need to think about the broader perspective and work upon how all these technologies can work together.

With serverless, the feedback loop is instantaneous. The code should begin monitoring the minute it is deployed. The DevOps may be dealing with the monitoring hundreds of functions. Moreover, the data is available in real-time, which means you need to tweak and adjust it on-the-fly.

While it may not seem so at the moment, it still is a real deal!

01 Business Case of Serverless

Incorporating serverless isn't an easy task. It is recommended to have a business initiative associated with it. Most organizations work according to a hierarchy that starts with the decision-making power move through the CEO & Board of Directors, then comes to the CTO. The remaining staff is mandated to follow accordingly.

But, off late, there has been a trend of imparting decision-making powers down the hierarchical structure. The reason behind this is simple; firms need faster and reliable solutions to handle emerging markets.

Allan Naim, Product Manager of Container Engine and Kubernetes at Google, predicts the future, saying it is not too distant that every organization will become an IT company irrespective of the segment of the industry. It is because the customer data is becoming as profitable as their products or services.

To leverage this data, organizations need to move quickly and update their offerings following the continuously evolving landscape. Legacy apps cannot accommodate these expectations, especially for IoT and mobility.

Competition has become tight over the years with the inception of startups everywhere. That being the case, including serverless architecture with legacy systems is the need of the hour for business organizations.

02 High Adaptability + High Specialization

Serverless architecture is highly specialized and completely adaptable. These qualities make it suitable to serve excellent quality IT products to a large number of customers in a short span. It is not only changing the way organizations write code but their structure too.

For example, developers' roles are highly specialized within legacy environments, whereas they are quite diluted while working with serverless environments. So, every developer is free to operate on multiple parts of the application rather than the assigned development process. This way the applications are continuously monitored and updated as and when developed.

Adrian Cockcroft, ex-Cloud Architect at Netflix mentioned, for three days in August 2008, Netflix couldn't ship DVDs due to the corruption in their database. That was unacceptable. Netflix has to do something. At the same time, building data-centers isn't a competitive advantage but delivering video is! And that's when we moved to AWS and later to microservice architecture.

It isn't that Netflix has never faced a downtime, but it is way more reliable than what we had before. The only reason is that we have taken steps to make their service reliable.

03 Agility & Speed

Organizations like Netflix, PayPal, and Condé Nast are known for their ability to scale high-volume websites. However, their monolithic architecture restricted them from adding new or changing old functionalities rapidly. Therefore, six years ago these companies brought changeover into their systems. The main reason for this change was to get rid of their dependencies. With efficient testing and quick deployment of altered code, meeting their customer demands was easier.

With serverless architecture, agility and speed are paramount. Incredibly fast-moving organizations are continually striving for simplicity and their ability to incorporate changes quickly without much of a hassle.

Anders Bresell, Head of Technology Development at Telenor Connexion, states : Using AWS, our ARTS platform took just four and a half months to launch and achieved a return on investment in six months. It would have taken at least double the time if we'd run it as an in-house project.

Key takeaways from the case study-

- 50% reduction in Time-to-Market for Data
- Analytics Platform
- Launch of IoT products in 3 months rather

04 Major Change in Software Development

Serverless architecture is a natural fit for modern software development cycles. For example, web languages such as Node.js that work well with small components can help you integrate changes rapidly.

Since serverless functions are self-contained, it is easy to change, replace, or remove the codebase. If you want to attach new functionalities, you can do so without any delay.

Moreover, continually evolving apps require a new way of thinking. There is a sea of change in the way how software is developed today. Significantly, with speed, you can change your code with ease.

Nick Rockwell, CTO of The New York Times Co. in an interview with CIO Journal states: With the growing practice of moving digital resources to the cloud and its subsequent technologies, we will face significant changes in the way we develop our software, especially with Serverless.

It has made few inroads into mainstream IT though most organizations aren't ready for such a massive shift, and corporate system of records don't match the technology. However, we already see the leading indicators of serverless adoption, and this will accelerate.

05 Data-driven Approach

With serverless architecture, the data is inherent to each function and can only be accessed through API. The data in these functions are private, allowing them to be loosely coupled to operate and evolve independently.

However, this creates two main challenges: first, to maintain consistency across multiple functions and implement queries that can fetch data from various parts.

Though there are multiple ways to solve these issues, as far as serverless architecture provides speed, aka 'the coin of the realm,' it is wise to implement it pronto.

06 Breaking the Boundaries

Serverless architecture is breaking down the boundaries between software development and its operations. Companies, therefore, need to conduct their preliminary research on whether they are prepared to implement this new approach or not.

This does not mean completely abandoning the legacy systems and moving towards serverless architecture. The legacy systems display excellent operations in some of the cases. So, changing it without prior thought and business case would be idiocy.

There has been a massive rise in computing resources' innovations with fog computing and edge techniques. Similarly, mobile technologies are also leveraging these computing resources as a service to provide a seamless experience.

The most significant advantage in the current era is low-cost bandwidth and the emergence of 5G, pushing the rapid development of apps. If you are trying to switch your applications to serverless, here is a practical approach to consider.

Chapter 3

Migrating to Serverless Architecture

We are already into a hyperconnected, digital-centric, mobile world. Organizations that will fail to upgrade their technology approach will fall behind competitors. However, by successfully incorporating easily scalable and flexible architecture these organizations can quickly adapt to the demands of modern world applications.

Breaking the Monolithic Architecture

Replacing Monolithic systems does not happen overnight! This is one of the critical challenges that you will face while deciding to migrate to a serverless architecture.

To make this decision simple, development teams must decide where, when, and how they want to incorporate serverless architecture into their current system. But, first , you need to embrace a whole new way of developing apps.

In an article for Forrester Research, Ted Schadler, Michael Facemire, and John McCarthy call for a need to move towards a '**Four-Tier Engagement Architecture.**' It is undoubtedly evident that the aging "Web" isn't designed to handle mobile apps or sites, and it cannot control the real-time demands of connected products.

The traditional monolithic architecture can't flex, scale, or respond to the emerging needs of an excellent mobile experience and connected products.

With the dramatic changes in the computing and penetration of mobile devices in the market, developers must embrace an entirely new application development approach.

The four-tier architecture that we call an 'engagement platform' can help developers. The platform separates the technicalities into four parts: client, delivery, aggregation, and services. It is a new approach to architecture. It supports a distributed, natively engineered engaging experience, compelling performance, and modular integration on any device, network, or scale.

01 **Client tier:-** It is a presentation layer that insulates each app or device's unique capabilities like desktop, mobile, or dedicated app from the backend services.

02 **Delivery tier:** This layer uses intelligence from the client layer to determine the best way to deliver appropriate content in a personalized manner

03 **Aggregation tier:** It can provide discoverability between app services. At the same time, it also facilitates bidirectional translation between user requests and backend or third-party services.

04 **Service tier:** This element dynamically composes data and business processes through a set of continuous deployable services.

Client Tier

- Mobile clients
- Wearables
- Internet of things
- Responsible for experiment delivery

Delivery Tier

- Optimizes content for proper display on device
- Caches content for performant delivery
- Drives personalization by using analytics to monitoruser behavior

Services Tier

- Existing on-premises systems of record, services, and data
- External third-party services(e.g., Box, Twillo, Urban, Airship)

Aggregation Tier

- Aggregates and federates services tier data
- Provides discovery for the underlying service library
- Performs data protocol translation(e.g., Box, Twillo)

The services tier is realized as many microservices that provide the business process, integration to external systems, and database - Polyglot persistence. All the other stories are responsible for client experience, optimized data caching, app delivery, and API Gateway, which has additional security responsibility with protocol transformation.

We can instead call this pattern as an n-tier architecture. The reason being, microservices or functions are not bound to three or four-tier architecture, because,

- Massive decoupling of application components to amplify their usage.
- The architecture layer increases with the number of third-party services.

But, if this is the future, how do you get there?

The three-tier architecture has been long patronized by IBM, Oracle, Microsoft, and many other giants. However, they have already been taken aback by the upcoming four-tier architecture used by Netflix, Kinvey, Uber, Salesforce, and many more.

The time has come for the CIOs and Development teams to pave the way for n-tier architecture through a serverless or microservice approach. Let's find out how-

Selecting the Service Provider & Tools

It sounds fascinating to move forward to this new approach, but what tools will you need to achieve this? Here's how to decide:

1. First, consider and decide your serverless architecture for the given application, whether to re-architect the whole application or to start with some parts initially, and then move ahead gradually. The next step is to choose a serverless service provider you would like to opt for.
2. Second, figure out how different services will communicate with each other before optimizing their performance, mainly FaaS & BaaS.
3. Third, since serverless architecture provides much agility and speed, you need to optimize those speed gains continually. With this, you should decide, if at all, you will need third-party services or other platforms.

Due to the granularity of serverless architecture, you have the flexibility to choose and incorporate any tool you like at any given time. Let's have a look at the Serverless Technology Landscape.

A Practical Approach: From Monolithic to Serverless

Here is a fact - every application is not a green-field. Although techies of Silicon Valley do like to believe it to be, Let's be realistic! For example, one of our clients had a lot of legacy code based on monolithic architecture. So, throwing all that code away and starting afresh was never an economically feasible solution.

In such cases , it is always better to find some parts of your legacy system and reuse them in the right way. That's why refactoring your existing legacy application into serverless architecture is the best, most feasible approach. You can achieve this while keeping your current systems running and gradually moving them to a more sustainable solution.

It boils down to keeping our external APIs the same while changing how our code operates or how it is packaged, i.e., refactoring.

Migrating to Serverless would thus mean incorporating functions into your application architecture without changing anything to what it does. You won't add any new functionalities to your app, but you will change how your code is packaged and perhaps how the API is expressed.

Organizations can gradually build a new application consisting of microservices and run it parallelly with your monolithic application. Over the period, your monolithic application's functionalities will reduce until either they disappear or become a microservice.

So, let's look at the process-

Step 1: Restructuring the Application

The best place to get started is by revisiting your application's packaging structure and embracing some new packaging practices before changing the code.

The problem with the approach of the 2000s was that it tied each piece of code together with the same deployment schedules and the physical servers. Changing anything to these legacy systems means retesting everything, which makes adding changes too expensive to even consider.

But now, with functions and containers, the economics has changed.

With such developments, you can start reconsidering the packaging. Here are three steps that you need to get started with:

01 Prioritize Module Conversion:

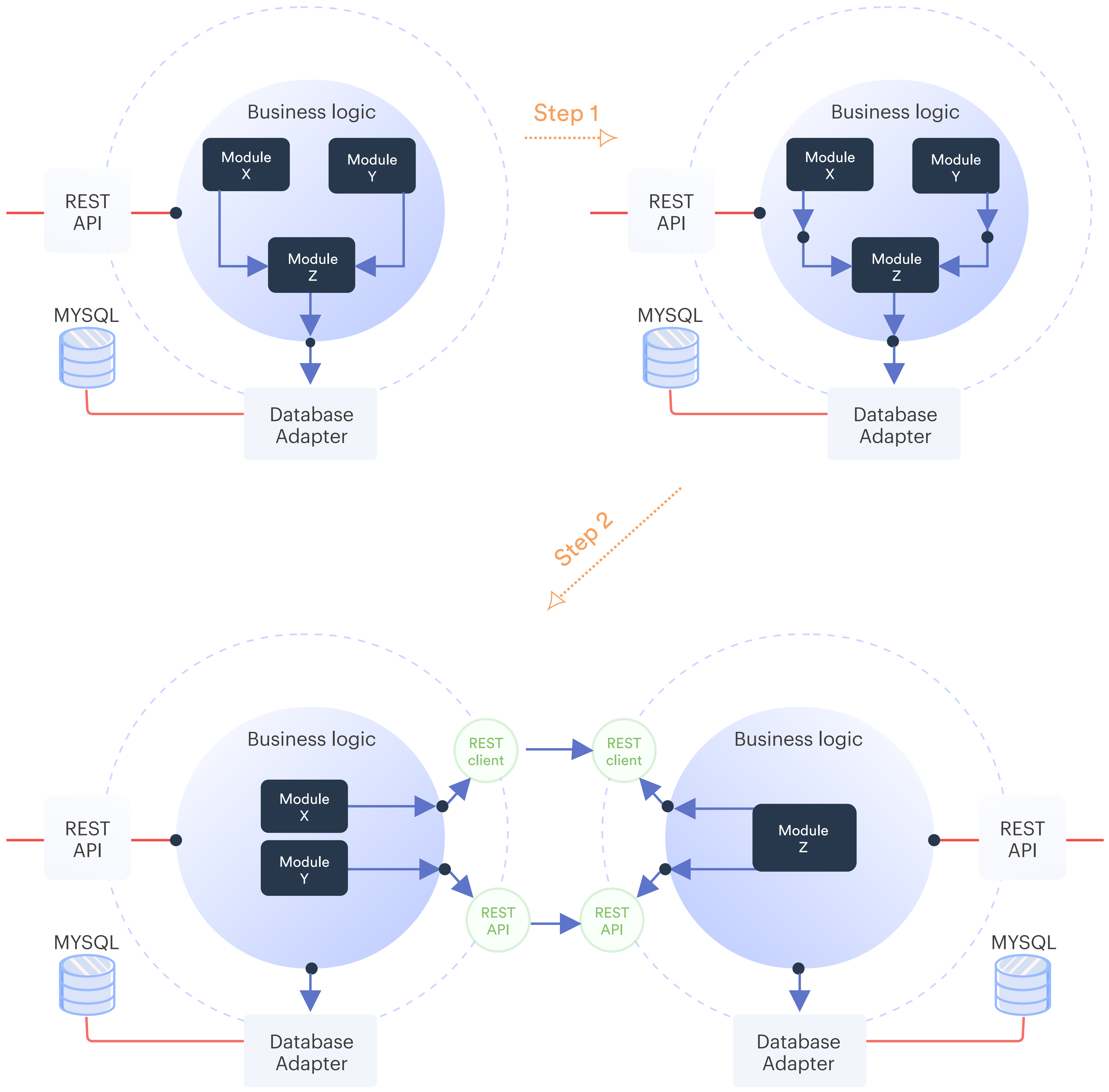
A typical monolithic application consists of innumerable modules, and figuring out which one to convert first to microservices can be a tiring task. A good practice would be to convert those modules which are easy to extract. Once you convert a module into a microservice, you can develop and deploy it independently.

While continuing it, you should look for coarse-grained boundaries within your monolithic application. For example, it'd be straightforward to convert a module that communicates via asynchronous messages with the rest of the application.

02 Extracting a Module

Let's first understand how to implement a coarse-grained interface to your monolithic application before converting them into free-standing microservices. It communicates through an API to identify your application's different parts, which can function with bidirectional APIs. A monolith will need data owned by the microservice and vice versa.

Once you've done this, you can rewrite your code to communicate through an API so that your microservice and monolith app can communicate. Consider the diagram shown below:


Business logic
Module X
Module Y
Module Z
REST API
MYSQL
Database Adapter
Step 1
Business logic
Module X
Module Y
Module Z
REST API
MYSQL
Database Adapter
Step 2
Business logic
Module X
Module Y
REST API
MYSQL
Database Adapter
REST client
REST API
REST client
REST API
Business logic
Module Z
REST API
MYSQL
Database Adapter

In this example, Module Z is the part that we want to extract. As you can see, it is connected to Module X through an inbound interface and Module Y through an outbound interface.

How do you make it a standalone service? Let the module Z remain as it is. Then, separate it from the monolithic application and change the code of Module X & Module Y to communicate with Module Z through APIs.

Once you execute this successfully, you have a service that can be independently developed and deployed. Each time you convert a module, you are moving closer to incorporating microservice architecture.

03 Build, deploy & manage independently

Once all of your modules are converted into microservices, you can manage them independently through CI/CD pipeline, which we will discuss later. It is a step towards leveraging the advantages of continuous delivery.

Step 2: Refactoring the Data

Once you build and repackage the services as per our discussion in the previous step, it is time to focus on the most challenging part of migrating to serverless- refactoring your application's data structures.

Let's get started with some of the basic rules that you should follow:

01 Isolated Data

First, have a look at the database tables that your code uses. If the tables are independent or in a group of few tables joined by relationships, you can split those tables out from the rest of the data design. Next , you can consider your options for database service. This will depend on the kind of queries you want to perform on your data.

02 Batch Data Updates

Even after having only a few relationships, you may consider moving your data into a NoSQL database. But first, you need to consider if you need to perform a batch update in your existing database..

03 Denormalizing Tables

If you have more than a few relationships with the other tables, you can denormalize your tables. However, don't! The initial purpose of denormalization was to reduce duplication and thus reduce disc space, which was expensive. But that's not the case anymore as query time is the thing you should optimize, and denormalization is a full-proof way to achieve that.

Step 3: Moving towards Functions

Once you've successfully converted the monolithic modules into microservice architecture, it is time to transform your microservices into ephemeral functions.

Why move to functions?

Remy Chantenay, who successfully migrated from Containers to Serverless, says that Lambda functions are attractive for multiple positions. Cost efficiency might come off as the primary reason since you only pay for what you use!

When you couple Lambda with API Gateway, it becomes an infinitely scalable solution. It doesn't require more efforts, instances, load balancing, or provisioning-- in short, a paradise for your DevOps Team.

You should be focused on the product, not on the architecture. Serverless lets you do exactly that!

Course of Action

What you can do is migrate one service at a time, and to get started, here are some of them-

- User login and account information
- Authorization & session management
- Preferences or configuration settings
- Notification & communication services
- Photos, media, and metadata

Case Study: How Condé Nast Found their Way to Serverless Architecture

Before the CloudConf 2018, we had a chance to catch up with Marco Vigano, Digital CTO at Condé Nast. Under Marco's supervision, Condé Nast migrated successfully from on-premise setup to cloud services. Later, with the integration of Serverless Functions, they could reduce annual costs up to 4 times.

Hiren Dhaduk
VP of Technology

Marco Vigano
Digital CTO

Many would like to follow the same. Here's the take away from the conversation!

#1. Tell us about yourself and Condé Nast.

I am Marco Vigano. I work as a Digital CTO at Condé Nast Italia, a media company owning some critical fashion & lifestyle magazines like Vogue, Vanity Fair, GQ, Wired, etc.

#2. Can you brief us on the on-premise architecture and their challenges?

If we look by the numbers, it goes like this: we get over 30 million unique visitors each month with 250 million page views.

If we talk about the traffic, 20% of them are from desktop and 80% from mobile. If we talk about the traffic sources, 29% is from social channels like Facebook and Twitter, while 46% is from search engines.

With our on-premise infrastructure, there were a lot of issues, especially with scaling. We were not being able to meet the demands of rising traffic.

Apart from that, development teams struggled with aggressive time-to-market with no optimal delivery and uptime. It was mainly due to the lack of any automation in our CI/CD pipeline. Plus, it cost us way higher than what we had expected.

#3. How did Condé Nast's journey towards migration start?

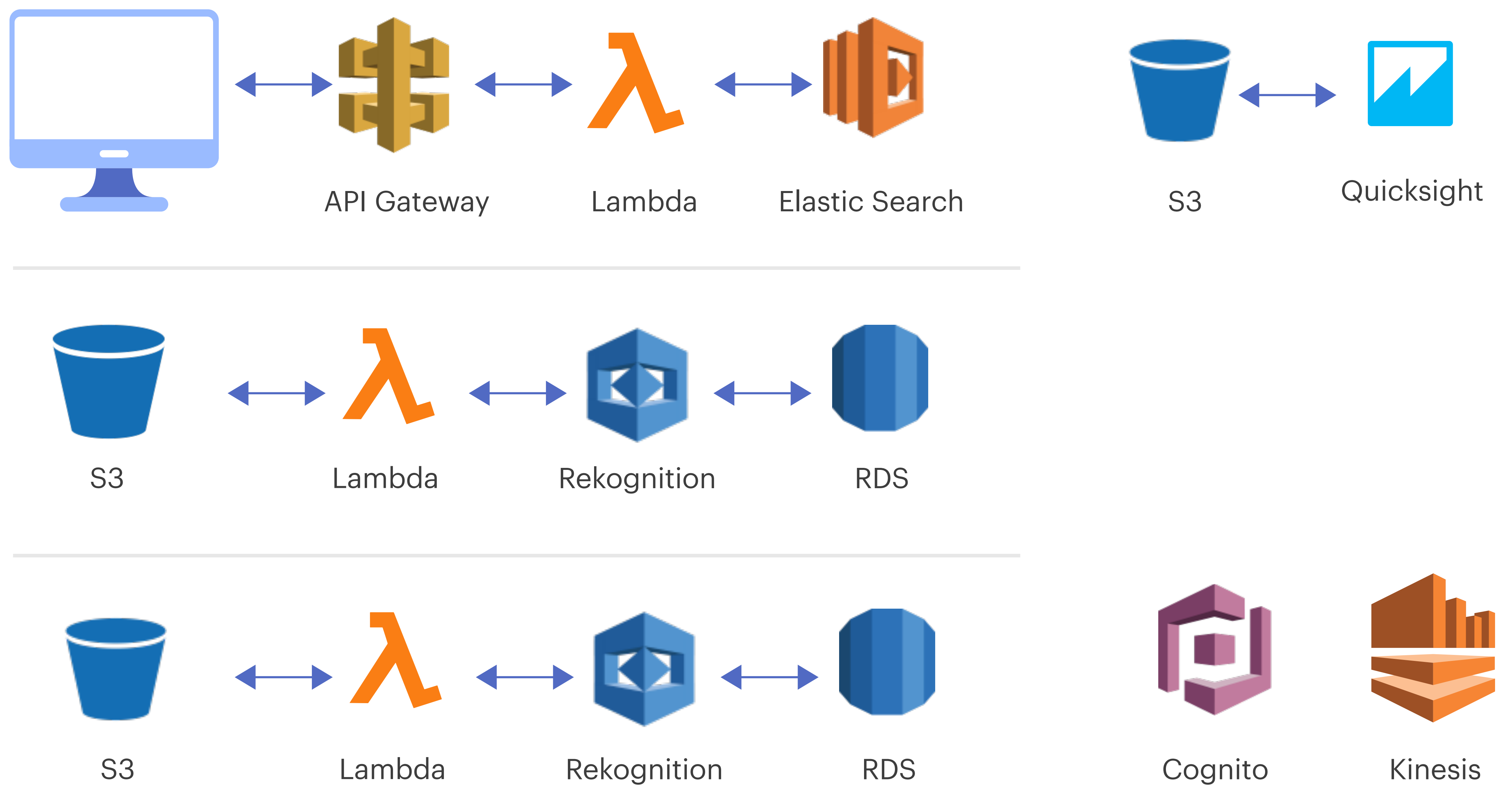
Initially, we thought of moving just one service to the cloud. We ran a pilot project where we first evaluated our system, prepared our service (Wired Magazine) for migration, migrated one service at a time, and then kept testing & iterating continuously.

It started costing us a lot of money, including the cost of an on-premise system, cloud, people, and external service providers. We ended up with 150 servers, 30 databases, and more than 50 load balancers.

And that's when we started destroying our monolithic systems, migrated all of our services to the cloud, and adopted the serverless architecture.

#4. Tell us about the present architecture- Genius, which uses serverless.

Genius is the brain behind our websites, mostly Vanity Fair. Our primary goal is to enhance user experience and offer them personalized content recommended by their business perspective. Or in other words, Genius is a recommendation engine.



As it is evident from the architecture, we use AWS Lambda functions at two places. Firstly, for user authentication through AWS API Gateway and secondly, for face recognition.

Whenever a new image is uploaded to the S3 bucket, it triggers a Lambda function, which puts the image to the Rekognition Engine.

This part is essential for us since we need to identify and tag every image, whether they are objects and celebrities. This same logic has been applied for PhotoVogue.

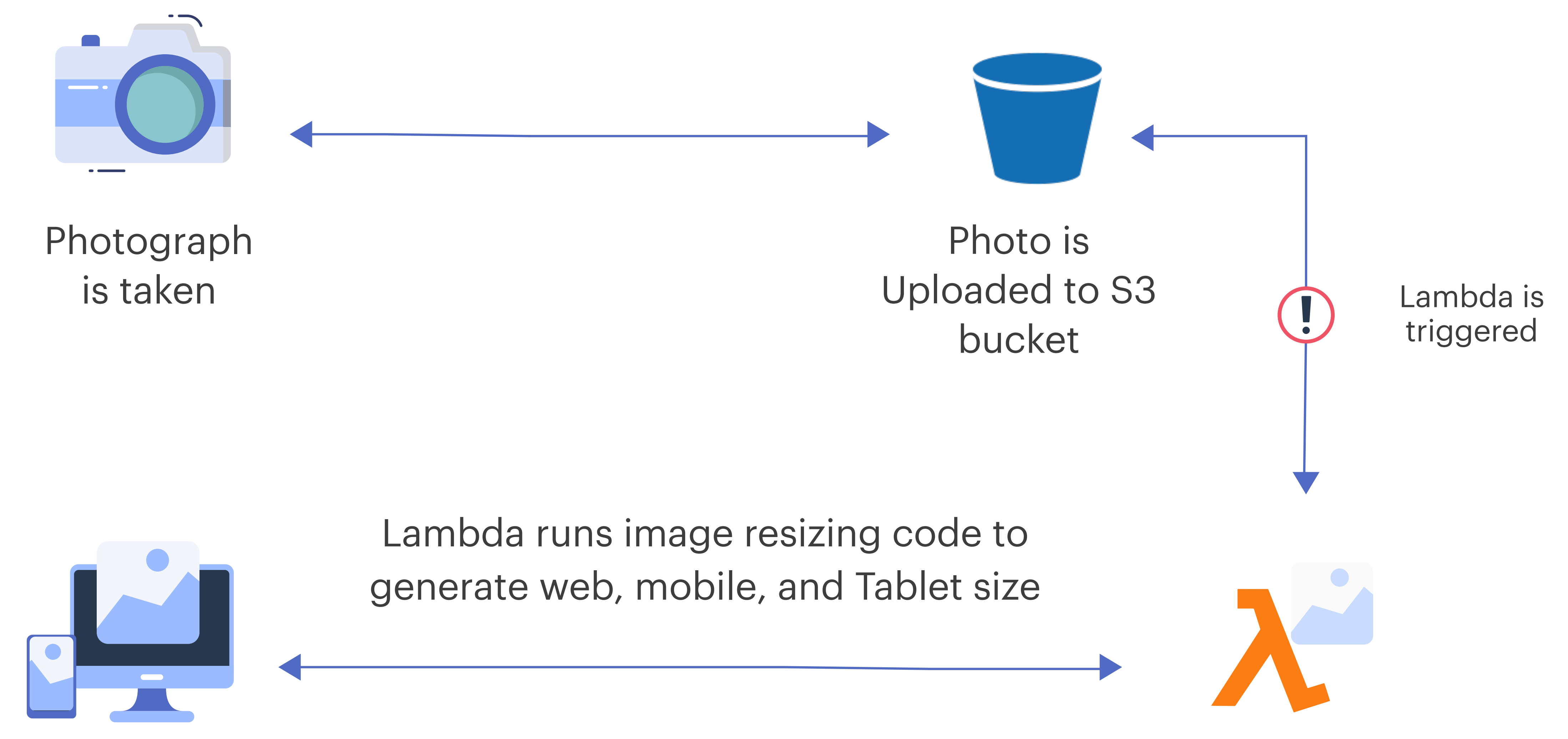
#5. Can you share more information about PhotoVogue?

PhotoVogue is an online platform where photographers can showcase their work. With the growing popularity, we have 130,000 photographers worldwide, having 800,000 images, each up to 50 MB.

Well, the real challenge was scalability. The legacy infrastructure wasn’t able to meet the demands of a growing user base. Plus, we wanted to give a seamless experience to our editorial staff and photographers.

#6. How did you overcome these challenges, and what was the role of serverless in it?

Well, serverless was the best possible solution. It not only provided us with flexible scalability and ease of maintenance but cost-effectiveness as well.


Photograph is taken
Photo is Uploaded to S3 bucket
Lambda is triggered
Lambda runs image resizing code to generate web, mobile, and Tablet size

Example: Image thumbnail creation

Whenever an image is uploaded, a Lambda function is triggered, which converts it into multiple formats, which allows our editorial staff to process them quickly. It increased our user experience by up to 90 percent.

We are now able to provide our resources in hours that used to take hours. We used to spend more time maintaining infrastructure, which has reduced drastically to explore new services.

With serverless, there are no scalability problems. Mainly, our IT costs have reduced by 30 percent. For example, just after migrating to serverless, PhotoVogue held an event in Milan where we had 20 percent more uploads than the regular days; however, we dealt with it seamlessly.

All thanks to serverless!

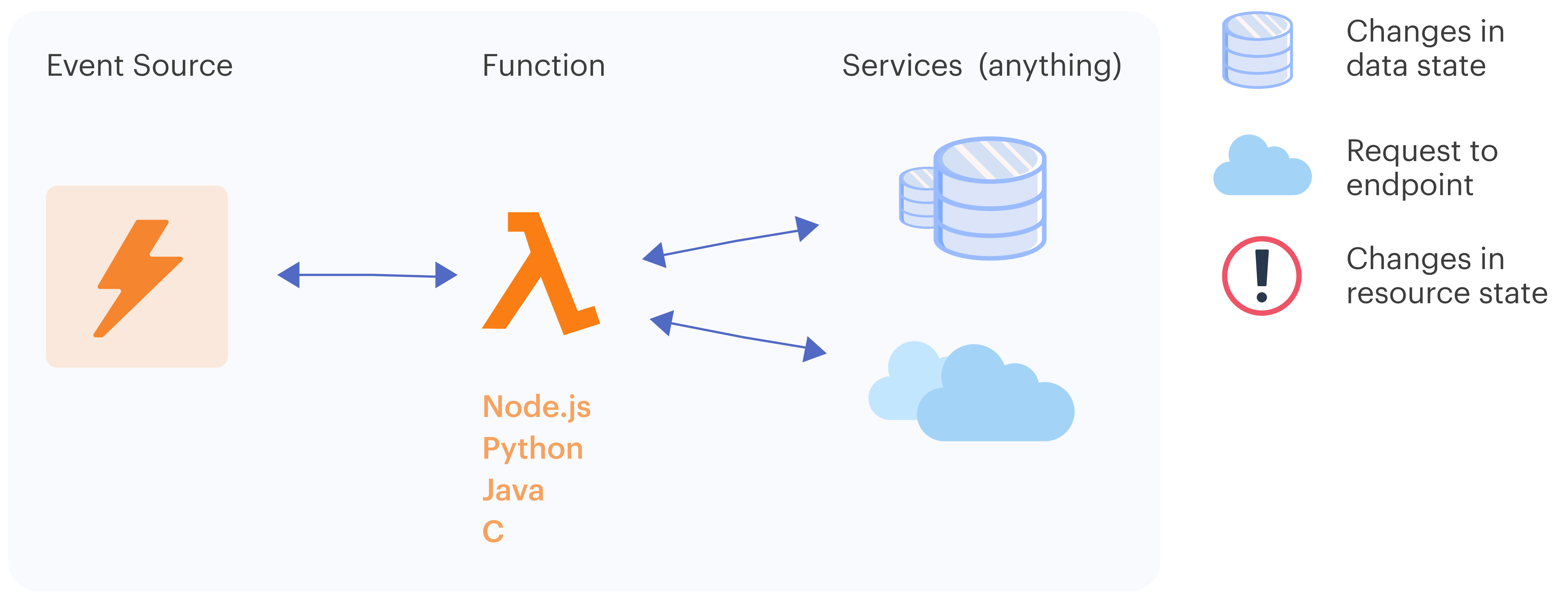
Creating CI/CD Pipeline

Step 1: Understanding CI/CD

Serverless Application Overview

Before considering developing a CI/CD pipeline for serverless apps, considering some of the paradigms is important. Lambda functions are a unit of deployment, and there will be multiple Lambda functions in a single application.

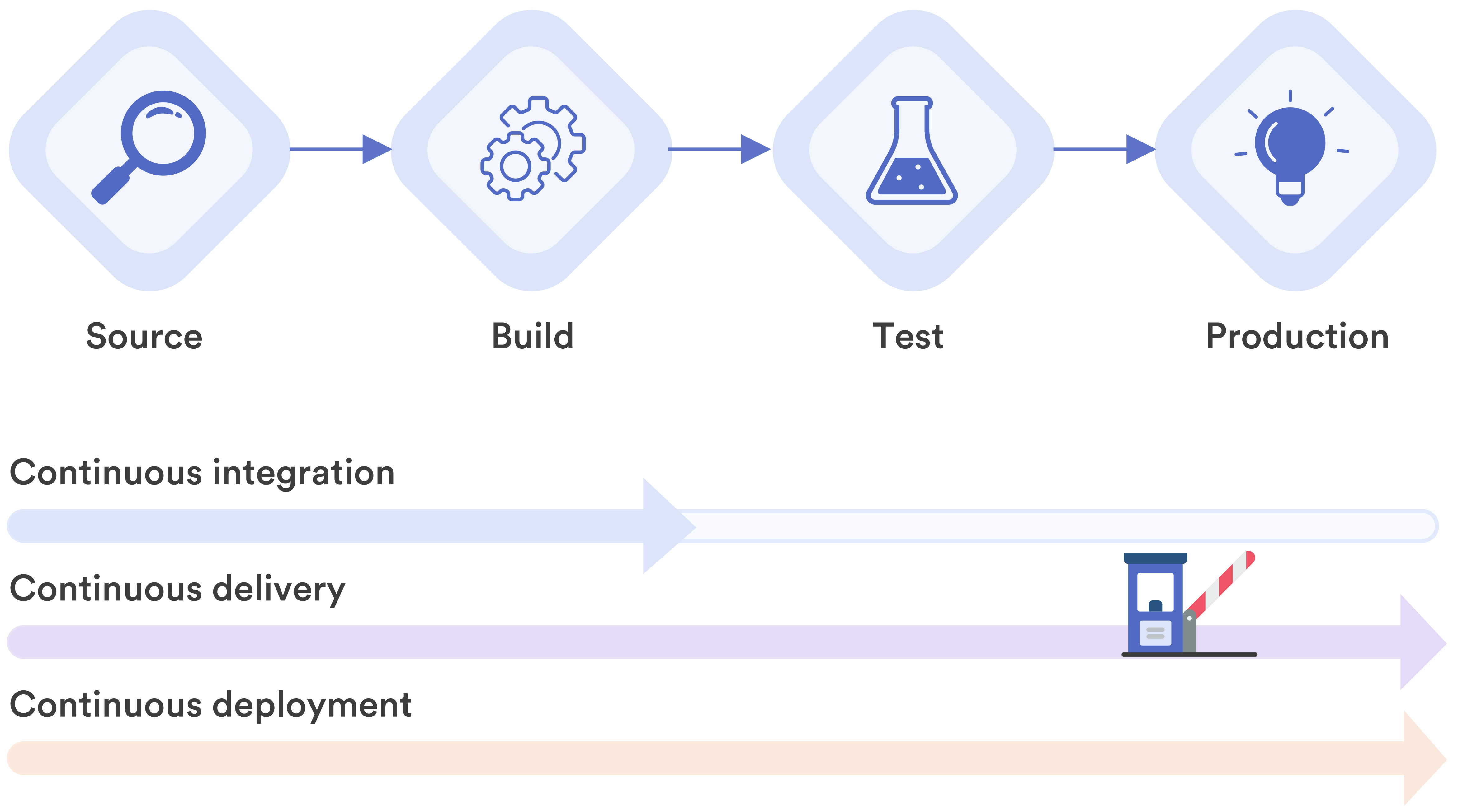
It typically includes an event source that will trigger your functions, which can either be shared or unique for each part. This function, on the other side, will be communicating with the service.



Continuous Integration/ Continuous Delivery/Continuous Deployment

Let's start with how we talk about CI/CD at Simform. The whole development process is combined with four stages:

- Source stage where you write your code
- Build a stage where you will test the code and run it over with various tests
- Next is the testing stage where you execute your code and
- monitor its performance with other services, also called a pre-production environment.
- The final step is Production. This is where you ship your code into production and release it.



Continuous integration is the idea that you are continuously committing code and being pushed through the system for the build stage. **Continuous delivery** typically involves the automation further past the continuous integration, out through test and production with a gate (either manual or logic). Lastly, **continuous deployment** is a holy grail of the whole automation as an end of this process with no barriers involved.

Use Cases

Apart from that, CI/CD pipelines highly depend on use cases which can fall under six primary categories:

Web Apps
- Static websites
- Complex web apps
- Packages for flask and express

IT automation
- Policy engines
- Extending Services
- Infrastructure Management

Backends
- Apps & services

Chatboats
- Powering chatbot logic

Data Processing
- Real time
- MapReduce
- Batch

Amazon Alexa
- Powering voice-enabled apps
- Alexa skills kit

Each of these six different use cases will have unique architecture and services they will interface with. However, they are powered by AWS Lambda. If we think about creating CI/CD for all these use cases, it will be different for each of them, and there are many other options that we need to take into account. Hence, we will discuss an everyday use case in this chapter.

Step 2: Necessary Tools

Apart from that, CI/CD pipelines highly depend on use cases which can fall under six primary categories:

- Build tools
- Testing tools
- Deploy tools
- Infrastructure management tools
- Lifecycle automation tools

Building our package

For building the deployment package, there are four major languages. You need to create different deployment packages to push it up in the Lambda for your choice of language.

01 **Node.js & Python:** With these languages, you need to create a .zip file consisting of your code and other dependencies of the root level. You may use npm/pip to install those libraries.

02 **Java:** For Java, you can upload a .zip file or a standalone jar with compiled class & resources at root level. You may use Maven/ Eclipse IDE plugins.

SIMFORM

03 C# (.NET Core): With C#, you can either push a .zip file with all the code/dependencies or a standalone .dll file. You may use NuGet/VisualStudio plugins.

Building a Testing Validation Model

After completion of building your deployment package, there are certain things that you will need to verify in the code. Here we have created a testing validation model for reference-

Make sure the code:

- is without syntax issues
- meets company standards for format
- compiles
- is sufficiently tested at the code level via unit tests

Next step involves checking if the serverless service:

- functions as it is supposed to, concerning other components
- has appropriate mechanisms to handle failures up or downstream

Deploying a Serverless App

Thereafter comes the deployment of all these sections. There are two options available over here- either you can go service by service, develop packages, and upload Lambda functions or use a more structured tool- SAM (Serverless Application Model).

SAM is an extension to CloudFormation that allows you to write templates for deploying serverless apps in a simplified manner. It supports almost everything which CloudFormation supports, including Lambda Functions, API Gateway & DynamoDB.

Step 3: Configuring Multiple Environments

A sound DevOps engineer knows why it is essential to have different building environments, test, and run their application. Additionally,

Why?

- To avoid overlapping usage of resources
- Safely test new code without impacting your users
- Safely test infrastructure changes

How?

- through AWS account strategies
- using infrastructure as code tools
- using variables unique to each environment
- automation/application delivery testing

Step 4: Delivery via CodePipeline

The next step is to pull it all together and automate the process. You may use AWS CodePipeline, which has some fantastic benefits:

- Continuous delivery service for fast and reliable application updates
- Model and visualize your software release process.
- Build, test and deploy your code every time there is a code change
- Integrates with third-party tools and AWS

Here's a basic pipeline flow that you can follow if you're using CodePipeline

- Commit your code to a source code repository
- Package/Test in CodeBuild

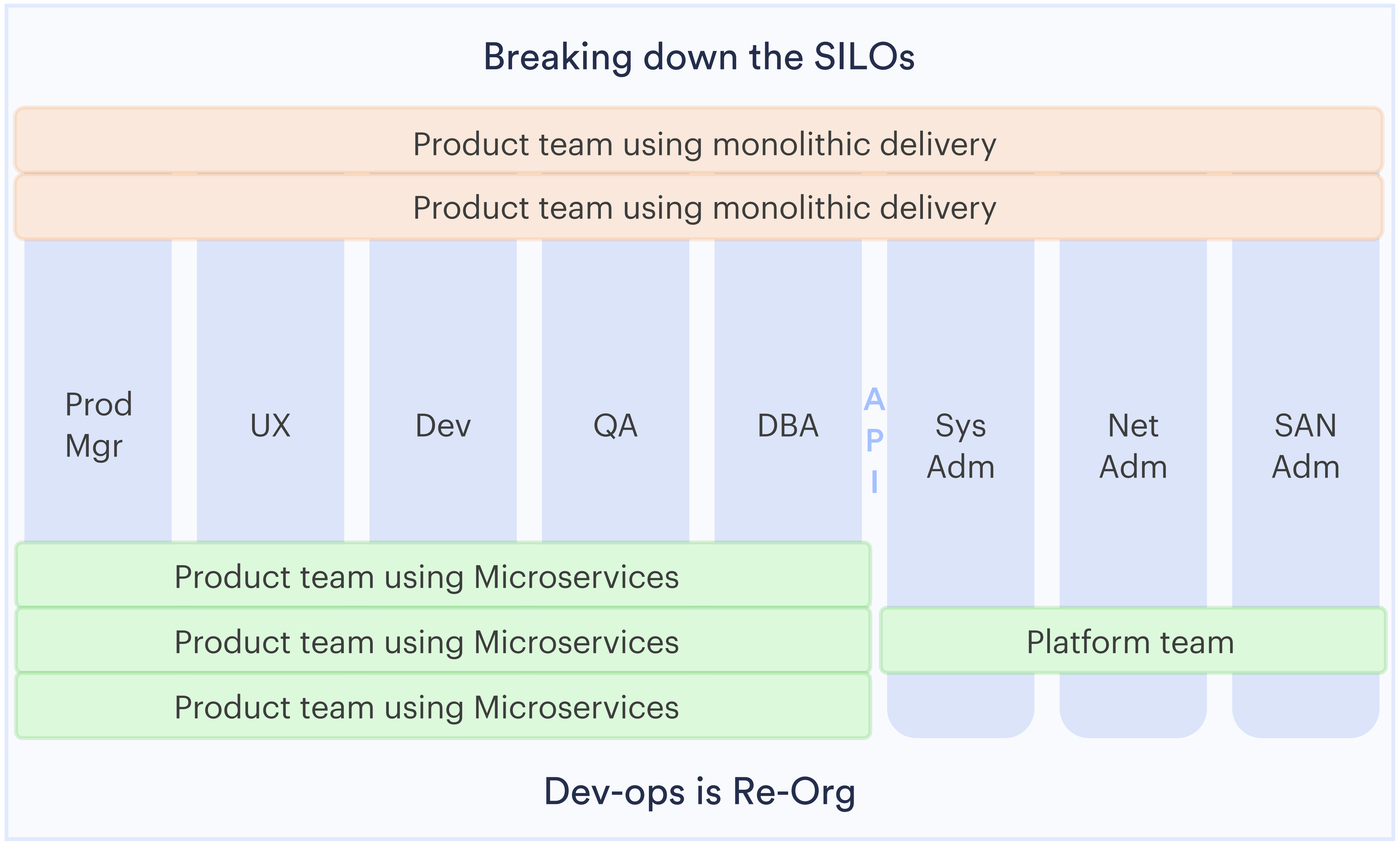
- Use CloudFormation actions in CodePipeline to create or update stacks via SAM templates.
- Make use of specific stage/environment parameter files to pass in Lambda variables.
- Test your application between stages/environments

Managing People & Processes

The adoption of serverless architecture involves more than just the incorporation of new technology. You will be required to adopt new processes and development team dynamics to make the transition hassle-free over time.

Often Team Leaders break applications down by technology domain and assign responsibilities accordingly to different teams. With serverless architecture, applications are broken down by services grouped based on business capabilities.

All software such as UI/UX, external connections, and storage are incorporated equally amongst each business domain. Each team handles the full application development cycle from UI to the database.


Breaking down the SILOs
Product team using monolithic delivery
Product team using monolithic delivery
Prod Mgr
UX
Dev
QA
DBA
API
Sys Adm
Net Adm
SAN Adm
Product team using Microservices
Product team using Microservices
Product team using Microservices
Platform team
Dev-ops is Re-Org

This change in structure affects the people working within it. Development teams, accustomed to legacy systems, may have difficulty migrating from a mono-language environment to a multi-language one. Serverless frees them up to be more autonomous and accountable, considering the 'big picture.'

However, for developers working with the old ways, operating in the serverless landscape can be overwhelming. You must continuously be iterating and analyzing the ability of your team to evolve successfully. It will take time to adopt the new guidelines, procedures, and working style.

Devs Must Evolve

Along with migration or organizations from monolithic/microservices to serverless architecture, developers must also evolve. Since it is easy to deploy functions in serverless architecture, developers are also expected to get involved with the production and monitoring environment.

It might come off as contrasting with the monolithic approach where developers just write and throw the code ‘over the fence’ for the operations and testing team to deploy and maintain. We move towards a serverless DevOps approach with development teams, IT operations, and Quality Assurance teams merged into tiny groups. They are collectively responsible for the three main tasks: development, deployment, and monitoring.

Serverless is changing how team structure is fabricated, allowing organizations to build teams centered over different services. This also facilitates autonomy and responsibility in a constrained area. The approach empowers organizations to adjust rapidly to ever-changing business demands without any interruptions to the core activities.

Devs are expected to handle these further challenges:

01 Ability to help companies in integrating technology with business planning and strategy.

02 Helping business leaders in understanding how open APIs can open new opportunities for business lines in the marketplace.

03 How to choose the right serverless technology, simplify the development stack, and push back when vendors necessitate unproductive middlewares.

04 Learn and continuously update from serverless leaders like Netflix, Uber, etc. and decide which serverless model will be best for the organizations.

05 Be able to handle the responsibilities of operating and monitoring hundreds of functions at the same time.

06 They manage a complex network of teams, including cloud architects, operations, QA, and integrators that may not wholly understand the serverless approach.

Everything should be finetuned, from refactoring the data to configuring the code pipeline and even aligning your CI/CD with the serverless architecture. As we discussed the entire monolithic breakdown and its refactoring approach towards serverless, many practices can help you finetune your applications.

Chapter 4

Serverless Best Practices: Performance, Monitoring & Security

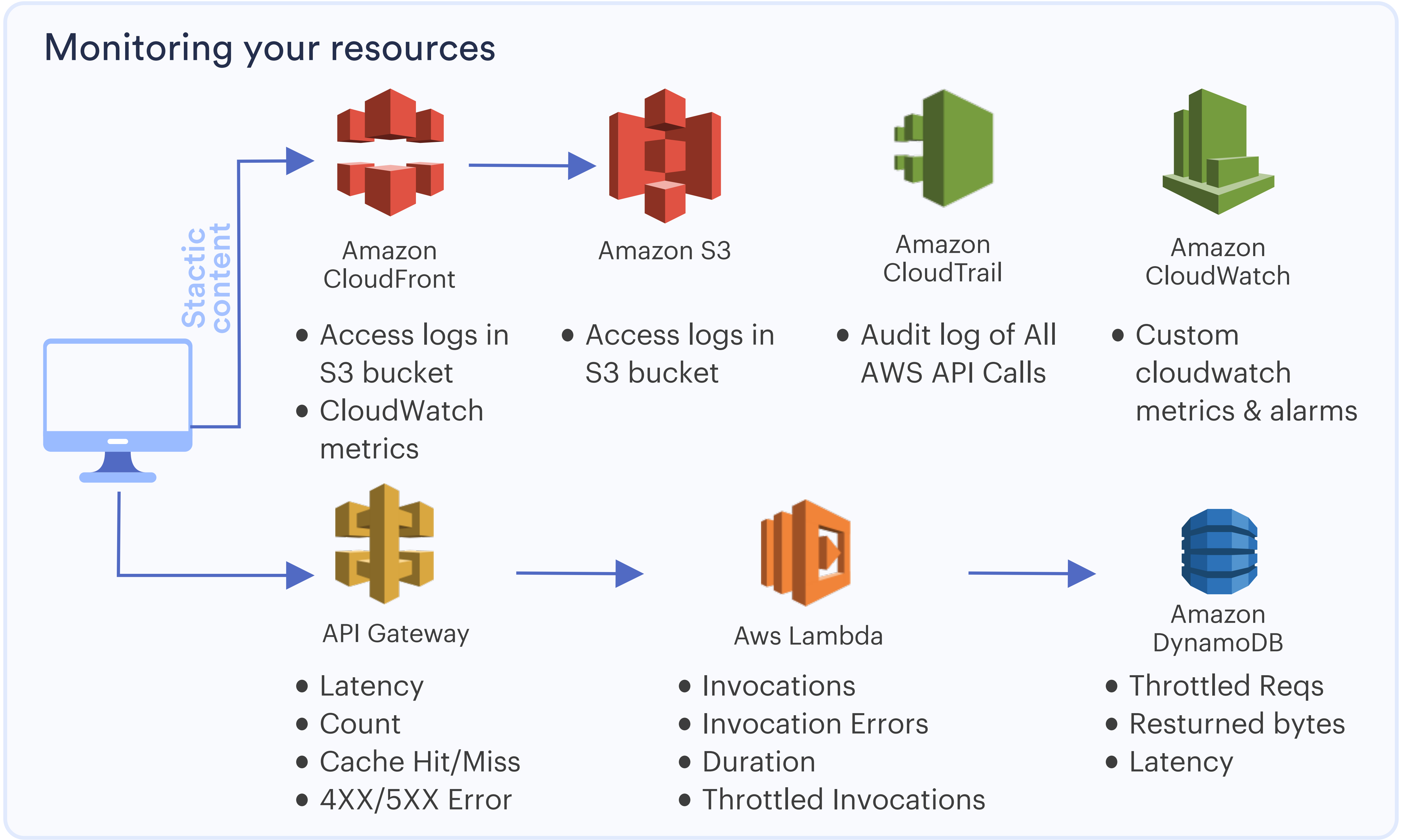
With non-serverless systems, baseline performance benchmarks don't change quickly. Hence, it is easy to predict how the system will behave in the future. With Serverless Architecture, it isn't the same.

Once you start playing with functions, you'll realize that it isn't easy to predict their behavior, especially performance. So, here we will discuss some of the best practices in implementation, monitoring, and security with context to a basic serverless application.

Note: This is an example with context to AWS Lambda and AWS's respective services. If you're using another service provider, things may vary accordingly.

Monitoring & Performance

It is essential to know what's happening with your application post-deployment. You should have clear visibility into whether things are performing well or not and the pain and friction points. AWS has native monitoring and logging functions.



Access logs for CloudFront & S3 can help you monitor which access your app and where are all those requests coming from. With CloudTrail, you can watch all the API calls within your account.

Most people refrain from analyzing these logs. However , a suggested practice is to store all these logs in S3 by default and have accident detection alarms for alerting and monitoring purposes. Also, with X-Ray, you can see and trace the relationship between Lambda Functions internally and externally.

Concurrency

Concurrency is one of the most critical factors, which is going to affect your functions' performance. For example, each transaction will invoke multiple tasks that might be separated by only a few microseconds for any necessary application.

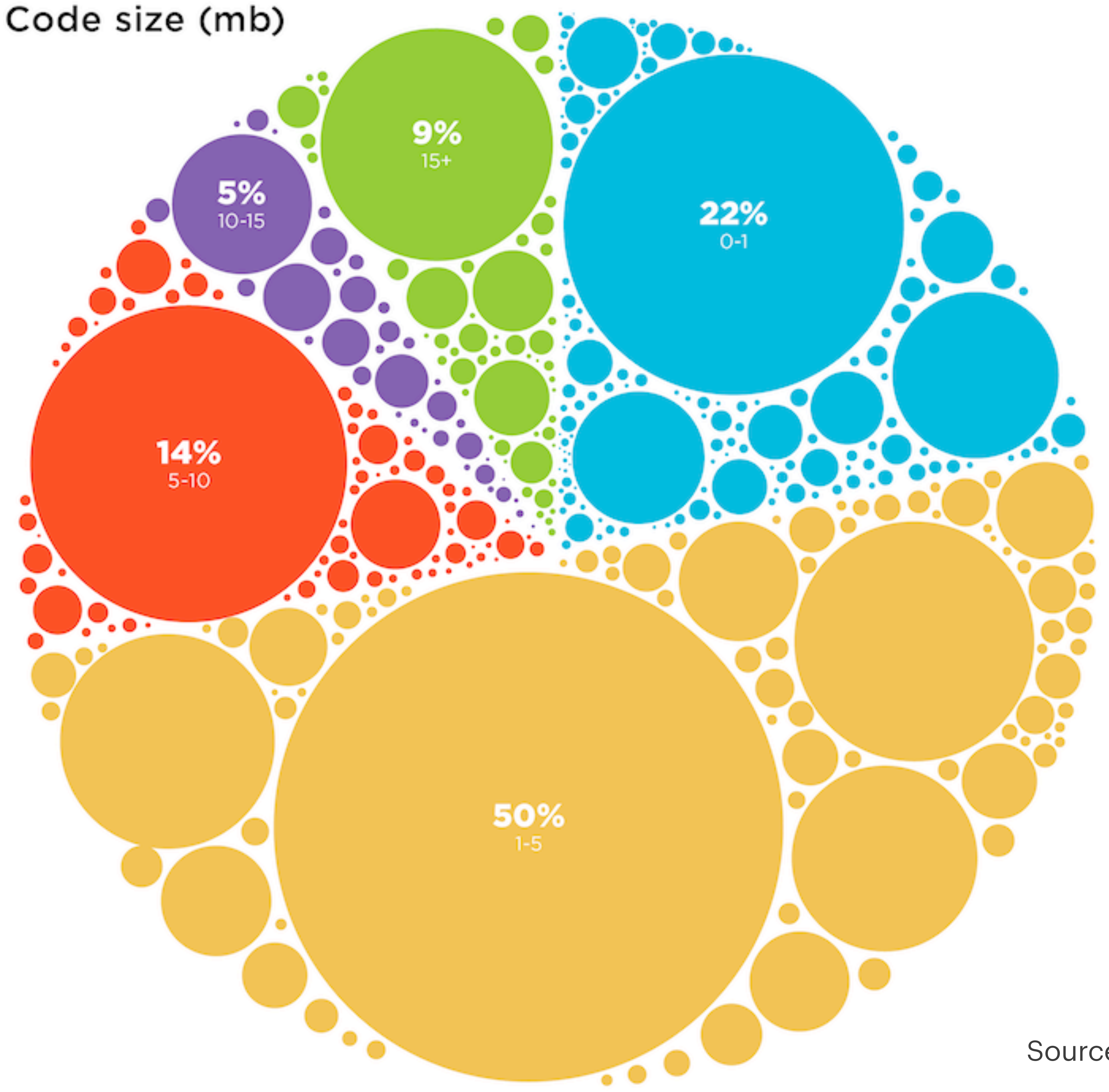
By default, AWS Lambda has a limit of 1000 concurrent executions, per account, per region. It might be able to handle short-term bursts that exceed this limit but will start throttling the further requests and return an HTTP status code 429.

It'd be better if you have a good understanding of your application's requirements and performance. In any case, adequate monitoring would suffice to detect accidental throttling.

Package Size

AWS Lambda currently supports 50MB as the maximum size of the code packages. With the increase in the package size, the time it takes to build and deploy will also increase. Not only that, it will have its footprints over start-up time and may contribute to more significant latencies.

Code size (mb)

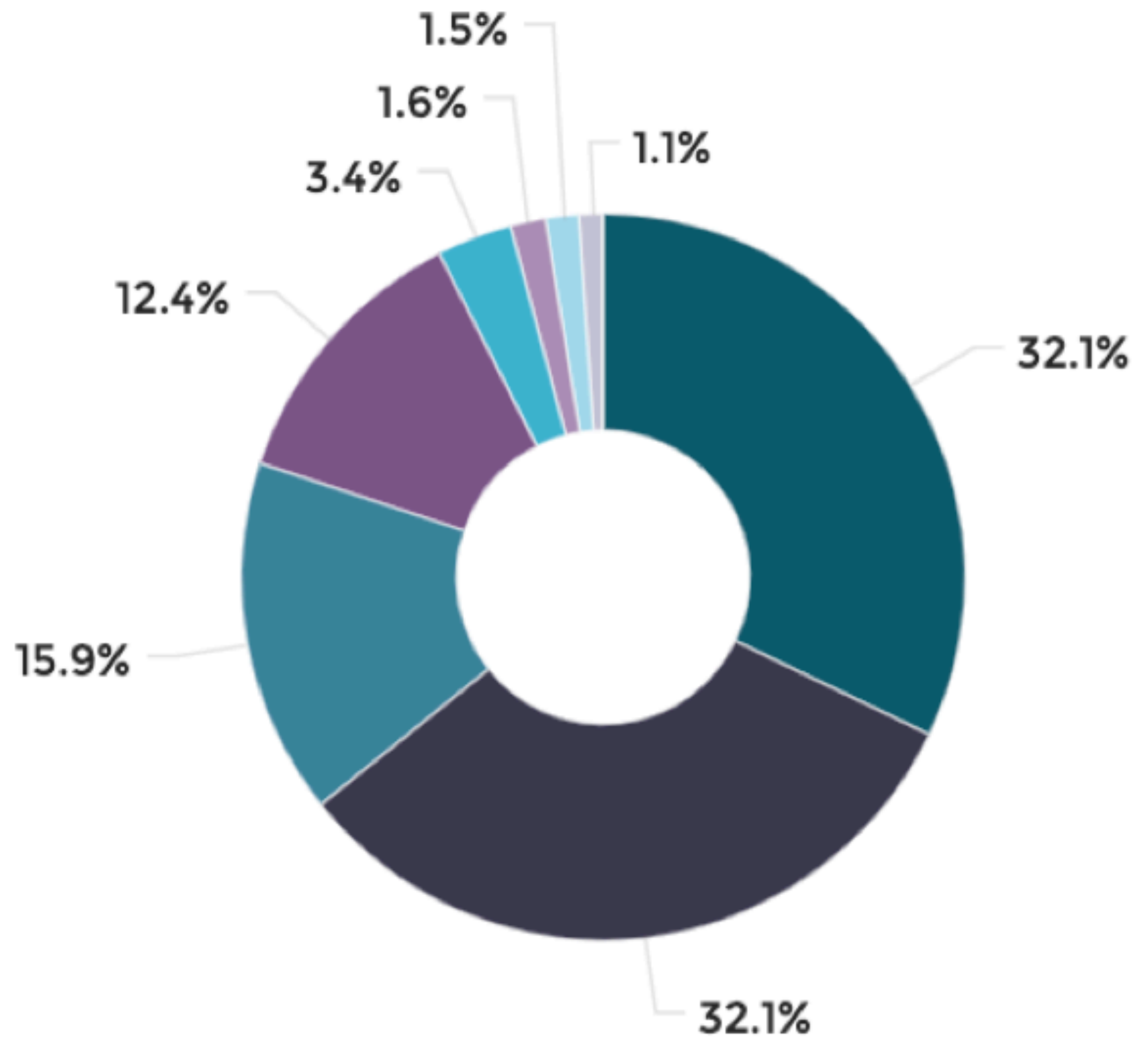


Source: : State of Serverless Report, 2017.

In general, it is better if you look out for your dependencies and limit them to a minimum so that you have smaller package sizes and short start-up times. Your language preference will also impact your package size, which we will discuss in the next point.

Language Selection

AWS Lambda supports code written in Node.js (JavaScript), Python, Java (Java 8 compatible), and C# (.NET Core), and Go. Your code can include existing libraries, even native ones. These are good options and should fit seamlessly with your development team's skillset.



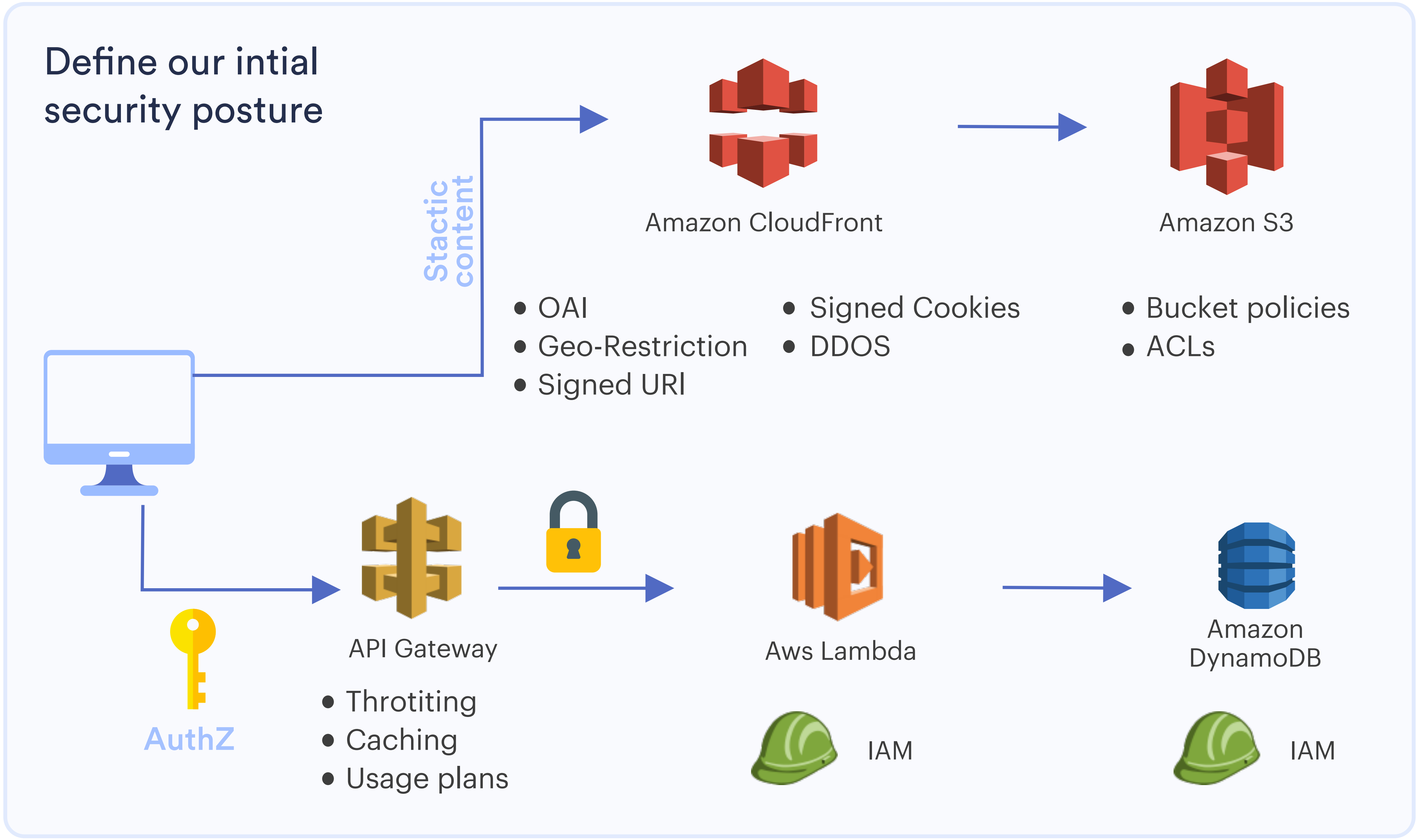
●	128 MB	32.1%
●	1024MB	32.1%
●	512MB	15.9%
●	256MB	12.4%
●	1536MB	3.4%
●	2048MB	1.6%
●	384MB	1.5%
●	3008MB	1.1%

Source: : Average Scripting Language, Runtimes, 2019.

Scripting languages like Node.js and Python are the most commonly used language runtime. Python and Node.js are faster than Java and C# if performance and start-up time are your primary concerns and lean towards the former ones.

Security

Security is one of the first questions that surface in the mind while developing an application. It is good, as one should think about safety, not after the app is running but while designing, building it, and considering to operate it. A good security posture is absolutely critical.



The best practice is to use OAI to control where your users come from and geo-based restrictions to limit who can access your app. With S3, you can create strict Bucket Policies and object ACLs. Lastly, you need IAM to secure AWS Lambda and Dynamodb.

01. Security Perimeter & Granularity

With serverless architecture, there is more granularity to the development process. It empowers you with more flexibility, though, one should note that with more granularity comes more surface area, which means an increase in attack surface.

Also, the security perimeter is redefined. With a traditional architecture, all the functionalities are tied up, and you will have to monitor them as one; however, with serverless architecture, each function has its security perimeter. It forces a higher responsibility on development teams to control and observe what each part can and cannot do.

02. Statelessness of Functions

The statelessness of functions is one of the most significant security features of serverless architecture. Typical security breaches occur over time, with the malicious agents being implanted in the system.

Since functions are naturally ephemeral and die once their execution is finished, it is not fruitful for attackers to implant agents, which unfolds over time. Due to this, attackers need to do it repeatedly, which has good chances of risk detection.

03. Monitoring of Unused Functions

With serverless architecture, it is easy to write small packages and deploy them. That's where most of the people fall into the trap and end up creating functions at a rapid rate. Another reason they often cite is they cannot pay unless it is being used. It is true!

This results in several functions that are often difficult to monitor. This happens because once you have added functions, it's quite hard to remove them.

Due to a lack of precise monitoring tools, it is hard to track which parts are being used. It creates a bundle of stale functions with vulnerable dependencies, and this isn't something you'd like for your application.

SIMFORM

Concluding remarks...

The ebook is an ignitor to spark a discussion on how migration to serverless is not just a business decision! It has many layers to it, and we intend to encourage CTOs, CEOs, and technical peers to have a healthy dialogue. It will help businesses create efficient, smooth, and seamless migration to a serverless architecture.

Going through the ebook, and even outside elements of serverless, whatever may be your confusion, get in touch with me **hiren@simform.com** or social platforms on **Twitter** or **LinkedIn**.

Hiren Dhaduk
VP of Technology
hiren@simformlabs.com

We are Simform!

With over 10+ years of experience under our belt, we are more than ready to supercharge your project with extraordinary code. 10 years ago, Simform was one person. Today, we're Over 300 people strong and growing.

Simform is a custom software development powerhouse. Let's get in touch to discuss your next project!

Contact Us

Custom Software Development Services

Software Testing Services

Mobile Application Development Services

Dedicated Software Development Team

Web Application Development Services

Software Product Development Services

API Integration Services

Application Development Services

Hire Dedicated Developers

AI/ML Development Services

SIMFORM