**5**

**Managing the Enterprise Cloud Architecture**

In the previous chapters, we learned about different cloud technology strategies and started drafting a service model, including governance principles. Where do we go from here? From this point onward, you will be—as a business—managing your IT environments in multi-cloud. Successfully managing this new estate means that you will have to be very strict in maintaining the enterprise architecture. Hence, this chapter is all about maintaining and securing the multi-cloud architecture.

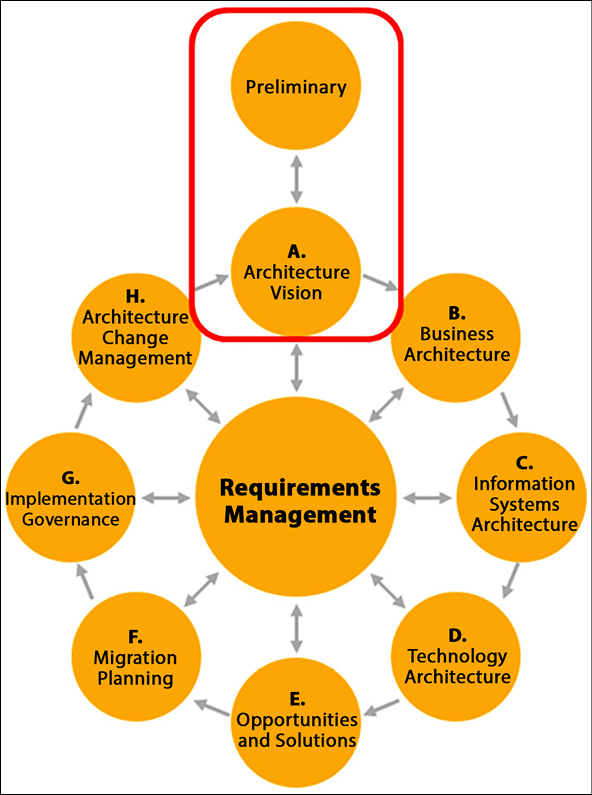
This chapter will introduce the methodology to create an enterprise architecture for multi-cloud using **The Open Group Architecture Framework** (**TOGAF**) and other methodologies, including **Continuous Architecture** and the recently published **Open Agile Architecture**. We will study how to define architecture principles for various domains such as security, data, and applications using quality attributes. We will also learn how we can plan and create the architecture in different stages. Lastly, we will discuss the need to validate the architecture and how we can arrange it.

In this chapter, we will cover the following topics:

* Defining architecture principles for multi-cloud
* Using quality attributes in architecture
* Creating the architecture artifacts
* Change management and validation as the cornerstone
* Validating the architecture

**Defining architecture principles for multi-cloud**

We’ll start this chapter from the perspective of enterprise architecture using the **Architecture Development Method** (**ADM**) cycle in TOGAF as a guiding and broadly accepted framework for enterprise architecture. In *Chapter 2*, *Business Acceleration Using a Multi-Cloud Strategy*, we learned that the production cycle for architecture starts with the business, yet there are two steps before we actually get to defining the business architecture: we have a preliminary phase where we set out the framework and the architecture principles. These feed into the very first step in the actual cycle, known as architecture vision, as shown in the following diagram:



*Figure 5.1: The preliminary phase and architecture vision in TOGAF’s ADM cycle*

The key to any preliminary phase is the architecture principles; that is, your guidelines for fulfilling the architecture. There can be many principles, so the first thing that we have to do is create principle groups that align with our business. The most important thing to remember is that principles should enable your business to achieve its goals. Just to be very clear on this aspect: going to the cloud is not a business goal, just like cloud-first is not a business strategy. These are technology statements at best, nothing more. But principles should do more: they have to support the architectural decisions being made and for that reason, they need to be durable and consistent.

When businesses decide that the cloud might be a good platform to host business functions and applications, the most used principles are flexibility, agility, and cost-efficiency. The latter is already highly ambiguous: what does cost-efficient mean? Typically, it means that the business expects that moving workloads to cloud platforms is cheaper than keeping them on-premises. This could be the case, but the opposite can also be true if incorrect decisions in the cloud have been made by following bad architectural decisions based on ambiguous principles. In short, every principle should be challenged:

* Does the principle support the business goals?
* Is the principle clear so that it can’t be subject to multiple interpretations?
* Is the principle leading toward a clearly defined solution?

Some suggested groups for defining principles are as follows:

* Business
* Security and compliance
* Data principles
* Application principles
* Infrastructure and technology principles
* Usability
* Processes

Let’s talk about each category in detail, but before we do so, we must learn by what values we are defining the architecture. These values are captured in quality attributes, to be discussed in the next section.

**Using quality attributes in architecture**

Quality attributes is a term that sprouts from a different framework, called **Continuous Architecture**. Cloud architectures tend to be fluid, meaning that they are very dynamic since they have to respond to changing customer demands fast and continuously. Cloud services are used to enable the business agility that we discussed in the previous chapter. Cloud services, therefore, need to be scalable, but still reliable. Lastly, enterprises do not want to be locked in on any platform, so environments must be able to communicate with other systems or even move to other platforms.

Architectures of systems might change constantly, due to fast developments and continuous new releases enabled by DevOps. Architects have a tedious task to accomplish this. Continuous architecture might be a good guide.

This framework defines quality attributes to which architecture must comply:

* **Operability**: This part of the architecture covers automation in the first place, but also monitoring and logging. In essence, operability is everything that is needed to keep systems operational in a secure state. This requires monitoring:
  + A key decision in monitoring is not what monitoring tool we will use, but what we have to monitor and to what extent. In multi-cloud, monitoring must be cross-platform since we will have to see what’s going on in the full chain of components that we have deployed in our multi-cloud environment. This is often referred to as end-to-end monitoring: looking at systems from an end user perspective. This is not only related to the health status of systems, but also whether the systems do what they should do and are free from bugs, crashes, or unexpected stops.
  + Monitoring systems collect logs. We will also need to design where these logs will have to go and how long these will have to be stored. The latter is important if systems are under an audit regime. Auditors can request logs.
  + Monitoring is also maybe even more related to the performance of these systems. From an end user perspective, there’s nothing more frustrating than systems that respond slowly.
* **Performance**: Where an architect can decide that a system that responds within 1 second is fast, the end user might have a completely different definition of fast. Even if they agree that the performance and responsiveness of the system is slow, the next question is how to determine what the cause of degrading performance is. Monitoring the environment from the end user’s perspective, all the way down to the infrastructure, is often referred to as end-to-end.

There are monitoring environments that really do end-to-end, typically by sending transactions through the whole chain and measuring health (heartbeat, to check if a system is still alive and responding) and performance by determining how fast transactions are processed. Keep in mind that this type of monitoring usually deploys agents on various components in your environment. In that case, we will have to take into consideration how much overhead these agents create on systems, thus taking up extra resources such as CPU and memory. The footprint of agents should be as small as possible, also given the fact that there will undoubtedly be more agents or packages running on our systems. Just think of endpoint protection, such as virus scanning, as an example.

* **Configurability**: Companies use cloud technology to be able to respond quickly to changes, and to become agile. As a result, systems are not static in the cloud. Systems in the cloud must be easy to configure. This is what configurability means: a set of parameters that define how systems must behave. Configurations set the desired state of a resource. As an example, we can use a virtual machine: developers can pick a VM from a portal of a cloud provider. This will be a default VM with default configurations. These configurations might not match the desired state for which the company has defined guidelines and guardrails. In that case, the specific desired configuration must be added separately. Configurations are defined in settings and configuration files. They might include:
  + CPU and memory allocation
  + Storage settings and disk usage
  + Boot sequence
  + Security settings such as hardening the system
  + Access management
  + Operating system configurations
* The configuration tells the resource how it should operate, and how it should “act.” Our example is a VM, but it applies to every resource that we use in the cloud. Developers or administrators need to specify how a resource must be deployed, including all components that are part of the resource. Obviously, this is not something that developers and administrators want to do every time a specific resource is deployed. Configurations will have standards, and if there are standards, they can be automated. The moment a new resource is deployed, a process is automatically triggered to apply the desired settings to that resource. In that case, only the master configuration files have to be managed. Cloud providers offer tools to manage configurations, for instance, with **Desired State Configuration** (**DSC**), which is used in PowerShell.

DSC is declarative scripting that defines and automates the application of settings to Microsoft Windows and Linux systems. AWS offers Systems Manager for the same purpose. GCP has this embedded in Compute Engine.

* **Discoverability**: This is exactly what it says it is. It’s about finding resources in cloud platforms. Resources must be visible for other resources in order to be able to communicate with each other and, obviously, these resources must be visible for monitoring. But there’s more to discoverability. It also includes **service discovery**: the automatic detection of instances and services that run on these instances within a network, for instance, a tenant on a cloud platform:
  + This is extremely important in a microservices architecture: each microservice needs to know where instances are located to discover the services that are hosted on these instances. Service discovery allows the discovery of services dynamically without the need for IP addresses of the instances. These IP addresses will change dynamically, but the services will still be able to “find” each other. The technology typically involves a central services registry.
* **Security**: Here, we are focusing on data protection. After all, data is the most important asset in your multi-cloud environment. The architecture should have one goal: safeguarding the integrity of data. The best way to start thinking of security architecture is to think from the angle of possible threats. How do you protect the data, how do you prevent the application from being breached, and what are the processes in terms of security policies, monitoring, and following up on alerts? The latter is a subject for **Security Operations** (**SecOps**). Security will be extensively discussed in part 4 of this book: *Controlling Security with DevSecOps.*
* **Scalability**: The killer feature of the public cloud is scalability. Whereas previously developers had to wait for the hardware to be ready in the data center before they could actually start their work, we now have the capacity right at our fingertips. But scalability goes beyond that: it’s about full agility and flexibility in the public cloud and being able to scale out, scale up, and scale down whenever business requirements call for that.
* Scaling out is also referred to as horizontal scaling. When we scale out an environment, we usually add systems such as virtual machines to that environment. In scaling up—or vertical scaling—we add resources to a system, typically CPUs, memory, or disks in a storage system. Obviously, when we can scale out and up, we can also go the opposite direction and scale systems down. Since we are paying for what we really use in the public cloud, the costs will come down immediately, which is not the case if we have invested in physical machines sitting in a traditional, on-premises environment.

You will be able to lower the usage of these physical machines, but this will not lower the cost of that machine since it’s fully **CAPEX**—**capital expenditures or investments**. We will discuss financials in part 3 of this book: *Controlling Costs in Multi-Cloud Using FinOps.*

* Typical domains for the scalability architecture are virtual machines (also as hosts for container clusters), databases, and storage, but network and security appliances should also be considered. For example, if the business demand for scaling their environment up or out increases, typically, the throughput also increases. This has an impact on network appliances such as switches and firewalls: they should scale too. However, you should use the native services from the cloud platforms to avoid scaling issues in the first place.
* Some things you should include in the architecture for scalability are as follows:
  + **Definition of scale units**: This concerns scalability patterns. One thing you have to realize is that scaling has an impact. Scaling out virtual machines has an impact on scaling the disks that these machines use, and thus the usage of storage.
  + But there’s one more important aspect that an architect must take into account: can an application handle scaling? Or do we have to rearchitect the application so that the underlying resources can scale out, up, or down without impacting the functionality and, especially, the performance of the application? Is your backup solution aware of scaling?
  + Defining scale units is important. Scale units can be virtual machines, including memory and disks, database instances, storage accounts, and storage units, such as blobs in Azure or buckets in AWS. We must architect how these units scale and what the trigger is to start the scaling activity.
  + **Allowing for autoscaling**: One of the ground principles in the cloud is that we automate as much as we can. If we have done a proper job of defining our scale units, then the next step is to decide whether we allow autoscaling on these units or allow an automated process for dynamically adding or revoking resources to your environment. First, the application architecture must be able to support scaling in the first place. Autoscaling adds an extra dimension. The following aspects are important when it comes to autoscaling:
    - The trigger that executes the autoscaling process.
    - The thresholds of autoscaling, meaning to what level resources may be scaled up/out or down. Also, keep in mind that a business must have a very good insight into the related costs.
  + **Partitioning**: Part of architecting for scalability is partitioning, especially in larger environments. By separating applications and data into partitions, controlling scaling and managing the scale sets becomes easier and prevents large environments from suffering from contention. Contention is an effect that can occur if application components use the same scaling technology but resources are limited due to set thresholds, which is often done to control costs.

Now, we have to make sure that our systems are not just scalable but are also resilient and robust, ensuring high availability:

* **Robustness**: Platforms such as Azure, AWS, and GCP are just there, ready to use. And since these platforms have global coverage, we can rest assured that the platforms will always be available. Well, these platforms absolutely have a high availability score, but they do suffer from outages. This is rare, but it does happen. The one question that a business must ask itself is whether it can live with that risk—or what the costs of mitigating that risk are, and whether the business is willing to invest in that mitigation. That’s really a business decision at the highest level. It’s all about business continuity.
* Robustness is about resilience, accessibility, retention, and recovery. In short, it’s mostly about availability. When we architect for availability, we have to do so at different layers. The most common are the compute, application, and data layers. But it doesn’t make sense to design availability for only one of these layers. If the virtual machines fail, the application and the database won’t be accessible, meaning that they won’t be available.
* In other words, you need to design availability from the top of the stack, from the application down to the infrastructure. If an application needs to have an availability of 99.9 percent, this means that the underlying infrastructure needs to be at a higher availability rate. The underlying infrastructure comprises the whole technology stack: compute, storage, and network.
* A good availability design counters for failures in each of these components, but also ensures the application—the top of the stack—can operate at the availability that has been agreed upon with the business and its end users. However, failures do occur, so we need to be able to recover systems. Recovery has two parameters:
  + **Recovery Point Objective** (**RPO**): RPO is the maximum allowed time that data can be lost. An RPO could be, for instance, 1 hour of data loss. This means that the data that was processed within 1 hour since the start of the failure can’t be restored. However, it’s considered to be acceptable.
  + **Recovery Time Objective** (**RTO**): RTO is the maximum duration of downtime that is accepted by the business.
* RPO and RTO are important when designing the backup, data retention, and recovery solution. If a business requires an RPO of a maximum of 1 hour, this means that we must take backups every hour. A technology that can be used for this is snapshotting or incremental backups. With snapshots, we take an instant copy of the data and save that copy, while incremental backups will take backups of the changes that have occurred since the last backup was made. Taking full backups every hour would create too much load on the system and, above that, implies that a business would need a lot of available backup storage.
* It is crucial that the business determines which environments are critical and need a heavy regime backup solution. Typically, data in such environments also needs to be stored for a longer period of time. Standard offerings in public clouds often have 2 weeks as a standard retention period for storing backup data. This can be extended, but it needs to be configured and you need to be aware that it will raise costs significantly.
* One more point that needs attention is that backing up data only makes sense if you are sure that you can restore it. So, make sure that your backup and restore procedures are tested frequently—even in the public cloud.
* **Portability:**There are multiple definitions of portability. Often, the term is referring to application portability. When the same application has to be installed on multiple platforms, portability can significantly reduce costs. The only requirement for portability is that there is a general abstraction between the application logic and the system interfaces. With cloud portability, we mean that applications can run on different cloud platforms: applications and data can be moved between clouds without (major) disruption. The topic is mostly related to the so-called “vendor” or “cloud” lock-in. Companies want to have the freedom to switch providers and suppliers, including cloud platforms. The term is also frequently mixed with interoperability, indicating that applications are cloud agnostic and can be hosted anywhere. Container technology using, for example, Kubernetes can run on any platform supporting Kubernetes.
* However, in practice, portability and interoperability are much harder than the theory claims. The underlying technology that cloud providers use might differ, causing issues in migrating from one cloud to another, especially with microservices that use specific native technology. One other aspect that plays an important role is data gravity, where the data literally attracts the applications toward the data. This can be a massive amount of data that can’t be easily moved to another platform. Data gravity can heavily influence the cloud strategy of a company.
* **Usability**: This is often related to the ease of use of apps, with clear interfaces and transparent app navigation from a user’s perspective. However, these topics do imply certain constraints on our architecture. Usability requires that the applications that are hosted in our multi-cloud environment are accessible to users. Consequently, we will have to think of how applications can or must be accessed. This is directly related to connectivity and routing: do users need access over the internet or are certain apps only accessible from the office network? Do we then need to design a **Demilitarized Zone (DMZ)** in our cloud network? And where are jump boxes positioned in multi-cloud?
  + Keep in mind that multi-cloud application components can originate from different platforms. Users should not be bothered by that: the underlying technical setup should be completely transparent for users. This also implies architectural decisions: something we call technology transparency. In essence, as architects, we constantly have to work from the business requirements down to the safe use of data and the secured accessibility of applications to the end users. This drives the architecture all the way through.

Continuous architecture is not the only framework that helps in defining agile architecture. While we mentioned TOGAF in this chapter, The Open Group, as an organization that develops standard methodologies for architecture, has issued a different framework that is more suitable for agile architecture than TOGAF, which in itself is rather static. For all good reasons, this new framework is called **Open Agile Architecture** (**O-AA**).

The framework addresses business agility as the main purpose for doing architecture in enterprises, embracing the digital transformation these enterprises are going through. O-AA puts the customer and the customer experience at the heart of the architecture. Since customer demands and experiences change continuously, the architecture must be able to adopt this: it catches the interactions between the customer and all the touchpoints the customer has with the enterprise, for example, through the products of the enterprise, but also the website, social media, customer contact centers and even the **Internet of Things (IoT)** such as mobile devices and sensors that are connected to the internet.

This book will not provide deep dives into TOGAF or O-AA. Please refer to the website of The Open Group for more information and the collateral. O-AA was released in 2020. Van Haren Publishing recently released the methodology in book form, available from TOGAF at <https://publications.opengroup.org/>

Next to the quality attributes, we also must define some principles for our multi-cloud architecture. This is discussed in the next section.

**Defining principles from use cases**

Before we start defining the architecture, we must define the business use case.

A business use case describes the actions a business must take in a specific, defined order to deliver a product or a service to a customer. That product or service must represent a value to that customer since that value can be represented by a price. That price in turn must match the costs that are related to the delivery of the product or service, topped by a margin so that the business makes a profit. So, the business use case describes the workflow to produce and deliver but also defines how the business adds value for its customers while making a profit.

From the use case, the architect derives the principles, starting with the business principles.

**Business principles**

Business principles start with business units setting out their goals and strategy. These adhere to the business mission statement and, from there, describe what they want to achieve in the short and long term. This can involve a wide variety of topics:

* Faster response to customers
* Faster deployment of new products (time to market)
* Improve the quality of services or products
* Engage more with employees
* Real digital goals such as releasing a new website or web shop
* Increase revenue and profit, while controlling costs

The last point, controlling costs, is a topic of FinOps, which will be discussed in chapters 10, *Managing Costs with FinOps*, and 11, *Improving Cost Management with the FinOps Maturity Model*.

As with nearly everything, goals should be **SMART**, which is short for **specific, measurable, attainable, relevant, and timely**. For example, a SMART-formulated goal could be “*the release of the web shop for product X in the North America region on June 1*.” It’s scoped to a specific product in a defined region and targeted at a fixed date. This is measurable as a SMART goal.

Coming back to TOGAF, this is an activity that is performed in phase B of the ADM cycle; that is, the phase in the architecture development method where we complete the business architecture. Again, this book is not about TOGAF, but we do recommend having one language to execute the enterprise architecture in. TOGAF is generally seen as the standard in this case. Business principles drive the business goals and strategic decisions that a business makes. For that reason, these principles are a prerequisite for any subsequent architectural stage.

**Principles for security and compliance**

Though security and compliance are major topics in any architecture, the principles in this domain can be fairly simple. Since these principles are of extreme importance in literally every single aspect of the architecture, it’s listed as the second most important group of principles, right after business principles.

Nowadays, we talk a lot about zero trust and security by design. These can be principles, but what do they mean? **Zero trust**speaks for itself: organizations that comply with zero trust do not trust anything within their networks and platforms. Every device, application, or user is monitored. Platforms are micro-segmented to avoid devices, applications, and users from being anywhere on the platform or inside the networks: they are strictly contained. The pitfall here is to think that zero trust is about technological measures only. It’s not. Zero trust is first and foremost a business principle and looks at security from a different angle: zero trust assumes that an organization has been attacked, with the only question left being what the exact damage was. This is also the angle that frameworks such as MITRE ATT&CK take.

**Security by design** means that every component in the environment is designed to be secure from the architecture of that component: built-in security. This means that platforms and systems, including network devices, are hardened and that programming code is protected against breaches via encryption or hashing. This also means that the architecture itself is already micro-segmented and that security frameworks have been applied. An example of a commonly used framework is the **Center for Internet Security** (**CIS**) framework, which contains 20 critical security controls that cover various sorts of attacks on different layers in the IT stack. As CIS themselves rightfully state, it’s not a one size fits all framework. An organization needs to analyze what controls should be implemented and to what extent.

We’ll pick just one as an example: data protection, which is control 13 in the CIS framework. The control advises that data in transit and data at rest are encrypted. Note that CIS doesn’t say what type of **Hardware Security Modules** (**HSMs**) an organization should use or even what level of encryption.

It says that an organization should use encryption and secure this with safely kept encryption keys. It’s up to the architect to decide on what level and what type of encryption should be used.

In terms of compliance principles, it must be clear what international, national, or even regional laws and industry regulations the business has to adhere to. This includes laws and regulations in terms of privacy, which is directly related to the storage and usage of (personal) data.

An example of a principle is that the architecture must comply with the **General Data Protection Regulation**(**GDPR**). This principle may look simple—comply with GDPR—but it means a lot of work when it comes to securing and protecting environments where data is stored (the systems of record) and how this data is accessed (systems of engagement). Technical measures that will result from this principle will vary from securing databases, encrypting data, and controlling access to that data with authentication and authorization. In multi-cloud, this can be even more challenging than it already was in the traditional data center. By using different clouds and PaaS and SaaS solutions, your data can be placed anywhere in terms of storage and data usage.

**Data principles**

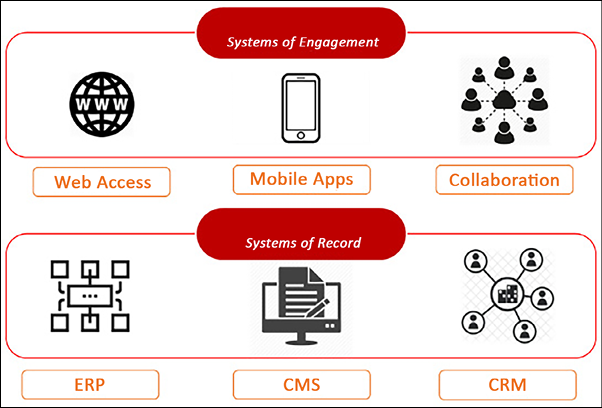
As we mentioned previously, here’s where it really gets exciting and challenging at the same time in multi-cloud environments. The most often used data principles are related to data confidentiality and, from that, protecting data. The five most important data principles are:

* **Accuracy**: Data should be accurate, complete, and reliable. Inaccurate data can lead to flawed insights and decisions and can have serious consequences.
* **Relevance**: Data should be relevant to the problem or question at hand. Unnecessary or irrelevant data can add noise to the analysis and make it harder to extract meaningful insights.
* **Timeliness**: Data should be timely and up to date. Outdated or stale data can lead to incorrect or misleading conclusions.
* **Consistency**: Data should be consistent in terms of format, definitions, and units of measurement. Inconsistent data can create confusion and make it difficult to combine or compare different sources of information.
* **Privacy and security**: Data should be protected from unauthorized access and use. Sensitive data, such as personal or financial information, should be handled with care and stored securely to avoid breaches or leaks.

So, how do you ensure that these principles are adhered to? We briefly touched on two important technology terms that have become quite common in cloud environments earlier in this chapter:

* **Systems of record**: Systems of record are data management or information storage systems; that is, systems that hold data. In the cloud, we have the commonly known database, but due to the scalability of cloud platforms, we can now deploy huge data stores comprising multiple databases that connect thousands of data sources. Public clouds are very suitable to host so-called data lakes.
* **Systems of engagement**: Systems of engagement are systems that are used to collect or access data. This can include a variety of systems: think of email, collaboration platforms, and content management systems, but also mobile apps or even IoT devices that collect data, send it to a central data platform, and retrieve data from that platform.

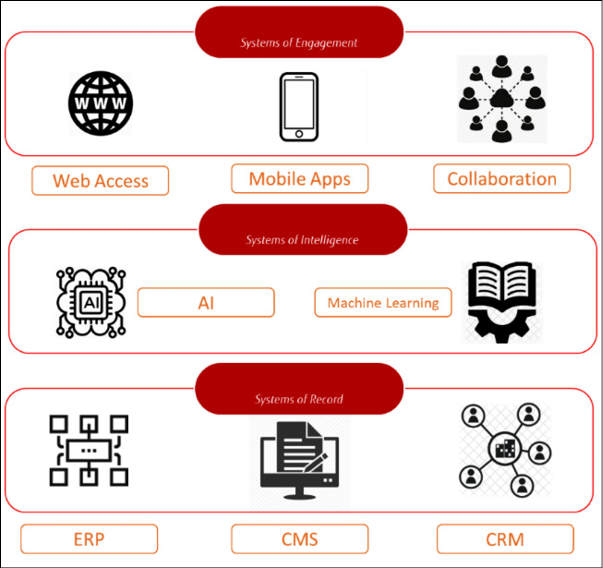
A high-level overview of the topology for holding systems of record and systems of engagement is shown in the following diagram, with **Enterprise Resource Planning** (**ERP**), **Content Management** (**CMS**), and **Customer Relationship Management** (**CRM**) systems being used as examples of systems of record:



*Figure 5.2: Simple representation of systems of engagement and systems of record*

The ecosystem of record and engagement is enormous and growing. We’ve already mentioned data lakes, which are large data stores that mostly hold raw data. In order to work with that data, a data scientist would need to define precise datasets to perform analytics. Azure, AWS, and Google all have offerings to enable this, such as Data Factory and Databricks in Azure, EMR and Athena in AWS, and BigQuery from Google.

Big data and data analytics have become increasingly important for businesses in their journey to become data-driven: any activity or business decision, for that matter, is driven by actual data. Since clouds can hold petabytes of data and systems need to be able to analyze this data fast to trigger these actions, a growing number of architects believe that there will be a new layer in the model. That layer will hold “systems of intelligence” using machine learning and **artificial intelligence (AI)**. Azure, AWS, and Google all offer AI-driven solutions, such as Azure ML in Azure, SageMaker in AWS, and Cloud AI in Google. The extra layer—the systems of intelligence—can be seen in the following diagram:



*Figure 5.3: Simple representation of the systems of intelligence layer*

To be clear: systems of record or engagement don’t say anything about the type of underlying resource. It can be anything from a physical server to a **virtual machine (VM)**, a container, or even a function. Systems of record or engagement only say something about the functionality of a specific resource.

**Application principles**

Data doesn’t stand on its own. If we look at TOGAF once more, we’ll see that data and applications are grouped into one architectural phase, known as phase C, which is the information systems architecture. In modern applications, one of the main principles of applications is that they have a data-driven approach, following the recommendation of Steven Spewak’s enterprise architecture planning. Spewak published his book *Enterprise Architecture Planning*in 1992, but his approach is still very relevant, even—and perhaps even more—in multi-cloud environments.

Also mentioned in Spewak’s work: the business mission is the most important driver in any architecture. That mission is data-driven; enterprises make decisions based on data, and for that reason, data needs to be relevant, but also accessed and usable. These latter principles are related to the applications disclosing the data to the business. In other words, applications need to safeguard the quality of the data, make data accessible, and ensure that data can be used. Of course, there can be—and there is—a lot of debate regarding, for instance, the accessibility of data. The sole relevant principle for an application architecture is that it makes data accessible. To whom and on what conditions are both security principles.

In multi-cloud, the storage data changes, but also the format of applications. Modern applications are usually not monolithic or client-server-based these days, although enterprises can still have a large base of applications with legacy architectures. Cloud-native apps are defined with roles and functions and build on the principles of code-based modularity and the use of microservices. These apps communicate with other apps using APIs or even triggers that call specific functions in other apps. These apps don’t even have to run on the same platform; they can be hosted anywhere. Some architects tend to think that monolithic applications on mainframes are complex, so use that as a guideline to figure out how complex apps in multi-cloud can get.

However, a lot of architectural principles for applications are as valid as ever. The technology might change, but the functionality of an application is still to support businesses when it comes to rendering data, making it accessible, and ensuring that the data is usable.

Today, popular principles for applications are taking the specific characteristics of cloud-native technology into consideration. Modern apps should be enabled for mobility, be platform-independent using open standards, support interoperability, and be scalable. Apps should enable users to work with them at any time, anywhere.

One crucial topic is the fact that the requirements for applications change at almost the speed of light: users demand more and more from apps, so they have to be designed in an extremely agile way so that they can adopt changes incredibly fast in development pipelines. Cloud technology does support this: code can easily be adapted. But this does require that the applications are well-designed and documented, including in runbooks.

**Infrastructure and technology principles**

Finally, we get to the real technology: machines, wires, nuts, and bolts. Here, we’re talking about virtual nuts and bolts. Since data is stored in many places in our multi-cloud environment and applications are built to be cloud-native, the underlying infrastructure needs to support this. This is phase D in TOGAF, the phase in architecture development where we create the target technology architecture, which comprises the platform’s location, the network topology, the infrastructure components that we will be using for specific applications and data stores, and the system interdependencies. In multi-cloud, this starts with drafting the landing zone: the platform where our applications and data will land.

One of the pitfalls of this is that architects create long, extensive lists with principles that infrastructure and technology should adhere to, all the way up to defining the products that will be used as a technology standard. However, a catalog with products is part of a portfolio. Principles should be generic and guiding, not constraining. In other words, a list of technology standards and products is not a principle. A general principle could be about bleeding edge technology: a new, non-proven, experimental technology that imposes a risk when deployed in an environment because it’s still unstable and unreliable.

Other important principles for infrastructure can be that it should be scalable (scale out, up, and down) and that it must allow micro-segmentation. We’ve already talked about the Twelve-Factor App, which sets out specific requirements for the infrastructure. These can be used as principles. The principles for the Twelve-Factor App were set out in 2005, but as we already concluded in *Chapter 2*, *Collecting Business Requirements*, they are still very accurate and relevant.

The Twelve-Factor App sets the following three major requirements for infrastructure:

* The app is portable between different platforms, meaning that the app is platform-agnostic and does not rely on a specific server or system’s settings.
* There’s little to no difference between the development stage and the production stage of the app so continuous development and deployment are enabled. The platform that the app is deployed on should support this (meaning that everything is basically code-based).
* The app supports scaling up without significant changes needing to be made to the architecture of the app.

In the next section, we will discuss the principles for usability and processes. We will also touch upon the transition and transformation to cloud environments.

**Principles for processes**

The last group of principles is concerned with processes. This is not about the **IT System Management** (**ITSM**) processes, but about the processes of deployment and automation in multi-cloud. One of the principles in multi-cloud is that we will automate as much as we can. This means that we will have to define all the tasks that we would typically do manually in an automated workflow. If we have a code-only principle defined, then we can subsequently set a principle that states that we must work from the code base or master branch. If we fork the code and we do have to make changes to it, then a principle is that altered code can only be committed back to the master code if it’s tested in an environment that is completely separated from acceptance and production. This is related to the life cycle process of our environment.

So, processes here focus more on our way of working. Today, a lot of companies are devoted to agile and DevOps. If that’s the defined way of working, then it should be listed as a principle; for example, an organization embraces agility in its core processes. This can be done through the **Scaled Agile Framework** (**SAFe**) or the Spotify model. Following that principle, a company should also define the teams, their work packages, and how epics, features, product backlogs, and so on are planned. However, that’s not part of the principle anymore. That’s a consequence of the principle.

As with all principles, the biggest pitfall is making principles too complex. Especially with processes, it’s important to really stick to describing the principle and not the actual process.

We have discussed the architecture principles and why we are doing architecture. The next step is to define the components of the architecture. This is the topic of the next section.

**Creating the architecture artifacts**

The hierarchy in documents that cover the architecture starts with the enterprise architecture. It’s the first so-called architecture artifact. The enterprise architecture is followed by the high-level design and the low-level design, which covers the various components in the IT landscape. We will explore this in more detail in the following sections. Keep in mind that these sections are merely an introduction to the creation of these artifacts. You will find samples of these artifacts at <https://publications.opengroup.org/i093>, where you can download a ZIP file containing relevant templates.

**Creating a business vision**

Creating a business vision can take years, but it’s still a crucial artifact in architecture. It sets out what the business wants to achieve. This should be a long-term outlook since it will drive architectural decisions. Though cloud environments enable the agile deployment of services, deployment should never become ad hoc.

A business vision focuses on the long-term goals in terms of finance, quality of services/products, sustainability of the business, and, above all, the potential growth of the business and market domains that it’s targeting. The business vision is the input for the enterprise architecture. It’s the only document that will not be produced by architects, although the enterprise architect might be an important stakeholder that gives their view on the vision. After all, the vision must be realistic and obtainable. In other words, the architecture must be able to support the vision and help achieve its goals.

**Enterprise architecture**

The enterprise architecture is the first document that will be written by architects. Typically, this is the deliverable that is created by a team of architects, led by the enterprise or business architect. They will work together with domain architects. The latter can also be a cloud architect or an architect specialized in cloud-native development. The enterprise architecture describes the business structures and processes and connects these to the use of data in the enterprise. This data drives the business in the enterprise. In essence, enterprise architecture bridges business and IT.

**Principles catalog**

This document lists all the architecture principles that have to be applied to any architecture that will be developed. We discussed this in detail in the first section of this chapter, *Defining architecture principles for multi-cloud*. Principles are assembled per architecture domain.

**Requirements catalog**

This document lists all the requirements that a business has issued in order to achieve its goals since these are set out in the business vision. Going from a business vision to a requirements catalog is a long haul, so there are intermediate steps in creating the enterprise architecture and the principles catalog. From there, business functionality must be translated into requirements regarding the use of data and application functionality. Since not everything is known in detail at this stage, the catalog also contains assumptions and constraints. At the end, the catalog holds a list of building blocks that represent solutions to the business requirements.

**High-level design**

This is not an official TOGAF document. TOGAF talks about a solution concepts diagram. In practice, a lot of people find it hard to read just a diagram and grasp the meaning of it. A high-level design provides the solution concepts and includes the rationales of why specific concepts have been chosen to fulfill the requirements. Typically, a high-level design is created per architecture domain: data, application, and technology. Cloud concepts are part of each of these domains. Networking, computing, and storage are concepts that fit into the technology design. Data logics and data streams are part of the data design. Application functions must be described in the design for applications.

**Low-level design**

This document contains the nitty-gritty details per building block. Low-level designs for data comprise data security and data transfer. Application designs contain the necessary software engineering diagrams and distribution patterns. Technology designs hold details on the core and boundaries (networks and security), including lists of used ports; IP plan and communication protocols; platform patterns and segmentation processing units (VMs, containers, and so on); storage division; interfaces; and so on.

One note that has to be made here is that in *Chapter 4*, *Service Design for Multi-Cloud*, we agreed that we would work with everything as code. So, does it make sense to have everything written out in documents that are stored in some cabinet or drawer, never to be looked at again? Still, documenting your architecture is extremely important, but we can also have our documentation in wikis that can easily be searched through and directly linked to the related code that is ready to be worked with or even deployed. In today’s world of multi-cloud, DevOps, and CI/CD pipelines, this will be the preferred way of working.

Working in DevOps pipelines and having documentation in wikis enforces the fact that the cycle of creating and maintaining these artifacts never stops. Code and wikis can easily be maintained and are more agile than chunky documents. Keep in mind that artifacts will constantly be updated. This is the ground principle of continuous architecture (reference: *Continuous Architecture*, by Murat Erder and Pierre Pureur, 2016). Continuous architecture doesn’t focus on solutions, for a good reason.

In multi-cloud, there are a zillion solutions and solution components (think of PaaS) already available and a zillion more on their way. Continuous architecture focuses on the quality of the architecture itself and describes how to design, build, test, and deploy solutions, as in DevOps pipelines. Other than this, it has a strong focus on continuous validation of the architecture, which is something we will explore in the last section of this chapter—that is, *Validating the architecture*.

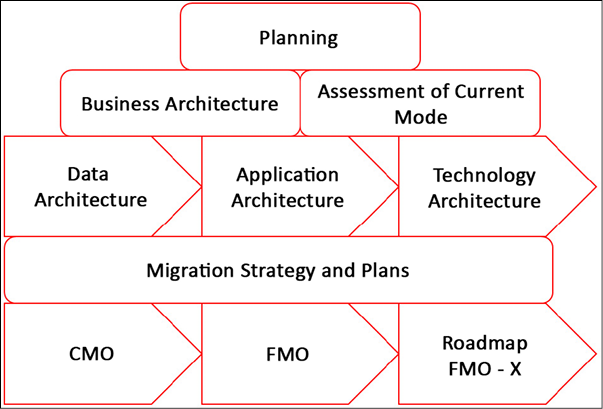
**Planning transition and transformation**

We have done a lot of work already. Eventually, this should all add up to an architecture vision: a high-level view of the end state of our architecture and the objective of that architecture. However, an architecture is more than just a description or a blueprint of that end state. An architecture should also provide a roadmap—that is, a guide on how we will reach that end state. To be honest, there’s nothing new under the sun here. On the contrary, this is how IT programs are typically run: it’s all about transition and transformation.

Let’s assume that our end state is full cloud adoption. This means that the business has all of its IT systems, data, and applications in the cloud. Everything is code-based and automated, deployed, and managed from CI/CD pipelines. We’ve adopted native cloud technology such as containers and serverless functions. In *Chapter 2*, *Business Acceleration Using a Multi-Cloud Strategy*, we defined this as the dynamic phase, but that’s a technical phase. The dynamic phase can be part of the end state of our architecture. However, we need to be absolutely sure that this dynamic technology does fit the business needs and that we are ready to operate this environment in the end state. We will refer to this end state as the **Future Mode of Operation** (**FMO**).

How do we get to this FMO? By starting at the beginning: the current situation, the **Current Mode of Operation** (**CMO**), or the **Present Mode of Operation** (**PMO**). A proper assessment of the existing landscape is crucial to get a clear, indisputable insight into the infrastructure, connections, data, and applications that a business has in its IT environment. From there, we can start designing the transition and transformation to the FMO.

If we combine the methodology of Spewak—the model that we discussed under the section about *Application principles*—with CMO-FMO planning, the model will look as follows:



*Figure 5.4: Spewak’s Enterprise Architecture model plotted with transition planning*

If we don’t change anything in our application and we simply move it to a public cloud using IaaS or bare-metal propositions, then we can talk about transition. The technology phase would be the standard phase. Transition just means that we move the workloads, but we don’t change anything at all in terms of the technology or services. However, we are also not using cloud technologies to make our environment more agile, flexible, or cost-efficient. If we want to achieve that, we will need to make a transformation: we need to change the technology under the data and applications. This is a job that needs to be taken care of through architecture. Why are we changing? What are we changing? How are we changing applications? And also, how do we revert changes if things don’t work as planned; that is, what is our fallback solution?

There’s one debate that needs discussing in terms of transition and transformation. As explained already, transition means that we are not changing the underlying technology and services. We just move an environment from A to B, as it is. But is that true when we are shifting environments to a public cloud? Moving an application to Azure, AWS, or GCP always implies that we are changing something: either the underlying platform or the services.

By moving an application to a major public cloud, the services will almost definitely change. We are introducing a third party to our landscape: a public cloud provider. Hence, we are introducing an agreement to our landscape. That agreement comprises terms and conditions on how our applications will be hosted in that public cloud. This is something the architecture should deal with in a clear change management process. We will cover change management in the next section.

**Change management and validation as the cornerstone**

We are working under architecture from this point onward. This implies that the changes that are made to the systems in our environment are controlled from the architecture. Sometimes, these changes have an impact on the architecture itself, where we will need to change the architecture. In multi-cloud environments, that will actually happen a lot.

Cloud platforms are flexible in terms of use and thus our architecture can’t be set in stone: it needs to allow improvements to be made to the environments that we have designed, thereby enabling these improvements to be documented and embedded in the architecture. Improvements can be a result of fixing a problem or mitigating an issue with enhancements. Either way, we have to make sure that changes that are the result of these improvements can be validated, tracked, and traced. Change management is therefore crucial in maintaining the architecture.

Since we have already learned quite a bit from TOGAF, we will also explore change management from this angle: phase H. Phase H is all about change management: keeping track of changes and controlling the impact of changes on the architecture. But before we dive into the stages of proper change management, we have to identify what type of changes we have in IT. Luckily, that’s relatively easy to explain since IT organizations typically recognize two types: standard and non-standard changes. Again, catalogs are of great importance here.

Standard changes can be derived from a catalog. This catalog should list changes that have been foreseen from the architecture as part of standard operations, release, or life cycle management. A standard change can be to add a VM. Typically, these are quite simple tasks that have either been fully automated from a repository and the code pipeline or have been scripted. Non-standard changes are often much more complex. They have not been defined in a catalog or repository, or they consist of multiple subsequent actions that require these actions to be planned.

In all cases, both with standard and non-standard changes, a request for change is the trigger for executing change management. Such a request has a trigger: a drive for change. In change management for architecture, the driver for change always has a business context: what problem do we have to solve in the business? The time to market for releasing new business services is too slow, for instance. This business problem can relate to not being able to deploy systems fast enough, so we would need to increase deployment speed. The solution could lie in automation—or designing systems that are less complex.

That is the next step: defining our architecture objectives. This starts with the business objective (getting services to market faster) and defining the business requirements (we need faster deployment of systems), which leads to a solution concept (automatic deployment). Before we go to the drawing board, there are two more things that we must explore.

Here, we need to determine what the exact impact of the change will be and who will be impacted: we need to assess who the stakeholders are, everyone who needs to be involved in the change, and the interests of these people. Each stakeholder can raise concerns about the envisioned change. These concerns have to be added to the constraints of the change. Constraints can be budgetary limits but also timing limits: think of certain periods where a business can’t absorb changes.

In summary, change management to architecture comprises the following:

1. Request for change
2. The request is analyzed through change drivers within the business context
3. Definition of business objectives to be achieved by the change
4. Definition of architecture objectives
5. Identifying stakeholders and gathering their concerns
6. Assessment of concerns and constraints of the change

These steps have to be documented well so that every change to the architecture can be validated and audited. Changes should be retrievable at all times. Individual changes in the environment are usually tracked via a service, if configured. However, a change can comprise multiple changes within the cloud platform. We will need more sophisticated monitoring to do a full audit trail on these changes, to determine who did what. But having said that, it remains of great importance to document changes with as much detail as possible.

**Validating the architecture**

You might recognize this from the process where we validate the architecture of software. It is very common to have an architecture validation in software development, but any architecture should be validated. But what do we mean by that and what would be the objective? The first and most important objective is quality control. The second objective is that improvements that need to be made to the architecture need to be considered. This is done to guarantee that we have an architecture that meets our business goals, addresses all the principles and requirements, and can be received for continuous improvement.

Validating the architecture is not an audit. Therefore, it is perfectly fine to have the first validation procedure be done through a peer review: architects and engineers that haven’t been involved in creating the architecture. It is also recommended to have an external review of your cloud architecture. This can be done by cloud solutions architects from different providers, such as Microsoft, AWS, and Google. They will validate your architecture against the reference architectures and best practices of their platforms, such as the Well-Architected Framework. These companies have professional and consultancy services that can help you assess whether best practices have been applied or help you find smarter solutions for your architecture. Of course, an enterprise would need a support agreement with the respective cloud provider, but this isn’t a bad idea.

The following is what should be validated at a minimum:

* **Security**: Involve security experts and the security officer to perform the validation process.
* **Discoverability:**Architects and developers must be sure that services can find each other, especially when the architecture is defined by microservices, for instance, using container and serverless technology.
* **Scalability**: At the end of the day, this is what multi-cloud is all about. Cloud environments provide organizations with great possibilities in terms of scaling. But as we have seen in this chapter, we have to define scale sets, determine whether applications are allowing for scaling, and define triggers and thresholds, preferably all automated through auto-scaling.
* **Availability and robustness**: Finally, we have to validate whether the availability of systems is guaranteed, whether the backup processes and schemes are meeting the requirements, and whether systems can be restored within the given parameters of RTO and RPO.

In summary, validating our architecture is an important step to make sure that we have completed the right steps and that we have followed the best practices.

**Summary**

In the cloud, it’s very easy to get started straight away, but that’s not a sustainable way of working for enterprises. In this chapter, we’ve learned that, in multi-cloud, we have to work according to a well-thought-out and designed architecture. This starts with creating an architecture vision and setting principles for the different domains such as data, applications, and the underlying infrastructure. Quality attributes are a great help in setting up the architecture.

With these quality attributes, we have explored topics that make architecture for cloud environments very specific in terms of availability, scalability, discoverability, configurability, and operability. If we have designed the architecture, we have to manage it. If we work under the architecture, we need to be strict in terms of change management. Finally, it’s good practice to have our architectural work validated by peers and experts from different providers.

With this, we have learned how to define enterprise architecture in different cloud platforms by looking at the different stages of creating the architecture. We have also learned that we should define principles in various domains that determine what our architecture should look like. Now, we should have a good understanding that everything in our architecture is driven by the business and that it’s wise to have our architecture validated.

Now, we are ready for the next phase. In the next chapter, we will design the landing zones using the Well-Architected Framework and introduce BaseOps: the basic operations of multi-cloud.

**Questions**

1. Name at least four quality attributes that are discussed in this chapter.
2. What would be the first artifact in creating the architecture?
3. What are the two types of changes?

**Further reading**

* The official page of The Open Group Architecture Framework: <https://www.opengroup.org/togaf>.
* *Enterprise Architecture Planning*, by Steven Spewak, John Wiley & Sons Inc