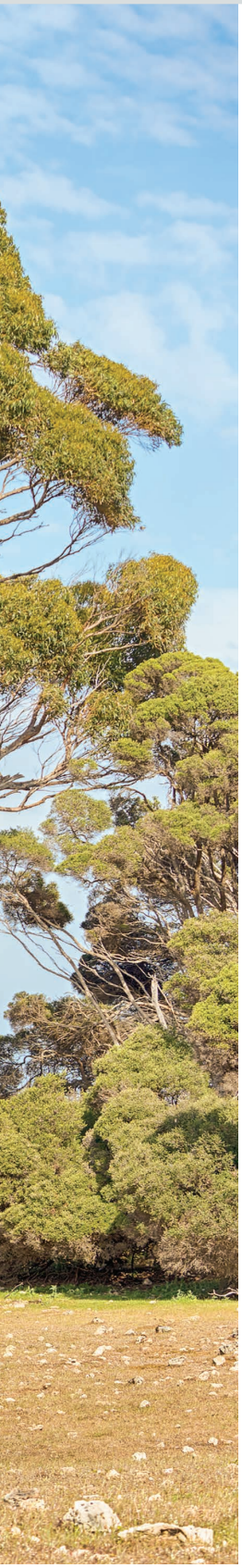
Cloud-Native Applications

Dennis Gannon, School of Informatics and Computing, Indiana University
Roger Barga, Amazon Web Services
Neel Sundaresan, Microsoft Corporation

2325-6095/17/$33.00 © 2017 IEEE



Cloud-native is a term that is invoked often but seldom defined beyond saying "we built it in the cloud" as opposed to "on-prem". However, there is now an emerging consensus around key ideas and informal applications design patterns that have been adopted and used in many successful cloud applications. In this introduction, we will describe these cloud-native concepts and illustrate them with examples. We will also look at the technical trends that may give us an idea about the future of cloud applications.

We begin by discussing the basic properties that many cloud-native apps have in common. Once we have characterized them, we can then describe how these properties emerge from the technical design patterns.

The most frequently cited properties of cloud-native include the following.

1. Cloud-native applications often operate at global scale. While an ordinary website can be accessed anywhere that the Internet is unblocked, true global scale implies much more. It implies that the application's data and services are replicated in local data centers so that interaction latencies are minimized. It implies that the consistency models used are robust enough to give the user confidence in the integrity of the application.
2. Cloud-native applications must scale well with thousands of concurrent users. This is another dimension of parallelism that is orthogonal to the horizontal scaling of data required for global-scale distribution and it requires careful attention to synchronization and consistency in distributed systems.
3. They are built on the assumption that infrastructure is fluid and failure is constant. This concept is the foundation for the original design of the Internet protocols, but applications that were built for execution on a single PC, mainframe, or supercomputer assume the underlying OS and hardware are rock solid. Consequently, when these applications are ported to the cloud, they fail with the first hiccup in the datacenter or network. Even when the failure rate for hardware or networks is extremely small, the law of large numbers guarantees that when you attempt global scale something is always broken or about to break.
4. Cloud-native applications are designed so that upgrade and test occur seamlessly without disrupting production. While not every cloud-native application is intended for use by a million concurrent users spread over the planet, most are designed for continuous operation. Monitoring critical instruments which must not be left unattended is one example. But all applications need to be upgraded, without interrupting normal operations, and the upgrades then need to be tested, so the architecture of the application must allow this.
5. Security is not an afterthought. As we shall see, many cloud-native applications are built from many small component and these components must not hold sensitive credentials. Firewalls are not sufficient because access controls need to be managed at multiple levels in the application. Security must be part of the underlying application architecture.

There are many cloud-native applications that we are all familiar with. The Netflix movie streaming service is certainly one. In her paper in this special issue, "Realizing Software Reliability in the Face of Infrastructure Instability," Cornelia Davis describes how the cloud-native design principles described here enabled that company to survive a catastrophic outage with little degradation. Other examples we use every day include Facebook, Twitter, Microsoft Office 365, and many of Google's cloud collaboration and productivity applications.

In addition, there are cloud services upon which cloud-native applications are built that are, themselves, cloud-native. These include Amazon Web Services (AWS) Kinesis streaming and Amazon Redshift, Microsoft Azure CosmosDB and Data Lake, as well as Google BigQuery.

The Technology of Cloud-Native Applications

The first wave of cloud computing relied on Infrastructure as a Service that replaced on-premise infrastructure with virtual machines running in cloud data centers. While this was suitable for small

applications such as basic web services, it was still very difficult to engineer scalability and manage security at the same time. Additionally, the lack of cloud-based data services, event services, and debugging facilities meant that the programmer had to cobble together solutions from different open source components.

By 2010, platform services that provided abstractions for data management and event handling began to appear in the commercial cloud offerings. In the area of big data analytics, companies like Google and Microsoft were already using internally developed, highly parallel distributed file systems and map-reduce tools. After Google published a research article about their experience, Yahoo! released an open source product called Hadoop that allowed anybody to deploy a distributed file system and analytics tools that anybody could deploy to virtual machines in any cloud. Hadoop may be considered the vanguard of cloud-native applications. But managing scale reliably was still a daunting task. Integrating Hadoop as a reliable, scalable platform service was a challenge for both vendors and expert users.

Microservices, Containers, and Service Fabrics

By 2013, the first major design pattern for cloud-native applications began to emerge. It was clear that to achieve scale and reliability, it was essential to decompose applications into very basic components, which we now refer to as microservices.

Microservice paradigm design rules dictate that each microservice must be managed, replicated, scaled, upgraded, and deployed independently of other microservices. Each microservice must have a single function and operate in a bounded context; that is, it has limited responsibility and limited dependence on other services. All microservices should be designed for constant failure and recovery and therefore they must be as stateless as possible. One should reuse existing trusted services such as databases, caches, and directories for state management. The communication mechanisms used by microservice systems are varied: they include Representational State Transfer web service calls, remote procedure call (RPC) mechanisms such as Google's Swift, and the Advanced Message Queuing Protocol. An instance of a microservice should only have the authorization to access specific classes of other microservices and such authorization needs to be verified by the target service instance.

It must also be possible to encapsulate each microservice instance so that it can be easily started, stopped, and migrated. In other words, a way was needed to package a microservice into a "container" that would be easier to manage than a full virtual machine image. The Linux kernel provided an easy solution to the encapsulation problem by allowing processes to be managed with their own namespaces and with limits on the resources that they used. This led to standards for containerizing application components, such as Docker. Similar mechanisms in the Windows Operating Systems allowed containerization to work there as well. Many container instances can be launched on a single server or virtual machine (VM) and the startup time can be less than a second.

Of course, breaking an application into a web of basic communicating microservices does not solve the problem of how to manage and scale it. What is needed is a type of service "fabric" that can monitor the application components and restart a failed component or start replicas to scale under load.

Google has built a system that runs in their data centers that manages all of their microservice-based applications. This has now been released as open source under the name Kubernetes. The basic unit of scheduling in Kubernetes is the pod, which is a set of one or more Docker-style containers together with a set of resources that are shared by the containers in that pod. When launched, a pod resides on a single server or VM. This approach has several advantages for the containers in that pod. Because all containers in a pod run on the same host, they all share the same Internet Protocol and port space and thus find each other through conventional means such as localhost. They can also share storage volumes that are local to the pod.

UC Berkeley's Algorithms, Machines and People Lab built a container orchestration system called Mesos that is now an Apache open source project. A commercial version is available from the company Mesosphere. A third solution comes from the Docker Company and it is called Swarm. Microsoft has its own Azure Service Fabric that is available to customers but they also support Kubernetes, Mesos, and Swarm on Azure. In the paper in this special issue "Key Characteristics of a Container Orchestration Platform to Enable a Modern Application," Asif Khan describes the properties and responsibilities of a microservice fabric in excellent detail.

Cloud-native applications that use Azure's Service Fabric (ASF) are Skype for business, Power BI (the business intelligence suite), the Azure Data Lake Analytics platform, the CosmosDB global-scale data management suite, and Cortana, the voice controlled digital assistant that can manage your calendar, office appointments, and favorite

music playlists. In addition, ASF manages many of the core Azure services that are running on thousands of machines.

Both Amazon and Google have similar lists of cloud-native apps that run on their microservice platforms. IBM has a microservice platform called the Blue Container Service that is built on Kubernetes. Google and IBM together with Lyft have released Istio, a tool for traffic management between services, policy enforcement, and service identity and security.

Cloud-native application deployment services are not restricted to the big public cloud players. As previously mentioned, Docker and Mesosphere provide solutions. OpenStack based clouds also support Kubernetes, Mesos, and Swarm.

The Cloud-native Foundation exists to promote best practices and community engagement. Their definition of cloud-native is as follows. "Cloud-native computing uses an open source software stack to be:

1. Containerized. Each part of the application (applications, processes, libraries, etc.) is packaged in its own container. This facilitates reproducibility, transparency, and resource isolation.
2. Dynamically orchestrated. Containers are actively scheduled and managed to optimize resource utilization.
3. Microservices oriented. Applications are segmented into microservices. This significantly increases the overall agility and maintainability of applications."[1]

We feel that these are indeed components of cloud-native, but as we argue in the section "Cloud-Native Beyond Microservices," there is much more to the story. In fact, Ken Owens, the chief technology officer of the cloud-native computing foundation, has written extensively on the topic and addresses some of the issue we discuss here.[2]

One item that should be considered as fundamental to microservice design is security. One common approach is role-based authorization, in which each microservice is assigned a role that limits its access to other services. Open Authorization frameworks like OAuth allow microservices to have limited access to other services. Amazon has a built-in service called Identity and Access Management that allows you to specify which entity can have access to specific resources. Azure has Role-Based Access Control that has similar capabilities. Shivpuriya has a good discussion of cloud-native security issues.[3]

There are many excellent examples of cloud-native applications, but only few papers describe the details of the application architecture. Johanson, et al. have described OceanTEA, a system for managing climate data using microservices.[4] An example of a global scale cloud service is the Azure Data Lake. Ramakrishnan et al. describe the microservice architecture in detail.[5]

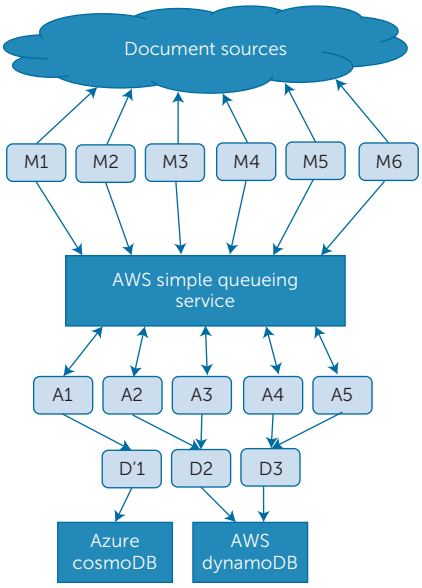


FIGURE 1. Simple microservice cloud-native application.

In their paper in this special issue, "Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation" Davide Taibi, Valentina Lenarduzzi, and Claus Pahl provide excellent insights into the application developer experience with microservice design.

An Application Walkthrough

To illustrate the way a cloud-native microservice design can be built in a bit more detail, consider the following scenario. Suppose you have a mechanism to continuously mine on-line data sources, looking for articles with scientific content. You want to build a cloud-native application that will use machine learning to characterize and classify the topic of these articles and store this information in a database. We will describe two ways to create this application. In the first, we will use microservices plus cloud provided services to handle the data. The second solution will be presented in the section "Cloud-Native Beyond Microservices".

The application decomposition is very simple. As illustrated in Figure 1, we have three types of microservices. M1–M6 are the data monitors that

skim the relevant RSS feeds and news sites looking for relevant sounding articles. A1–A5 are analysis services and D1–D3 are database managing services.

The data monitors will asynchronously push documents into the AWS Simple Queue Service and the analysis services will subscribe to this queue. The queue services act as a capacitor that allows uneven document arrival streams to be uniformly distributed over the analysis services. The data services simply manage access to the databases. The number and type of analysis services can be scaled up or down and replaced without disrupting the computation. The separation of the data services allows us to make changes on the fly there as well. For example, should the developers decide to try Azure's new CosmosDB instead of AWS DynamoDB, it is a simple matter of adding a new database node (D'1) in place of the original. Now a share of the results will go to CosmosDB. If the experiment is successful, more of the results can be routed there by changing D2 and D3. (This example is taken from Foster and Gannon and the source code is available in GitHub and the Docker Hub.)[6]

There is an important additional observation about this example. It is not based purely on microservices because it includes the AWS simple queue service and two database services. These services are highly reliable and themselves cloud-native applications.

Cloud-Native Beyond Microservices

Containerized microservice designs make it possible to build cloud-native applications that meet all the required properties outlined in the introduction. But new capabilities are making it possible to design applications with additional properties and with greater ease. As we have seen in the previous example, there is now a rich collection of services available in the cloud.

Serverless Computing

The major disadvantage of the microservice model as illustrated in the previous example is that we still need to provision a cluster of compute resources to run the services, then manage and scale these compute resources

Serverless computing is a style of cloud computing where you write code and define the events that should cause the code to execute and leave it to the cloud to take care of the rest. AWS's approach to this is called Lambda. For Azure it is called Functions and for Google it is called Cloud Functions. The concept is very simple. In the case of AWS Lambda, examples of the types of events that can trigger the execution of the function are the arrival of a new object in the S3 storage system, or the arrival of a record in the Amazon Kinesis data streaming service.

There is another type of cloud service that is related to serverless concept. These are called "fully managed" services because the service manages all of the infrastructure resourcing, management, and scaling, along with the workflow needed to carry out your computation. There is no need for the user to allocate resources. For example, Azure CosmosDB allows a user to add their own functions and procedures to their databases. These functions are executed by triggers or by user queries.

We can compose fully managed services to build new applications that have all the properties we require of cloud-native. To illustrate this idea, we now show how we can build a cloud-native application similar to the document classifier in the previous section but using fully managed cloud services.

The critical part of the application is the set of microservices that pull items from the queue and apply machine learning to perform document analysis. AWS and Azure both have managed services for machine learning. In Azure ML, one composes a machine learning application by using a "drag and drop" web tool. If needed, we can also add our customer application code.

Once the user has completed the design of their machine learning algorithm, Azure ML has tools to train the algorithm on sample data, and once training is complete, AzureML creates a scalable webservice that can be invoked by any client. To reproduce our application, we use two other managed services: the Azure Event Hub and the Azure Stream Analytics engine. From the cloud portal, one can configure the Event Hub to receive the messages from the data mining services and direct them to the Stream Analytics system. The user can now compose a SQL query for Azure Stream Analytics that will invoke our AzureML service instances. The same query can be coded to invoke a remote storage system like CosmosDB to save the result. The resulting application can now be diagramed as illustrated in Figure 2.

We hasten to add that this approach is not unique to Azure. We can compose a similar solution using AWS Kinesis data streaming services and Amazon ML.

Conclusions

In this short introduction to our special issue on cloud-native applications, we have enumerated the key properties that many cloud-native

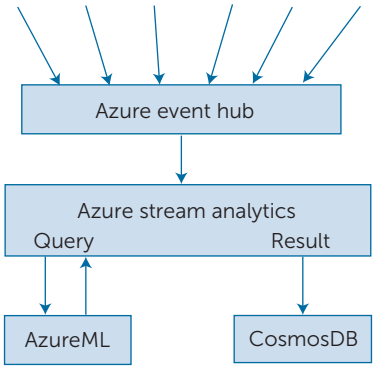


FIGURE 2. A fully managed solution to the document classifier.

applications share. The most common approach to build an application that exhibits these properties is based on a microservices architecture. In this paradigm of computing, the application is decomposed into very small functional units called microservices that communicate with each other as needed through RPC or web service invocations. The result is a design in which the number of running instances of each service can be a few or a few thousand depending upon need. Individual microservices can be replaced with upgraded versions and tested in situ without taking the application off-line. Microservice fabric controllers like Kubernetes, Mesos, and Swarm are resilient frameworks that can be deployed on a cluster of servers or VMs to monitor and manage the microservice ensemble.

Cloud vendors have been busy building high level services for data and event management that are, at their foundation, also cloud-native. These are fully managed services in that they do not require the user to allocate and scale resources in order to use them. In addition, they are highly customizable and composable. It is now possible for cloud users to build entire cloud-native applications without having to allocate or manage resources. And these applications have the same scaling and resilience properties as purely microservice solutions.

Serverless computing based on technology like AWS Lambda, Azure, and Google functions is another important new way to build cloud-native applications.

Future iterations of cloud-native applications are likely to allow application builders to design systems that push computation to the edge of the cloud network. These will likely be built from a combination of serverless and fully managed services.

References

1. Cloud Native Computing Foundation, "Frequently Asked Questions"; https://www.cncf.io/about/faq/.
2. K. Owens, "Developing Cloud Native Applications"; https://www.cncf.io/blog/2017/05/15/developing-cloud native-applications/.
3. V. Shivpuriya, "Security in a Cloud-Native Environment"; http://www.infoworld.com/article/3203265/cloud-security/security-in-a-cloudnative-environment.html.
4. A.N. Johanson et al., "OceanTEA: Exploring Ocean-Derived Climate Data Using Microservices"; http://eprints.uni-kiel.de/34758/1/CI2016.pdf.
5. R. Ramakrishnan et al., "Azure Data Lake Store: A Hyperscale Distributed File Service for Big Data Analytics," *Proc. 2017 ACM Int'l Conf. Management of Data* (SIGMOD '17), pp. 51–63.
6. I. Foster and D. Gannon, *Cloud Computing for Science and Engineering*, MIT Press, 2017; https://www.Cloud4SciEng.org.