CLOUD NATIVE: PRINCIPLES, APPLICATIONS, AND CHALLENGES

PRINCIPLES FOR CLOUD-NATIVE ARCHITECTURE

A cloud-native application is specifically designed from the ground up to take advantage of the elasticity and distributed nature of the cloud. To better understand what a cloud-native application is, it’s best to start with what it’s not—a traditional, monolithic application. Monolithic applications function as a single unit, often with custom-built operation systems, middleware, and language stacks for each application. Most scripts and processes are also purpose-built for the build, test, and deployment. Overall, this application architecture creates close dependencies, making it more difficult to change, test, deploy, and operate systems as they grow over time. What starts out as simple to design and deploy soon becomes complex, hard to evolve, and challenging to operate. By comparison, cloud-native applications make the most of modern infrastructure's dynamic, distributed nature to achieve greater speed, agility, scalability, reliability, and cost efficiency.

PRINCIPLE 1: DESIGN FOR AUTOMATION

Automation has always been a best practice for software systems, but cloud makes it easier than ever to automate the infrastructure as well as components that sit above it. Although the upfront investment is often higher, favouring an automated solution will almost always pay off in the medium term in terms of effort, but also in terms of the resilience and performance of your system. Automated processes can repair, scale, deploy your system far faster than people can. As we discuss later on, architecture in the cloud is not a one-shot deal, and automation is no exception—as you find new ways that your system needs to take action, so you will find new things to automate.

Some common areas for automating cloud-native systems are:

Infrastructure:

Automate the creation of the infrastructure, together with updates to it, using tools like Google Cloud Deployment Manager or Terraform.

Continuous Integration/Continuous Delivery:

Automate the build, testing, and deployment of the packages that make up the system by using tools like Google Cloud Build, Jenkins and Spinnaker. Not only should you automate the deployment, you should strive to automate processes like canary testing and rollback.

Scale up and scale down:

Unless your system load almost never changes, you should automate the scale up of the system in response to increases in load, and scale down in response to sustained drops in load. By scaling up, you ensure your service remains available, and by scaling down you reduce costs. This makes clear sense for high-scale applications, like public websites, but also for smaller applications with irregular load, for instance internal applications that are very busy at certain periods, but barely used at others. For applications that sometimes receive almost no traffic, and for which you can tolerate some initial latency, you should even consider scaling to zero (removing all running instances, and restarting the application when it's next needed).

Monitoring and automated recovery:

You should bake monitoring and logging into your cloud-native systems from inception. Logging and monitoring data streams can naturally be used for monitoring the health of the system, but can have many uses beyond this. For instance, they can give valuable insights into system usage and user behaviour (how many people are using the system, what parts they’re using, what their average latency is, etc). Secondly, they can be used in aggregate to give a measure of overall system health (e.g., a disk is nearly full again, but is it filling faster than usual? What is the relationship between disk usage and service uptake? etc). Lastly, they are an ideal point for attaching automation. Now when that disk fills up, instead of just logging an error, you can also automatically resize the disk to allow the system to keep functioning.

PRINCIPLE 2: BE SMART WITH STATE

Storing of 'state', be that user data (e.g., the items in the users shopping cart, or their employee number) or system state (e.g., how many instances of a job are running, what version of code is running in production), is the hardest aspect of architecting a distributed, cloud-native architecture. You should therefore architect your system to be intentional about when, and how, you store state, and design components to be stateless wherever you can.

Stateless components are easy to:

Scale:

To scale up, just add more copies. To scale down, instruct instances to terminate once they have completed their current task.

Repair:

To 'repair' a failed instance of a component, simply terminate it as gracefully as possible and spin up a replacement.

Roll-back:

If you have a bad deployment, stateless components are much easier to roll back, since you can terminate them and launch instances of the old version instead.

Load-Balance across:

When components are stateless, load balancing is much simpler since any instance can handle any request. Load balancing across stateful components is much harder, since the state of the user's session typically resides on the instance, forcing that instance to handle all requests from a given user.

PRINCIPLE 3: FAVOR MANAGED SERVICES

Cloud is more than just infrastructure. Most cloud providers offer a rich set of managed services, providing all sorts of functionality that relieve you of the headache of managing the backend software or infrastructure. However, many organizations are cautious about taking advantage of these services because they are concerned about being 'locked in' to a given provider. This is a valid concern, but managed services can often save the organization hugely in time and operational overhead.

Broadly speaking, the decision of whether or not to adopt managed services comes down to portability vs. operational overhead, in terms of both money, but also skills. Crudely, the managed services that you might consider today fall into three broad categories:

Managed open source or open source-compatible services:

Services that are managed open source (for instance Cloud SQL) or offer an open-source compatible interface (for instance Cloud Bigtable). This should be an easy choice since there are a lot of benefits in using the managed service, and little risk.

Managed services with high operational savings

Some services are not immediately compatible with open source, or have no immediate open source alternative, but are so much easier to consume than the alternatives, they are worth the risk. For instance, BigQuery is often adopted by organizations because it is so easy to operate.

Everything else:

Then there are the hard cases, where there is no easy migration path off of the service, and it presents a less obvious operational benefit. You'll need to examine these on a case-by-case basis, considering things like the strategic significance of the service, the operational overhead of running it yourself, and the effort required to migrate away.

However, practical experience has shown that most cloud-native architectures favor managed services; the potential risk of having to migrate off of them rarely outweighs the huge savings in time, effort, and operational risk of having the cloud provider manage the service, at scale, on your behalf.

PRINCIPLE 4: PRACTICE DEFENSE IN DEPTH

Traditional architectures place a lot of faith in perimeter security, crudely a hardened network perimeter with 'trusted things' inside and 'untrusted things' outside. Unfortunately, this approach has always been vulnerable to insider attacks, as well as external threats such as spear phishing. Moreover, the increasing pressure to provide flexible and mobile working has further undermined the network perimeter.

Cloud-native architectures have their origins in internet-facing services, and so have always needed to deal with external attacks. Therefore they adopt an approach of defense-in-depth by applying authentication between each component, and by minimizing the trust between those components (even if they are 'internal'). As a result, there is no 'inside' and 'outside'.

Cloud-native architectures should extend this idea beyond authentication to include things like rate limiting and script injection. Each component in a design should seek to protect itself from the other components. This not only makes the architecture very resilient, it also makes the resulting services easier to deploy in a cloud environment, where there may not be a trusted network between the service and its users.

PRINCIPLE 5: ALWAYS BE ARCHITECTING

One of the core characteristics of a cloud-native system is that it’s always evolving, and that's equally true of the architecture. As a cloud-native architect, you should always seek to refine, simplify and improve the architecture of the system, as the needs of the organization change, the landscape of your IT systems change, and the capabilities of your cloud provider itself change. While this undoubtedly requires constant investment, the lessons of the past are clear: to evolve, grow, and respond, IT systems need to live and breathe and change. Dead, ossifying IT systems rapidly bring the organization to a standstill, unable to respond to new threats and opportunities.

THE ONLY CONSTANT IS CHANGE

In the animal kingdom, survival favors those individuals who adapt to their environment. This is not a linear journey from 'bad' to 'best' or from 'primitive' to 'evolved', rather everything is in constant flux. As the environment changes, pressure is applied to species to evolve and adapt. Similarly, cloud-native architectures do not replace traditional architectures, but they are better adapted to the very different environment of cloud. Cloud is increasingly the environment in which most of us find ourselves working, and failure to evolve and adapt, as many species can attest, is not a long term option.

The principles described above are not a magic formula for creating a cloud-native architecture, but hopefully provide strong guidelines on how to get the most out of the cloud. As an added benefit, moving and adapting architectures for cloud gives you the opportunity to improve and adapt them in other ways, and make them better able to adapt to the next environmental shift. Change can be hard, but as evolution has shown for billions of years, you don't have to be the best to survive—you just need to be able to adapt.

Adapted from: https://cloud.google.com/blog/products/application-development/5-principles-for-cloud-native-architecture-what-it-is-and-how-to-master-it