**20**

**Conclusion: The Future of Multi-Cloud**

This book has dealt with designing, implementing, and controlling a multi-cloud platform. We talked about five major clouds—Azure, AWS, GCP, Oracle Cloud, and Alibaba Cloud—and discussed strategies to get the best out of these clouds for our businesses. We discovered that building and managing in the cloud can be complex. Yet, the cloud will definitively grow. We will look at the future of the cloud in this final chapter.

The cloud will grow and multi-cloud will grow. The biggest challenge is how organizations can stay in control of their applications in a multi-cloud setting since the cloud can become very complex. Maybe Google has the answer: **Site Reliability Engineering** (**SRE**). SRE incorporates aspects of software engineering and applies them to infrastructure and operations problems. We will also use this chapter to introduce the concept of SRE and its main principles.

In this chapter, we’re going to cover the following main topics:

* The growth and adoption of multi-cloud
* Understanding the concept of SRE
* Working with risk analysis in SRE
* Applying monitoring principles in SRE
* Applying principles of SRE to multi-cloud—building and operating distributed systems

**The growth and adoption of multi-cloud**

In recent years, multi-cloud has emerged as a popular approach for businesses to manage their cloud infrastructure. Let’s recap the definition of multi-cloud one more time: we speak about multi-cloud when we use two or more cloud service providers to host and run applications and services. As we look toward the near future, we can expect to see continued developments in multi-cloud as businesses seek to take advantage of its benefits while managing its risks. We’ll talk about managing risks later in this chapter when we explore the concept of SRE.

One of the primary reasons that businesses are looking more into multi-cloud is the need for flexibility and agility. Multi-cloud allows businesses to avoid vendor lock-in and take advantage of the unique features and capabilities offered by different cloud providers. This allows them to optimize their applications and services for specific use cases, such as high-performance computing or machine learning, and to quickly respond to changing market conditions or customer needs.

Another benefit of multi-cloud is improved reliability and resilience. By spreading their applications and services across multiple cloud providers, businesses can reduce the risk of downtime or service interruptions due to outages or other issues. This also allows them to implement disaster recovery and business continuity strategies that can help them recover quickly in the event of a major outage or data loss.

However, multi-cloud is not without its risks and pitfalls. One of the biggest challenges with multi-cloud is the complexity of managing multiple cloud environments, as we have seen throughout this book. This can lead to issues with security, compliance, and data governance, as well as increased operational costs and management overhead. Additionally, there may be challenges with data migration and integration across multiple cloud providers, as well as the risk of vendor lock-in with specific tools or services. We discussed these topics when applying BaseOps, FinOps, and DevSecOps.

To mitigate risks, businesses need to carefully plan and implement their multi-cloud strategy. This includes selecting the right cloud providers and services to meet their specific needs, developing a strong governance framework to manage security and compliance, and implementing robust monitoring and management tools to ensure visibility and control across all cloud environments.

Looking toward the future, we can expect to see continued developments in multi-cloud, including the emergence of new technologies and tools to help businesses manage their cloud environments more effectively. We may also see increased collaboration and standardization across cloud providers to make it easier for businesses to integrate and manage their multi-cloud environments.

Multi-cloud is a rapidly evolving area of cloud computing that offers significant benefits for businesses in terms of flexibility, agility, and resilience. Businesses will have to adopt strategies and methodologies to mitigate risks and avoid pitfalls associated with managing multiple cloud environments. A way to overcome the complexity of multi-cloud is by adopting SRE, which we will discuss in the following sections.

**Understanding the concept of SRE**

Originally, SRE was meant for mission-critical systems, but overall, it can be used to drive the DevOps process in a more efficient way. The goal is to enable developers to deploy infrastructure quickly and without errors. To achieve this, the deployment is fully automated. In this way of working, operators will not be swamped with requests to constantly onboard and manage more systems.

The original description of SRE as invented by Google is well over 400 pages long. In the *Further reading* section, a good book is listed to give you a real deep dive into SRE. This chapter is merely an introduction.

Key terms in SRE are **service-level indicators** (**SLIs**), SLO, and the error budget, or the number of failures that lead to the unavailability of a system. The terms are explained in more detail in the next paragraphs.

SLI and SLO differ from **SLA**, the **service-level agreement**. The SLA is an agreement between the supplier of a service and the end user of that service. SLAs comprise **key performance indicators** (**KPIs**), typically, indicators about the uptime of systems. For example, an SLA may contain a metric concerning an uptime—or **mean time to failure** (**MTTF**)—of 99.9% for a system. It means that the system may be unavailable to the end user for 44 minutes per month, often called downtime. Even the most reliable systems will suffer from failure every now and then, and then KPIs such as **mean time to repair or recovery** (**MTTR**) become important: the average time needed to fix an issue in systems.

Fixing problems can become a large part of the work that operation teams do: in SRE, this is referred to as toil. Explained simply, toil is manual reactive tasks that keep teams away from other proactive tasks and eventually slow down development. SRE is built on the principle that SRE teams have to spend up to 50% of their time improving systems, which means that toil has to be reduced as much as possible so that teams can spend as much time as possible developing.

To enable this, SRE teams set targets, defined in SLIs and SLOs:

* **SLO**: In SRE, this is defined as *how good a system should be*. The SLO is more specific than an SLA, which comprises a lot of different KPIs. One could also state that the SLA comprises a number of SLOs. However, an SLO is an agreement between the developers in the SRE team and the product owner of the service, whereas an SLA is an agreement between the service supplier and the end user.
* The SLO is a target value. For example, the web frontend should be able to handle 100 requests per minute. Don’t make it too complex in the beginning. By setting this SLO, the team already has a number of challenges to be able to meet this target, since it will not only involve the frontend but also the throughput on, for instance, the network and involved databases. In other words, by setting this one target, architects and developers will have a lot of work to do to reach that target.
* **SLI**: SLOs are measured by SLIs. In SRE, there are a few indicators that are really important: request latency, system throughput, availability, and error rate. These are the key SLIs. Request latency measures the time before a system returns a response. System throughput is the number of requests per second or minute. Availability is the amount of time a system is usable to the end user. The error rate is the percentage of the total number of requests and the number of requests that are successfully returned.
* **Error budget**: This is probably the most important term in SRE. The SLO also defines the error budget. The budget starts at 100 and is calculated by deducting the SLO. For example, if we have an SLO that says that the availability of a system is 99.9%, then the error budget is 100 - 99.9 = -0.1. This is where the SRE teams have to apply changes without impacting the SLO. It forces developers in the SRE team to either limit the number of changes and releases or to test and automate as much as possible to avoid the disruption of the system and overspending the error budget.
* Remember that SRE is about reducing toil in operations. SRE teams are DevOps teams and they have to make sure that they can spend more time in Dev than in Ops. That starts with the architecture of systems: are these systems fault-tolerant, meaning that systems will still continue to run even if one or more components fail? There might be a reduction in throughput or an increase in latency, but systems should still be available and usable.

To detect failures or performance degradation, monitoring is extremely important in SRE. But before teams get to monitoring, building, and operating with SRE, architects need to define the SLO and SLI. This is done through risk analysis, to be discussed in the next section.

**Working with risk analysis in SRE**

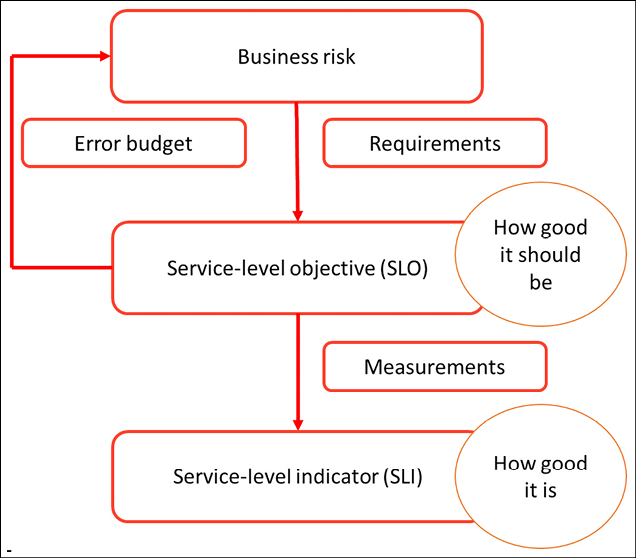
The basis of SRE is that reliability is something that you can design as part of the architecture of applications and systems. Next to that, reliability is also something that one can measure. According to SRE, reliability is a measurable quality, and that quality can be influenced by design decisions. Engineers can take measures to decrease the detection, response, and repair time, and they can develop systems in such a way that changes can be executed safely without causing any downtime. Architects can design fault-tolerant systems; engineers can develop them.

The major issue is it all comes at a cost, and whether systems really need to be fault-tolerant is a business decision, based on a business case. Already, in *Chapter 1*, *Introduction to Multi-Cloud*, we’ve learned that business cases are driven by risks. Let’s go over risk management one more time.

The basic rule is that *risk = probability x impact*. Enterprises use risk management to determine the business value of implementing measures that limit either the probability and/or the impact – or, to put it in SRE terminology: risk management is used to determine the value of reliability engineering. Risk management is also used to prioritize reliability measures in the product backlogs of SRE teams. That is done by following the risk matrix, sometimes referred to as PRACT:

* **Prevent**: The risk is avoided completely.
* **Reduce**: The impact or likeliness of the risk occurring is reduced.
* **Accept**: The consequences of the risk are accepted.
* **Contingency**: The measures are planned and executed when the risk occurs.
* **Transfer**: The consequences of the risk are transferred, for instance, to an insurance company.

If the impact of the failure is great, it might be worthwhile looking at a strategy that prevents the risk. This will drive the SLO—how good a system should be. In this case, the availability might be set to 99.99%, leaving only 0.01% for the error budget. This has consequences for the architecture of the system; after all, the risk rating only allows 52 minutes of downtime per year. The diagram shows how business risks drive SLOs in SRE:



*Figure 20.1: The concepts of SLO and SLI*

The error budget is used to control the risks and make decisions that don’t compromise the SLOs. To calculate the impact on the SLO, the following items have to be taken into consideration:

* **Time to detect**(**TTD**): The time taken to detect an issue in software or a system.
* **Time to resolve** (**TTR**): The time taken to resolve or repair the issue.
* **Frequency/year**: The frequency of errors per year.
* **Users**: The number of users that are impacted by the error.
* **Bad/year**: The number of minutes per year that a system is not usable, or the “bad minutes” per year.

An example will make things clearer. A team deploys bad code to an application. It takes 15 minutes to detect that the application is not performing well with this code and another 15 minutes to resolve the issue by executing a rollback. It’s estimated that this will happen at least once every 2 weeks, so around 26 times per year. It will impact 25% of the user population. This will lead to a number of bad minutes per year. If this is higher than the error budget, then the team needs to take measures to reduce the risk. If the SRE team doesn’t do that, it will lead to a lot of work in operations. Engineers will have to spend more time fixing problems.

**Applying monitoring principles in SRE**

Reliability is a measurable quality. To be able to measure the quality of the systems and their reliability, teams need real-time information on the status of these systems. As mentioned in the previous section, the TTD is a crucial driver in calculating risk and, subsequently, determining the SLO. Observability is therefore critical in SRE. However, SRE stands with the principle that monitoring needs to be as simple as possible. It uses the four golden signals:

* **Latency**: The time that a system needs to return a response.
* **Traffic**: The amount of traffic that is placed on the system.
* **Errors**: The number of requests placed on a system that fail completely or partially.
* **Saturation**: The utilization of the maximum load that a system can handle.

Based on these signals, monitoring rules are defined. As the starting point in SRE is avoiding too much work for operations or toil, the monitoring rules follow the same philosophy. Monitoring should not lead to a tsunami of alerts. The basic rules are as follows:

* The rule must detect a condition that is urgent, actionable, and visible to the user. The condition would not be detected without the rule.
* Can the team ignore the alert that is triggered by the rule? What would happen if the alert is ignored?
* If the alert can’t be ignored, then how can teams action the alert? For example, if a majority of users are affected by the condition, then the alert can’t be ignored, and action must be taken.
* Are there short-term workarounds to improve the condition? This doesn’t mean that SRE promotes short-term workarounds, but it does promote actions that ensure that systems are available and usable, even when an error occurs. Remember that SRE is about making systems reliable. And also remember, there’s nothing as permanent as a temporary workaround (the origin of this lies in this poem: <https://www.poetryfoundation.org/poetrymagazine/poems/55235/after-a-greek-proverb>). Architects and engineers really should avoid accepting workarounds as solutions.
* Are teams allowed to take action after the alert has been raised?
* Can actions on alerts be automated in a safe manner?

All monitoring rules in SRE must adhere to these principles. In short, monitoring in SRE is about making a good distinction between signals that require action and noise. Monitoring should only do two things in theory: define what is broken in a system and, next, determine why it is broken, getting to the root cause. Most monitoring systems focus on what is broken, the symptoms. It requires more sophisticated monitoring to correlate data and get to the cause of an error.

Especially in multi-cloud environments, the error of a system can find its cause in a system that is hosted on a different platform. Since this is already complex in itself, monitoring rules should be designed in such a way that teams are only alerted when the thresholds of the four golden signals are compromised, making systems unavailable to users.

**Applying principles of SRE to multi-cloud—building and operating distributed systems**

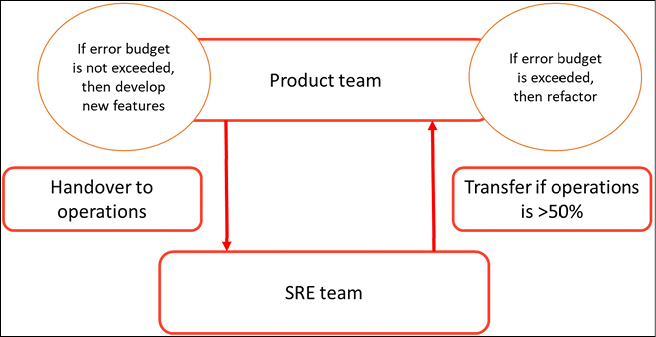
This book exists because a majority of enterprises are moving or developing systems in cloud environments. Today’s enterprises are in a constant transformation mode. This also means a big change in operations. To put it simply, they have to keep up with the speed of change. Traditional operations can’t handle this. We need SRE in the future of multi-cloud. SRE teams create reliable systems in cloud environments.

There are a couple of important rules for SRE to enable this:

* **Automate everything**: Automation leads to consistency, but automation also enables scaling. This requires a very well-thought-out architecture. Automation enables issues to be fixed faster since it only has to be fixed in one place: the code. Automation makes sure that the proper code is distributed over all systems involved. With large distributed systems spanning various cloud platforms, this would take days to do manually. SRE was invented by Google, which already had massive services running from cloud services—services that consumers were relying on, such as Gmail. Without automation, these services never would have been as stable as they are today. Without automation, operations would simply be drowned by manual tasks.
* **Eliminate toil**: This is a specific term used in SRE and might be a bit difficult to understand. It’s not just work that teams would rather not do; it’s every piece of work that keeps teams away from developing.

Toil is manual work, repetitive, and can be automated. But toil is also work that doesn’t add value to the product: it’s interruptive and slows down the development of services that add value. SRE has a rule for toil: an SRE team should not spend more than 50% of their work on toil. The rationale behind that is that toil can easily consume up to 100% of a team’s time.

* How does SRE deal with that? This is where the error budget is important again. If the SRE team needs to spend more than half its time on operations and toil, the error budget is likely exceeded. This calls for engineering, typically meaning that systems need to be refactored by the product team that is responsible for the system. Refactoring aims to improve the design and often also to reduce complexity. The following diagram shows the concept of eliminating toil in SRE.



*Figure 20.2: Concept of eliminating toil*

* **Keep it simple**: Simplicity is a key principle in SRE. Software needs to be simple as a prerequisite to be a stable and reliable system. As a consequence, SRE teams have a strong mandate to push back against product teams when systems are getting too complex. This is often caused by the fact that code for new features is added, but old code is not removed. Code needs to be simple and clean, the use of APIs should be limited, and, if used, APIs should be as simple as possible. SRE lives by the golden rule of less is more.
* **Release engineering**: To keep systems stable and reliable, while changes to developments are applied constantly and at high speed, companies need a rock-solid release process.

Google added release engineers to their teams, specialists that are experts in source code management, software compiling, packaging, configuration builds, and automation. In *Chapter 18*, *Designing and Implementing CI/CD Pipelines*, the principle of branches was discussed. SRE works with the principle of checking in code, not directly to a master branch but through feature branches.

* Testing is, as in any pipeline, a crucial gate in the release process. Here, another term is introduced that is more or less specific to SRE: the canary test. It refers to the tests that were used in the coal mines to detect whether shafts contained toxic gasses. To determine this, a canary would be sent into the shaft of the mine. If the canary came back alive, it was meant to be safe for miners to go in.
* In SRE, the canary test refers to a subset of servers or services where new code is implemented. These servers are then left for an incubation period. If the servers run fine after this period, all other servers get the new code. If the canary servers fail, then these servers are rolled back to the last known healthy state.

Testing is done against the key values of SRE, that is, latency, traffic, errors, and saturation.

* **Postmortem analysis**: Of course, SRE doesn’t mean that mistakes will not happen at all anymore. Multi-cloud environments with distributed systems in different clouds will eventually fail. Systems are getting more complex, mainly because of increasing demands by their users. New features are applied at an ever-increasing speed. Systems are getting more and more intertwined, so they’re bound to encounter issues every now and then because of a deployment mistake, bugs, or hardware failure—remember also that in cloud environments, there is some hardware involved—or, indeed, security breaches.
* In SRE, these issues are opportunities to improve systems and software. As soon as an issue has been solved, a postmortem analysis is conducted. It’s important to know that these postmortem analyses are blameless. Google itself even talks about a postmortem culture or even a philosophy. Teams register the issue, fix it, document the root causes, and implement the lesson learned, all without finger-pointing, all to grab the opportunity to make systems more resilient.

SRE is about constantly learning. It’s about learning by failure and learning by doing. If there’s one message that one should remember after completing this book, it is that multi-cloud architecture and governance are also about learning. Azure, AWS, Google, OCI, Alibaba Cloud, and all the other platforms will change constantly.

It’s a constant transformation.

**Summary**

Systems are getting more complex for many reasons: customers constantly demand more functionality in applications. At the same time, systems need to be available 24/7 without interruption. Cloud platforms are very suitable to facilitate development at high speed, and thus we foresee cloud providers growing fast. In other words, the cloud will definitively grow. This comes with challenges for a lot of businesses. Throughout this book, we discovered that building and managing cloud environments can be complex.

The cloud will grow, and likely the complexity of the cloud will grow too. To ensure reliability, especially with systems that are truly multi-cloud and distributed across different platforms, we should adopt the principles of SRE. The most important principles of SRE have been discussed in this chapter. You should have an understanding of the methodology, based on determining the SLO, measuring the SLI, and working with error budgets.

We’ve learned that these parameters are driven by business risk analysis. We also studied monitoring in SRE and learned how to set monitoring principles. In the last section, some important guidelines of SRE were introduced, covering automated systems, eliminating toil, simplicity, release engineering, and postmortem analysis.

The conclusion of the chapter and this book is to *learn by doing and learn by failure*. The world of multi-cloud is changing rapidly and thus companies will see themselves as being in a constant transformation mode.

**Questions**

1. Risk analysis is important in SRE. What are the five risk strategies, often referred to as PRACT?
2. SRE mentions four golden signals in applying monitoring rules. Latency and traffic are two of them. Name the remaining two.
3. SRE has a specific term for manual work that is often repetitive and should be avoided. What’s that term?
4. Postmortem analysis is a key principle in SRE. True or false: Postmortem analysis is about finding the root cause and finding out who’s to blame for the error.

**Further reading**

For more information on SRE, you can refer to *Practical Site Reliability Engineering* by Pethuru Raj, Packt Publishing.

**Join us on Discord!**

Read this book alongside other users, cloud experts, authors, and like-minded professionals.Ask questions, provide solutions to other readers, chat with the authors via. Ask Me Anything sessions and much more.

Scan the QR code or visit the link to join the community now.

<https://packt.link/cloudanddevops>