Chapter 1. Going serverless

This chapter covers

- Traditional system and application architectures
- Key characteristics of serverless architectures and their benefits
- How serverless architectures and microservices fit into the picture
- Considerations when transitioning from server to serverless

If you ask software developers what software architecture is, you might get answers ranging from "it's a blueprint or a plan" to "a conceptual model" to "the big picture." It's undoubtedly true that architecture, or lack thereof, can make or break software. Good architecture may help to scale a web or mobile application, and poor architecture may cause serious issues that necessitate a costly rewrite. Understanding the implication of choice regarding architecture and being able to plan ahead is paramount to creating effective, high-performing, and ultimately successful software systems.

This book is about how to go beyond traditional back-end architectures that require you to interact with a server in some shape or form. It describes how to create *serverless* back ends that rely entirely on a compute service such as Amazon Web Services (AWS) Lambda and an assortment of useful third-party APIs, services, and products. It shows how to build the next generation of systems that can scale and handle demanding computational requirements without having to provision or manage a single server. Importantly, this book describes techniques that can help developers quickly deliver products to market while maintaining a high level of quality and performance by using services and architectures that today's cloud has to offer.

The first chapter of this book is about why we think serverless is a game changer for software developers and solution architects. This chapter introduces key services such as AWS Lambda and describes the principles of serverless architecture to help you understand what makes a true serverless system.

What's in a name?

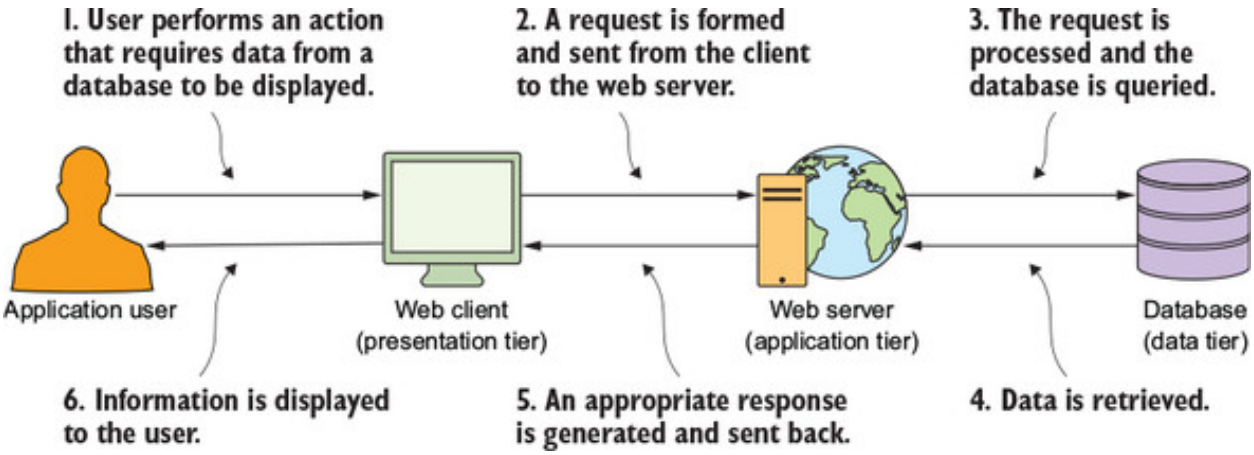
Before we start, we should mention that the word *serverless* is a bit of a misnomer. Whether you use a compute service such as AWS Lambda to execute your code, or interact with an API, there are still servers running in the background. The difference is that these servers are hidden from you. There's no infrastructure for you to think about and no way to tweak the underlying operating system. Someone else takes care of the nitty-gritty details of infrastructure management, freeing your time for other things. Serverless is about running code in a compute service and interacting with services and APIs to get the job done.

1.1. HOW WE GOT TO WHERE WE ARE

If you look at systems powering most of today's web-enabled software, you'll see back-end servers performing various forms of computation and client-side front ends providing an interface for users to operate via their browser, mobile, or desktop device.

In a typical web application, the server accepts HTTP requests from the front end and processes requests. Data might travel through numerous application layers before being saved to a database. The back end, finally, generates a response—it could be in the form of JSON or fully rendered markup—which is sent back to the client (figure 1.1). Naturally, most systems are more complex once elements such as load balancing, transactions, clustering, caching, messaging, and data redundancy are taken into account. Most of this software requires servers running in data centers or in the cloud that need to be managed, maintained, patched, and backed up.

Figure 1.1. This is a basic request-response (client-server) message exchange pattern that most developers are familiar with. There's only one web server and one database in this figure. Most systems are much more complex.



Provisioning, managing, and patching of servers is a time-consuming task that often requires dedicated operations people. A non-trivial environment is hard to set up and operate effectively. Infrastructure and hardware are necessary components of any IT system, but they're often also a distraction from what should be the core focus—solving the business problem.

Over the past few years, technologies such as platform as a service (PaaS) and containers have appeared as potential solutions to the headache of inconsistent infrastructure environments, conflicts, and server management overheard. PaaS is a form of cloud computing that provides a platform for users to run their software while hiding some of the underlying infrastructure. To make effective use of PaaS, developers need to write software that targets the features and capabilities of the platform. Moving a legacy application designed to run on a standalone server to a PaaS service often leads to additional development effort because of the ephemeral nature of most PaaS implementations. Still, given a choice, many developers would understandably choose to

use PaaS rather than more traditional, more manual solutions thanks to reduced maintenance and platform support requirements.

Containerization is a way of isolating an application with its own environment. It's a lightweight alternative to full-blown virtualization. Containers are isolated and lightweight but they need to be deployed to a server—whether in a public or private cloud or onsite. They're an excellent solution when dependencies are in play, but they have their own housekeeping challenges and complexities. They're not as easy as being able to run code directly in the cloud.

Finally, we make our way to Lambda, which is a compute service from Amazon Web Services. Lambda can execute code in a massively parallelized way in response to events. Lambda takes your code and runs it without any need to provision servers, install software, deploy containers, or worry about low-level detail. AWS takes care of provisioning and management of their Elastic Compute Cloud (EC2) servers that run the actual code and provides a high-availability compute infrastructure—including capacity provisioning and automated scaling—that the developer doesn't need to think about. The words *serverless architectures* refer to these new kinds of software architectures that don't rely on direct access to a server to work. By taking Lambda and making use of various powerful single-purpose APIs and web services, developers can build loosely coupled, scalable, and efficient architectures quickly. *Moving away from servers and infrastructure concerns, as well as allowing the developer to primarily focus on code, is the ultimate goal behind serverless.*

1.1.1. Service-oriented architecture and microservices

Among system and application architectures, service-oriented architecture (SOA) has a lot of name recognition among software developers. It's an architecture that clearly conceptualized the idea that a system can be composed of many independent services. Much has been written about SOA, but it remains controversial and misunderstood because developers often confuse design philosophy with specific implementation and attributes.

SOA doesn't dictate the use of any particular technology. Instead, it encourages an architectural approach in which developers create autonomous services that communicate via message passing and often have a schema or a contract that defines how messages are created or exchanged. Service reusability and autonomy, composability, granularity, and discoverability are all important principles associated with SOA.

Microservices and serverless architectures are spiritual descendants of service--oriented architecture. They retain many of the aforementioned principles and ideas while attempting to address the complexity of old-fashioned service-oriented architectures.

On microservices

There has been a recent trend to implement systems with microservices. Developers tend to think of microservices as small, standalone, fully independent services built around a particular business purpose or capability.

Ideally, microservices should be easy to replace, with each service written in an appropriate framework and language. The mere fact that microservices can be written in different general-purpose or domain-specific languages (DSL) is a drawing card for many developers. Benefits can be gained from using the right language or a specialized set of libraries for the job. Nevertheless, it can often be a trap, too. Having a mix of languages and frameworks can be hard to support, and, without strict discipline, can lead to confusion down the road.

Each microservice can maintain state and store data. And if microservices are correctly decoupled, development teams can work and deploy microservices independently of one another. On the other hand, eventual consistency, transaction management, and complex error recovery can make things more difficult (especially without a sound plan).

It can be argued that serverless architecture embodies many principles from microservices too. After all, depending on how you design the system, every compute function could be considered to be its own standalone service. But you don't need to fully embrace the microservices mantra if you don't want to.

Serverless architectures give you the freedom to apply as few or as many microservice principles as you would like without forcing you down a single path. This book shows examples of architectures where parts of a monolithic system are re-implemented as serverless architecture without applying all of the microservices tenets. It's then up to you to decide how far to take your architecture based on your requirements and preference (chapter 10 has more to say on the issue of microservices and design).

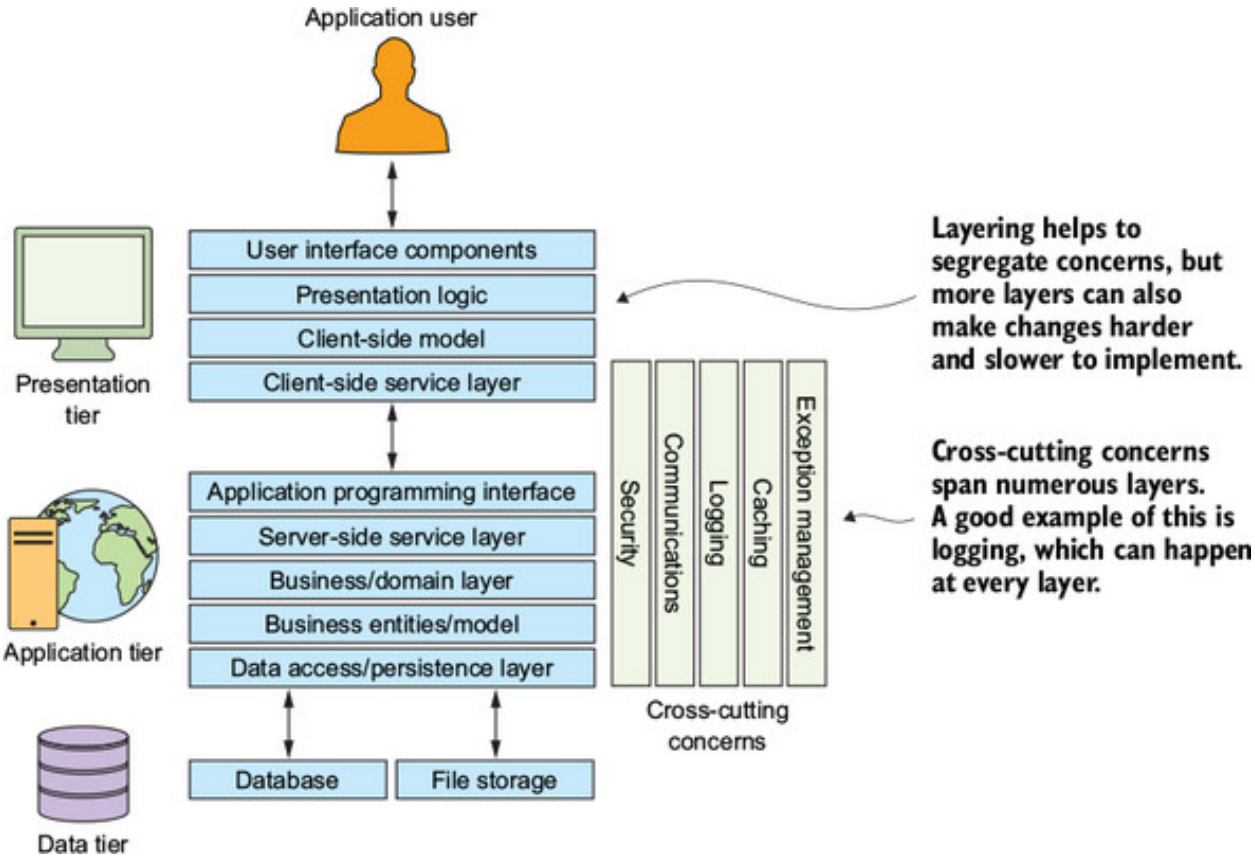
1.1.2. Software design

Software design has evolved from the days of code running on a mainframe to multitier systems where the presentation, data, and application/logic tiers feature prominently in many designs. Within each tier there may be multiple logical layers that deal with particular aspects of functionality or domain. There are also cross-cutting components, such as logging or exception-handling systems, that can span numerous layers. The preference for layering is understandable. Layering allows developers to decouple concerns and have more maintainable applications.

But the converse can also be true. Having too many layers can lead to inefficiencies. A small change can often cascade and cause the developer to modify every layer

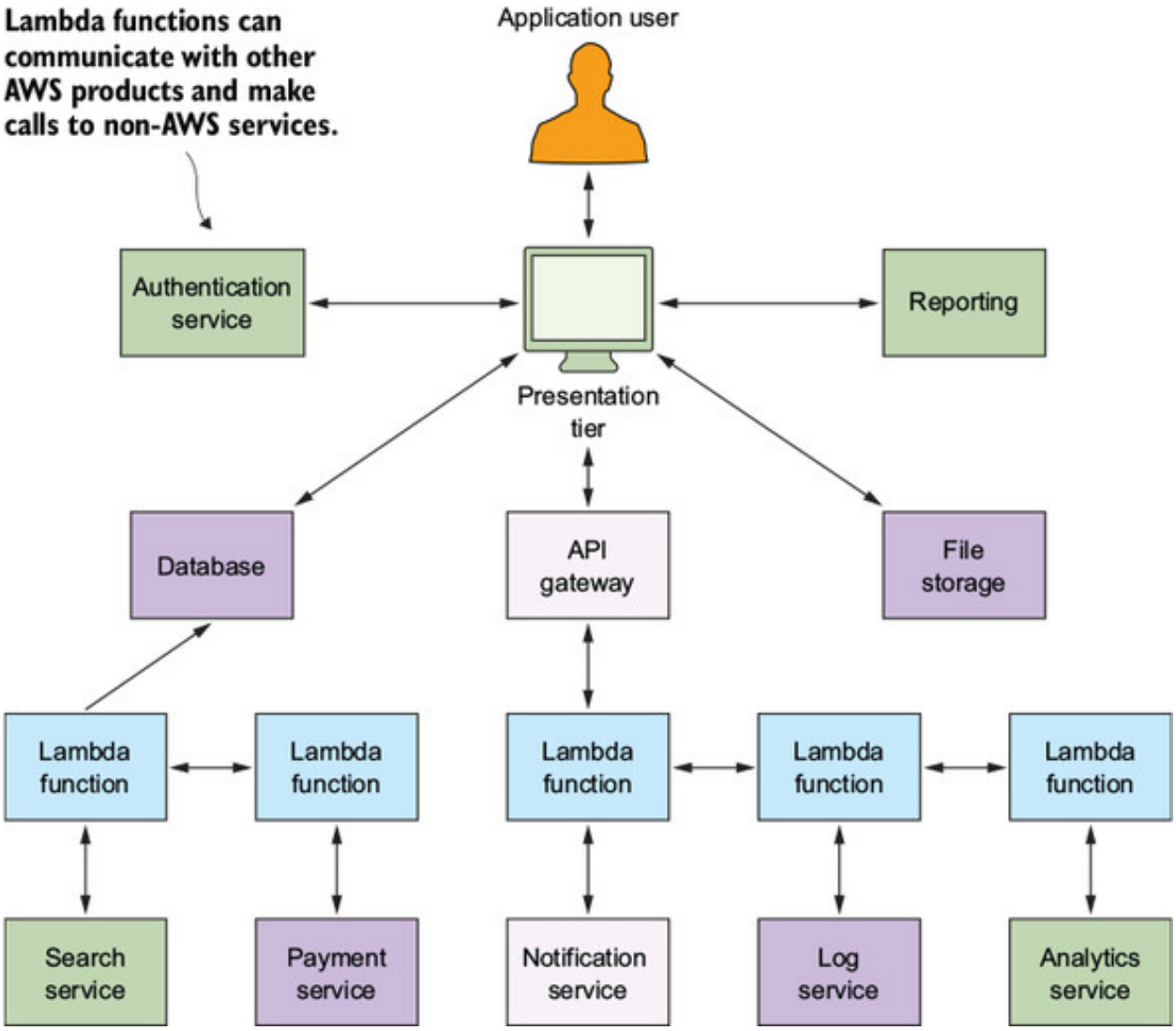
throughout the system, costing considerable time and energy in implementation and testing. The more layers there are, the more complex and unwieldy the system might become over time. Figure 1.2shows an example of a tiered architecture with multiple layers.

Figure 1.2. A typical three-tier application is usually made up of presentation, application, and data tiers. A tier may have multiple layers with specific responsibilities.



Serverless architectures can help with the problem of layering and having to update too many things. There's room for developers to remove or minimize layering by breaking the system into functions and allowing the front end to securely communicate with services and even the database directly, as shown in figure 1.3. All of this can be done in an organized way to prevent spaghetti implementations and dependency nightmares by clearly defining service boundaries, allowing Lambda functions to be autonomous, and planning how functions and services will interact.

Figure 1.3. In a serverless architecture there's no single traditional back end. The front end of the application communicates directly with services, the database, or compute functions via an API gateway. Some services, however, must be hidden behind compute service functions, where additional security measures and validation can take place.



A serverless approach doesn't solve all problems, nor does it remove the underlying intricacies of the system. But when implemented correctly it can provide opportunities to reduce, organize, and manage complexity. A well-planned serverless architecture can make future changes easier, which is an important factor for any long-term application. The next section and later chapters discuss the organization and orchestration of services in more detail.

Tiers vs. layers

There is confusion among some developers about the difference between layers and tiers. A *tier* is a module boundary that exists to provide isolation between major components of a system. A presentation tier that's visible to the user is separate from the application tier, which encompasses business logic. In turn, the data tier is another separate system that can

manage, persist, and provide access to data. Components grouped in a tier can physically reside on different infrastructures.

Layers are logical slices that carry out specific responsibilities in an application. Each tier can have multiple layers within it that are responsible for different elements of functionality such as domain services.

1.2. PRINCIPLES OF SERVERLESS ARCHITECTURES

Here we define five principles of serverless architectures that describe how an ideal serverless system should be built. Use these principles to help guide your decisions when building serverless applications:

1. Use a compute service to execute code on demand (no servers).
2. Write single-purpose stateless functions.
3. Design push-based, event-driven pipelines.
4. Create thicker, more powerful front ends.
5. Embrace third-party services.

Let's look at each of these principles in more detail.

1.2.1. Use a compute service to execute code on demand

Serverless architectures are a natural extension of ideas raised in SOA. In serverless architecture all custom code is written and executed as isolated, independent, and often granular functions that are run in a stateless compute service such as AWS Lambda. Developers can write functions to carry out almost any common task, such as reading and writing to a data source, calling out to other functions, and performing a calculation. In more complex cases, developers can set up more elaborate pipelines and orchestrate invocations of multiple functions. There might be scenarios where a server is still needed to do something. These cases, however, may be far and few between, and as a developer you should avoid running and interacting with a server if possible.

So, what is Lambda exactly?

AWS Lambda is a compute service that executes code written in JavaScript (node.js), Python, C#, or Java on AWS infrastructure. Source code (JARs or DLLs in case of Java or C#) is zipped up and deployed to an isolated container that has an allocation of memory, disk space, and CPU. The combination of code, configuration, and dependencies is typically referred to as a *Lambda function*. The Lambda runtime can invoke a function multiple times

in parallel. Lambda supports push and pull event models of operation and integrates with a large number of AWS services. Chapter 6 covers Lambda in more detail, including its event model, methods of invocation, and best practice with regard to design. Note that Lambda isn't the only game in town. Microsoft Azure Functions, IBM Bluemix, OpenWhisk, and Google Cloud Functions are other compute services you might want to look at.

1.2.2. Write single-purpose stateless functions

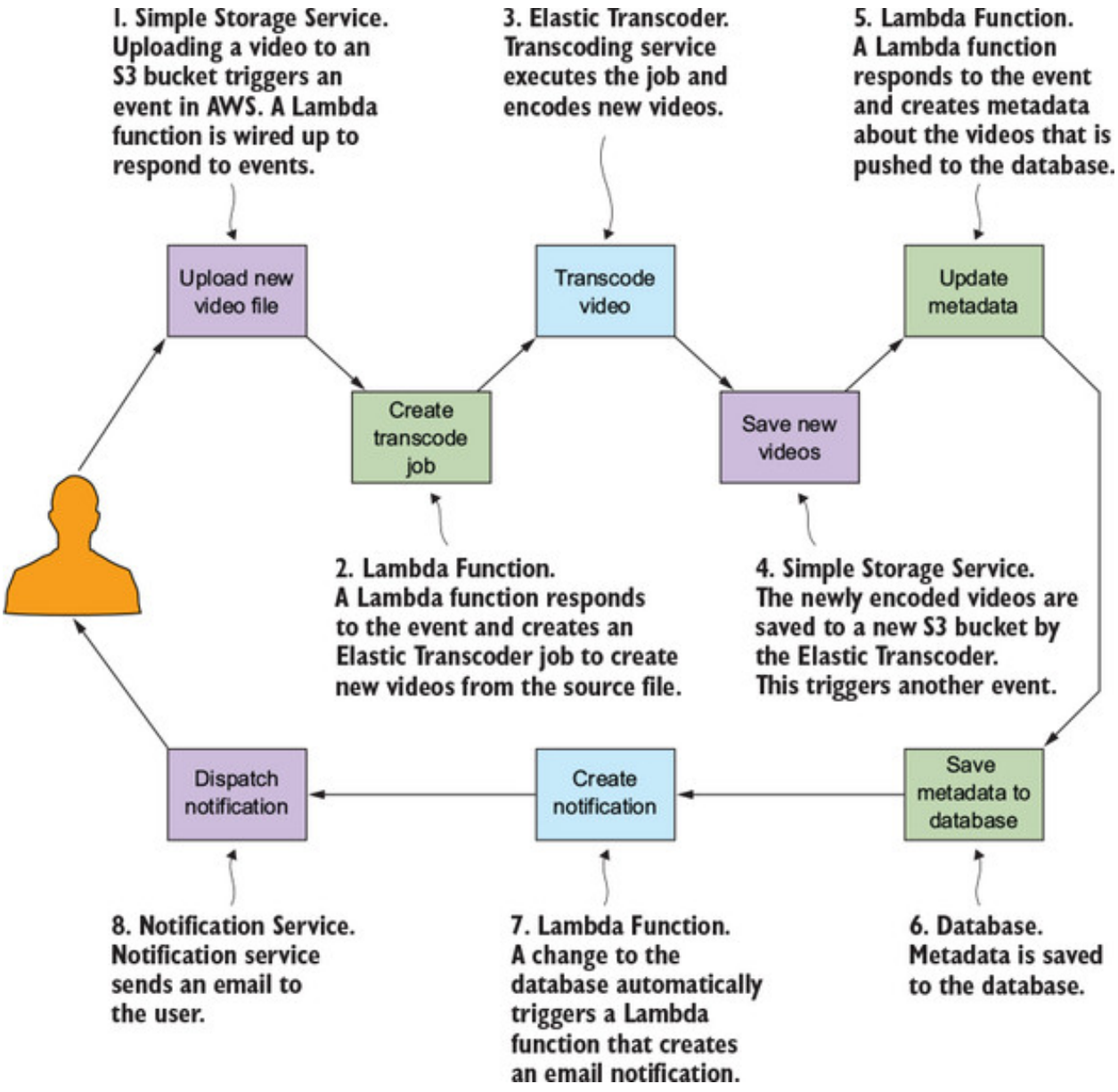
As a software engineer, you should try to design your functions with the single responsibility principle (SRP) in mind. A function that does just one thing is more testable and robust and leads to fewer bugs and unexpected side effects. By composing and combining functions and services in a loose orchestration, you can build complex back-end systems that are still understandable and easy to manage. A granular function with a well-defined interface is also more likely to be reused within a serverless architecture.

Code written for a compute service such as Lambda should be created in a *stateless* style. It must not assume that local resources or processes will survive beyond the immediate session (chapter 6 has more to say on this). Statelessness is powerful because it allows the platform to quickly scale to handle an ever-changing number of incoming events or requests.

1.2.3. Design push-based, event-driven pipelines

Serverless architectures can be built to serve any purpose. Systems can be built serverless from scratch, or existing monolithic applications can be gradually reengineered to take advantage of this architecture. The most flexible and powerful serverless designs are event-driven. In chapter 3, for example, you'll build an event-driven, push-based pipeline to see how quickly you can put together a system to encode video to different bitrates and formats. You'll achieve this by connecting Amazon's Simple Storage Service (S3), Lambda, and Elastic Transcoder together (figure 1.4).

Figure 1.4. A push-based pipeline style of design works well with serverless architectures. In this example a user uploads a video, which is transcoded to a different format.



Building event-driven, push-based systems will often reduce cost and complexity (you won't need to run extra code to poll for changes) and potentially make the overall user experience smoother. It goes without saying that although event-driven, push-based models are a good goal, they might not be appropriate or achievable in all circumstances. Sometimes you'll have to implement a Lambda function that polls the event source or runs on a schedule. We'll cover different event models and you'll work through examples in later chapters.

1.2.4. Create thicker, more powerful front ends

It's important to remember that custom code running in Lambda should be quick to execute. Functions that terminate sooner are cheaper because Lambda pricing is based on the number of requests, the duration of execution, and the amount of allocated memory. Having less to do in Lambda is cheaper. Moreover, building a rich front end (in lieu of a complex back end) that can talk to third-party services directly can be conducive to a better user experience. Fewer hops between online resources and reduced latency will result in a better perception of performance and usability of the application. In other words, you don't have to route everything through a compute service. Your front end may be able to communicate directly with a search provider, a database, or another useful API.

Digitally signed tokens can allow front ends to talk to disparate services, including databases, in a secure manner. This is in contrast to traditional systems where all communication flows through the back-end server.

Not everything, however, can or should be done in the front end. There are secrets that cannot be trusted to the client device. Processing a credit card or sending emails to subscribers must be done only by a service that runs outside the end user's control. In this case, a compute service must be used to coordinate action, validate data, and enforce security.

The other important point to consider is consistency. If the front end is responsible for writing to multiple services and fails midway through, it can leave the system in an inconsistent state. In this scenario, a Lambda function should be used because it can be designed to gracefully handle errors and retry failed operations.

1.2.5. Embrace third-party services

Third-party services are welcome to join the show if they can provide value and reduce custom code. It goes without saying, however, that when a third-party service is considered, factors such as price, capability, availability, documentation, and support must be assessed. It's far more useful for developers to spend time solving a problem unique to their domain than re-creating functionality already implemented by someone else. Don't build for the sake of building if viable third-party services and APIs are available. Stand on the shoulders of giants to reach new heights. Appendix A has a short list of Amazon Web Services and non-Amazon Web Services we've found useful. We'll look at most of those services in more detail as we move through the book.

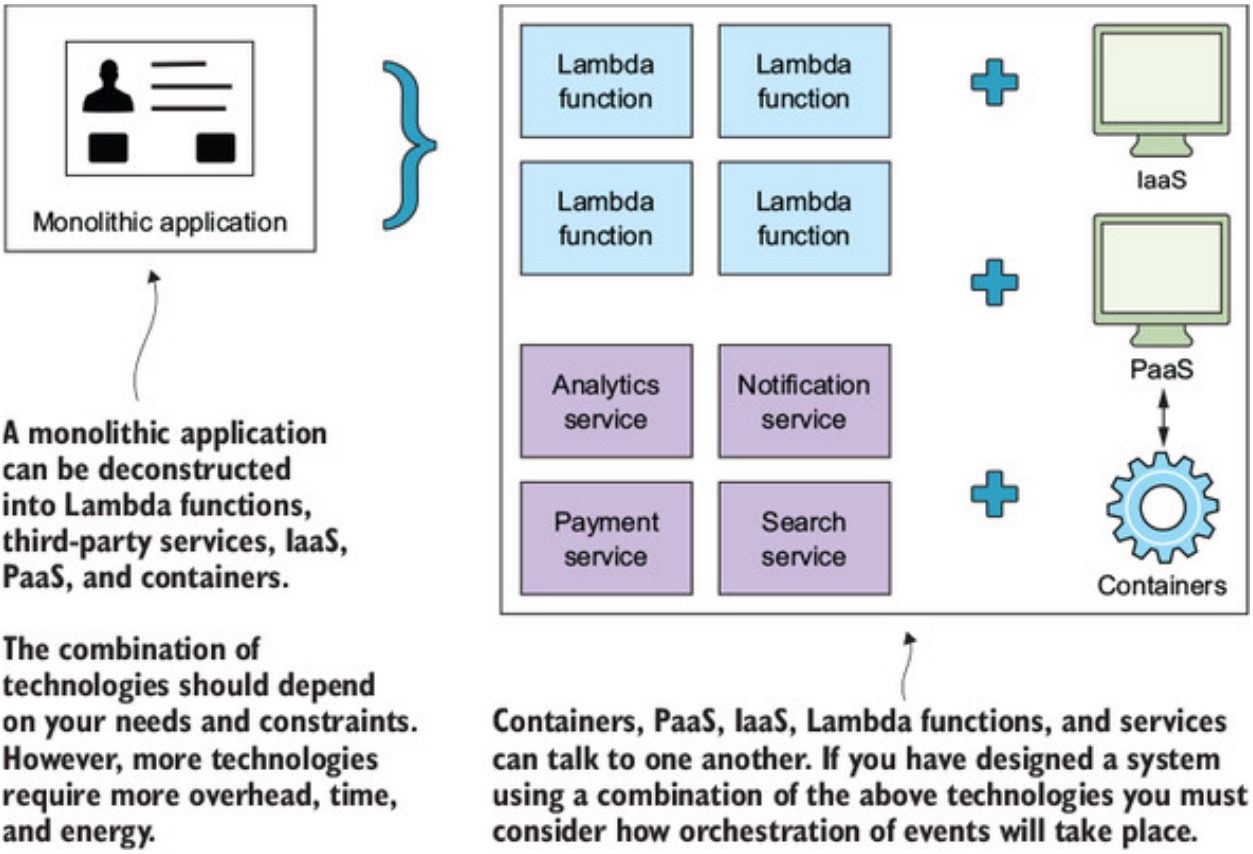
1.3. TRANSITIONING FROM A SERVER TO SERVICES

One advantage of the serverless approach is that existing applications can be gradually converted to serverless architecture. If a developer is faced with a monolithic code base,

they can gradually tease it apart and create Lambda functions that the application can communicate with.

The best approach is to initially create a prototype to test developer assumptions about how the system would function if it was going to be partly or fully serverless. Legacy systems tend to have interesting constraints that require creative solutions; and as with any architectural refactors at a large scale, there are inevitably going to be compromises. The system may end up being a hybrid—see figure 1.5—but it may be better to have some of its components use Lambda and third-party services rather than remain with an unchanged legacy architecture that no longer scales or that requires expensive infrastructure to run.

Figure 1.5. Serverless architecture is not an all-or-nothing proposition. If you currently have a monolithic application running on servers, you can begin to gradually extract components and run them in isolated services or compute functions. You can decouple a monolithic application into an assortment of infrastructure as a service (IaaS), PaaS, containers, Lambda functions, and third-party services if it helps.



The transition from a legacy, server-based application to a scalable serverless architecture may take time to get right. It needs to be approached carefully and slowly, and developers need to have a good test plan and a great DevOps strategy in place before they begin.

1.4. SERVERLESS PROS AND CONS

There are advantages to implementing a system as fully or partially serverless, including reduced cost and accelerated time to market. But you need to carefully consider the road to serverless architecture in the context of the application being created.

1.4.1. Decision drivers

Serverless is not a silver bullet in all circumstances. It may not be appropriate for latency-sensitive applications or software with specific service-level agreements (SLA). Vendor lock-in can be an issue for enterprise and government clients, and decentralization of services can be a challenge.

Not for everyone

Lambda runs in a public cloud, so mission-critical applications shouldn't necessarily be based on it. A banking system that performs high-volume transactions or a patient life-support system requires a higher level of performance and reliability than a public cloud system can provide. It's possible that organizations could employ dedicated hardware or run private or hybrid clouds with their own compute services that might meet serviceability and reliability requirements. In that case, these architectures could be adopted.

Service levels

AWS has an SLA for some services but not for others, so that may be a factor in your decision. For most systems, the reliability offered by AWS is sufficient, but some enterprise use cases may require additional guarantees. Non-AWS third-party services are in the same boat. Some may have strong SLAs, whereas others may not have one at all.

Customization

When it comes to Lambda, the efficiencies gained from having Amazon look after the platform and scale functions come at the expense of being able to customize the operating system or tweak the underlying instance. You can modify the amount of RAM allocated to a function and change timeouts, but that's about it (see chapter 6 for more information). Similarly, different third-party services will have varying levels of customization and flexibility.

Vendor lock-in

Vendor lock-in is another issue. If a developer decides to use third-party APIs and services, including AWS, there's a chance that architecture could become strongly

coupled to the platform being used. The implications of vendor lock-in and the risk of using third-party services—including company viability, data sovereignty and privacy, cost, support, documentation, and available feature set—need to be thoroughly considered.

Decentralization

Moving from a monolithic approach to a more decentralized serverless approach doesn't automatically reduce the complexity of the underlying system either. The distributed nature of the solution can introduce its own challenges because of the need to make remote rather than in-process calls and the need to handle failures and latency across a network.

1.4.2. When to use serverless

Serverless architecture allows developers to focus on software design and code rather than infrastructure. Scalability and high availability are easier to achieve, and the pricing is often fairer because you pay only for what you use. Importantly with serverless, you have a potential to reduce some of the complexity of the system by minimizing the number of layers and amount of code you need.

No more servers

Tasks such as server configuration and management, patching, and maintenance are taken care of by the vendor, which saves time and money. Amazon looks after the health of its fleet of servers that power Lambda. If you don't have specific requirements to manage or modify compute resources, then having Amazon or another vendor look after them is a great solution. You're responsible only for your own code, leaving operational and administrative tasks to a different set of capable hands.

Many uses

The statelessness and scalability of compute can be used to solve problems that benefit from parallel processing. Back ends for CRUD applications, e-commerce, back-office systems, complex web apps, and all kinds of mobile and desktop software can be built quickly using serverless architectures. Tasks that used to take weeks can be done in days or hours as long as the right combination of technologies is chosen. A serverless approach can work exceptionally well for startups that want to innovate and move quickly.

Low cost

The traditional server-based architecture requires servers that don't necessarily run at full capacity all of the time. Scaling, even with automated systems, involves a new server, which is often wasted until there's a temporary upsurge in traffic or new data.

Serverless systems are much more granular with regard to scaling and are cost--effective, especially when peak loads are uneven or unexpected. With Lambda you only pay for what you use (chapter 4 shows how to calculate cost for Lambda and the API Gateway).

Less code

We mentioned at the start of the chapter that serverless architecture provides an opportunity to reduce some of the complexity and code in comparison to more traditional systems. There's less need to have a multilayered back-end system, especially if you allow the front end to do more work and talk to services (and the database) directly.

Scalable and flexible

As a developer you don't need to use serverless architecture to replace your entire back end if you don't want to or are unable to do so. You can use Lambda to solve specific problems, especially if they stand to benefit from parallelization. It goes without saying that serverless systems can scale more easily than traditional systems. For example, consider the following solutions:

- ConnectWise, an IT services company, uses Lambda to process inbound logs, which has reduced their server maintenance needs from weeks to hours (https://aws.amazon.com/solutions/case-studies/connectwise/).
- Netflix uses Lambda to automate validation of backup completions and automate the encoding process of media files (https://aws.amazon.com/solutions/case-studies/netflix-and-aws-lambda/).

You can use Lambda for extract, transform, and load (ETL) jobs, real-time file processing, and virtually anything else without having to touch your existing codebase. Just write a function and run it.

1.5. SUMMARY

The cloud has been and continues to be a game changer for IT infrastructure and software development. Software developers need to think about the ways they can maximize use of cloud platforms to gain a competitive advantage.

Serverless architectures are the latest advance for developers and organizations to think about, study, and adopt. This exciting new shift in architecture will grow quickly as software developers embrace compute services such as AWS Lambda. And, in many cases, serverless applications will be cheaper to run and faster to implement.

There's also a need to reduce complexity and costs associated with running infrastructure and carrying out development of traditional software systems. The reduction in cost and time spent on infrastructure maintenance and the benefits of scalability are good reasons for organizations and developers to consider serverless architectures.

In this chapter you learned what serverless architecture is, looked at its principles, and saw how it compares to traditional architectures. In the next chapter, we'll explore important architectures and patterns, and we'll discuss specific use cases where serverless architectures were used to solve a problem.