O'REILLY®

Cloud Native for the Enterprise

Claudio Caldato

REPORT

Cloud Native for the Enterprise

Claudio Caldato

Cloud Native for the Enterprise

by Claudio Caldato

Copyright © 2020 O'Reilly Media, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Table of Contents

Adopting DevOps and Cloud Native Culture

Developers are excited by the prospect of using cloud native technologies to build the next generation of hyperscale applications and services. You can find a great definition of cloud native in this *New Stack* article:

> Cloud-native is a term used to describe container-based environments. Cloud-native technologies are used to develop applications built with services packaged in containers, deployed as microservices and managed on elastic infrastructure through agile DevOps processes and continuous delivery workflows.

As cloud native technologies and services evolve, the original definition can now be extended to include a range of technologies not strictly related to container, such as serverless and streaming (Azure cloud native, Oracle cloud native, and Amazon Web Services cloud native). Cloud native technologies address specific problems related to designing, building, and managing services that meet the requirements of modern cloud services: scalability, manageability, and reliability. The Cloud Native Computing Foundation (CNCF), a consortium with members from the open source community, startups, enterprises, and major public cloud providers, notes that

> cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.
>
> These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation,

they allow engineers to make high-impact changes frequently and predictably with minimal toil.

CNCF was established to build sustainable ecosystems and foster community around a constellation of high-quality projects that enable enterprises to build more scalable, resilient, and portable applications. These include more than 100 different technologies, and the pace of innovation is quite fast. Projects such as Kubernetes and Istio have release schedules based on three-month cycles.

Several factors introduce complexity in adopting these technologies. Along with the wide range of technologies that need to be mastered, there is the problem of determining which projects will graduate from sandbox and incubation phases to become mature and widely adopted. In addition, enterprises need to move or redesign their services and applications while they keep the business running. Finally, using microservices, containers, functions, Infrastructure as Code (IaC), streaming, and other cloud native technologies requires a cultural shift among developers and operators. Things that were working in the past no longer work; this creates some anxiety among developers who mastered the previous models.

This report is a guide for enterprise IT managers and application development teams who are considering adopting cloud native technologies and want to embrace DevOps culture. It is designed to help you navigate the transition to cloud native technologies and consider the challenges and opportunities of this transition. It also includes some hybrid scenarios in which existing on-premises IT infrastructure is used along with public cloud providers. This report takes an approach that is incremental and not exhaustive; if your enterprise is considering a cloud transition, though, it provides a helpful starting point.

Planning

These general guidelines will help you transition to cloud native:

Become familiar with open source technologies
> Most of the cloud native technologies are open source, so understand how open source works and what its legal implications are. The Linux Foundation guide covers various aspects of open source, such as running an open source program office and managing an open source project.

Evaluate the maturity of the technologies

Companies have different levels of tolerance for the inherent instability that new technologies bring. In general, cloud native projects that have graduated are stable and reliable for demanding production environments, but others might be in earlier stages of development. These incubation projects can evolve quickly and introduce significant architectural changes between releases. If a few enterprises already use the technology or if a public cloud provider supports it, that is a good indicator of maturity.

Determine the availability of support and consulting

Most enterprises need to hire external vendors for consulting or support. The availability of such vendors correlates with the maturity of the technology. Consulting companies usually invest in a new technology when they see a market for it—after the technology is established and adopted not only by early adopters, but also by more risk-averse enterprises.

Identify cloud providers

A good strategy should include a plan to determine which cloud provider is going to be used for the migration to the cloud. Even though most public cloud providers are members of the CNCF or adopters of cloud native technologies, a gap analysis is necessary to determine the best provider. Choosing a specific set of technologies can determine which cloud providers are the best candidates.

Assess your people

Consider the existing skill sets available across the company. This assessment will help determine how long the transformation is going to take and what type of support the team might need, including any skill gaps that need to be addressed, in order to be successful.

Find your champions

Champions don't necessarily need to be existing leads or managers; they need to be people who see the change as an opportunity so that they can motivate and support the other members of the team. Your champions might be employees who are eager to try new technologies and tools or who have already spent time learning cloud native technologies. In some cases,

champions can be new members of the team who are hired with specific industry expertise.

Identify the appropriate scope for the organization's capabilities
Cloud native technologies are designed for cloud/"hyperscale" scenarios, so they allow developers to more easily design and implement high-reliability and scalable services that were a major undertaking only a few years ago.

Cloud native and DevOps culture also offers a lot of opportunities. It has a big community of developers and wide support in the industry, and has been adopted by a range of users, from more risk-tolerant startups to more conservative enterprise customers. There are more than 2,200 contributors for Kubernetes; 1,800 for Docker Engine; 1,300 for Terraform; and 500 for Apache Kafka.

Enterprises should see increased productivity and, with the proper design, increased portability of their services or applications across different cloud providers.

Most developers see cloud native as an opportunity and are eager to be part of a big community of developers who are already using these technologies. For some enterprises, adoption means having the opportunity to contribute to the design.

For enterprise IT professionals, adopting CNCF projects can be challenging; they are based on new technologies, and the learning curve can be quite steep. However, it is possible to create a multistep adoption plan in which, at each stage, only a few technologies are adopted. By focusing on a small subset of technologies at a time, you can build your organization's internal expertise incrementally. This approach also creates a well-integrated stack that is optimized for the specific scenarios and use cases that your enterprise wants to address.

The CNCF Trail Map

Using the CNCF Trail Map, enterprises can adopt cloud native technologies incrementally. As expected, progression along the "trail" requires adoption of more complex software to deliver microservices, serverless, event-based streams, and other types of cloud native apps.

Let's look briefly at the five phases of adopting cloud native applications before we dive into each one in detail.

Phase 1: Containerization and CI/CD

The first phase comprises two parts: packaging an existing service in a container (or several), and setting up a simple Continuous Integration/Continuous Delivery (CI/CD) pipeline that takes the code, builds it, and packages it into the container.

Phase 2: Orchestration and observability

After a service is packaged in a container, it needs a system that can keep it running and allow it to scale if the workload increases. This is where orchestrators and monitoring come into play. The orchestrator manages the service, and there is infrastructure in place to monitor how it behaves at runtime.

Phase 3: Service proxy and service mesh

The next step is to introduce better control over how the different services interact with one another. Service discovery and proxy are the functionality needed to find other services in the network; for instance, in a Kubernetes cluster they provide Domain Name System–like functionality that can be used to map a service name to a specific IP address. Service mesh is an additional layer of functionality that offers more sophisticated control over the connection across services.

Phase 4: Distributed databases and storage

When the service grows, attention shifts to designing the storage infrastructure to handle the increased workload with resiliency and with the appropriate disaster recovery policies. Two key components here are stateful services and distributed databases:

Stateful services

> These are services that need to maintain a state between requests and that rely on some sort of storage infrastructure. A simple example of a stateful service is a shopping cart that needs to store the list of items a user selects. Databases such as MySQL and Postgres can be used to provide the storage support that microservices or functions (in the case of a serverless application) need.

Distributed databases

> These are storage services that cloud providers offer. They are fully managed and support multidatacenter configuration,

enabling data sharing across services running in different geographic regions. Examples of distributed databases are Azure Cosmos DB, Amazon Aurora, and Oracle Autonomous Database.

Phase 5: Messaging, serverless, and streaming

The last step introduces different design principles. Services are not built using a traditional approach based on, for instance, APIs, but they take advantage of a messaging infrastructure. Interaction across services is based on exchanging messages, making the paradigm completely asynchronous. This is where messaging technologies, serverless, and, more generally, streaming come into play.

Now that we've seen what the phases are, let's walk through how to implement each phase and what you'll need to consider as you do.

Crawl, Walk, Run

As described in the previous section, a good strategy is to adopt CNCF technologies with a pace that allows teams in the organization to develop the skills they need to manage them. Let's look at each of these phases in detail.

Implementing Phase 1: Containerization and CI/CD

Consider starting with a real project to modernize an existing service or application by packaging the code in a Docker container. The goal is to become familiar with the process of building a container and running it. Developers can become familiar with the Docker workflow and use client tools and manage the deployment with a container registry. (Container registry is very similar to Maven in the Java world; it offers the ability to publish assets, in this case containers, so that they can be easily distributed.)

Most developers will find Docker (and related technologies) an improvement over existing methodologies. Modernization via packaging in containers also offers a relatively simple deployment model: after a server is configured with the ability to run Docker containers, deploying services and creating live endpoints is easy. An additional advantage of this approach is it shows developers how these technologies offer a better overall developer experience. The Docker

website has excellent documentation and walkthroughs on how to get started.

Getting started with CI/CD

After developers have gained some experience with the container workflow, the next step is to look into DevOps and CI/CD to improve the process of building, testing, and deploying applications and services.

The idea of CI/CD—Continuous Integration/Continuous Delivery —comes from some of the "born-in-the-cloud" companies that adopted a fairly extreme approach (compared with what was typical at that time) of continuously improving their online services by having a fully automated process for building, testing, and deploying new code. In the context of most enterprise environments, there is no need to have this extreme approach, but the CI/CD workflow can benefit the organization even if the goal is not to be able to deploy a new version of a service every few hours.

CI/CD is based on the idea of processing changes in small chunks. With CI/CD, debugging and fault isolation are a lot simpler and quicker as the changes are typically smaller, and different versions have a limited number of code changes that could have caused the problem. It is also easier to test each version by having a smaller feature surface to test. CI/CD enforces best practices by, for instance, requiring developers to include unit tests with their code so that they can be used during the automated process to detect potential regressions in the future. With the containers, a fully automated build-and-test workflow ensures consistency and completeness in the assets produced—these assets will run in any environment. "Dependencies hell" is a problem of the past!

There are many CI/CD solutions suitable for enterprises, from Software as a Service (SaaS) solutions such as CircleCI and CodeShip, to solutions that can be deployed in-house, such as Jenkins, TeamCity, and Bamboo. Public cloud providers also offer CI/CD solutions (Oracle Visual Builder Studio, Azure DevOps, AWS CodePipeline). A cloud provider's solutions are better integrated with its infrastructure, so they can be a good option after you've selected a provider. In-house solutions could be the easiest to adopt because they are likely to be similar to the solutions already in use.

SaaS solutions fit well into the open source workflow model in which a group of developers work on the same project with different development tools. SaaS solutions also support multiple deployment targets and can be used to deploy a service on multiple providers.

The CI/CD workflow

The CI/CD workflow is not significantly different from a standard build system used, for instance, for Java applications, so the adoption of a "cloud native" CI/CD workflow should not be a major undertaking for developer teams that already rely on automation for building and testing.

A typical CI/CD workflow has five steps (see Figure 1):

1. Developers check the code into a repository. Most developers who use cloud native rely on Git, but you can use other source code management systems.

2. Each code check-in triggers the build process. Integration between the source code management system and the build system allows you to track check-ins and coordinate the build process. For Java applications, for instance, you can use Maven or Gradle.

3. After the build process is completed and assets are generated and tested, the code is packaged into one or more containers using Docker tools. The creation of containers is based on a definition file that specifies what should be included in the containers, which network ports should be accessible, and so on (you can find details on the container definition file here).

4. After the container files are generated, they need to be made accessible so that they can be downloaded when needed. The process is similar to publishing Java artifacts in Maven. The container registry can be a public location (e.g., Docker marketplace) or it can be a service provided by a cloud provider (e.g., Oracle Cloud Infrastructure Registry, Azure Container Registry).

5. The container(s) is deployed when the service is launched or updated; for instance, in an orchestrator such as Kubernetes (more on orchestrators and Kubernetes in the next section).
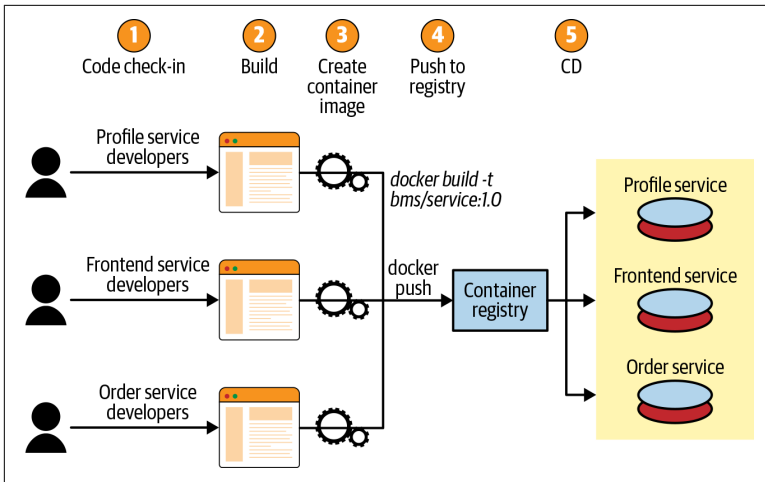
Figure 1. Microservices DevOps flow using native development

In the context of cloud native services, changes are usually small and, with an architecture based on multiple smaller services (instead of bigger monolith services), the workflow handles multiple independent services that need to be deployed and configured after they're built.

Microservices

The ability to update services quickly and frequently in the cloud drove the need to build services in a different way; smaller and independent components can be updated and scaled independently. *Microservices* are an architectural design that addresses these requirements. In adopting cloud native, consider how services are designed and how to establish good practices that development teams can follow. Service design has a big impact on how the service will be able to take advantage of what cloud native offers.

Microservice architectures are based on the idea of decomposing complex operations into a set of independent units (services) that collaborate to provide the required functionality. Communication between services becomes more relevant as a single transaction/request is processed by various services connected through the network (see Figure 2). If a synchronous communication model is used, network latency should be taken into account when designing the system. A good guide on microservices is available at Microservices.io.
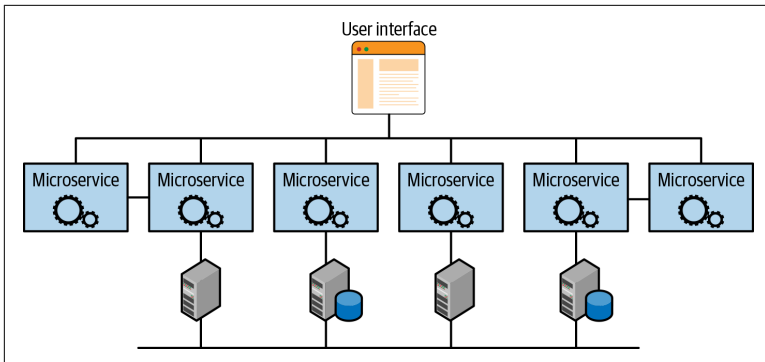
Figure 2. Microservice architecture (adapted from Oracle)

In the microservices architecture, an application is composed of a set of services that need to be deployed and managed as a unit. This is the role of *orchestrators*, which we discuss in the next section.

Implementing Phase 2: Orchestration and Observability

Deploying microservices-based applications, which can be composed of multiple containers that need to be connected and require the configuration of external endpoints, can become a complex process. After you deploy the application, you need to monitor it to ensure that it is always running. Moreover, different components of the applications might need to be scaled up or down in order to respond to changes in the workload.

Orchestrators

Orchestrators are designed to provide a runtime environment in which you can deploy, manage, scale, and monitor container-based applications. They can help with managing cloud native services created with the microservices architecture.

There are various orchestrator solutions provided by public cloud providers (including Oracle Container Engine for Kubernetes, Amazon Elastic Container Service , and Azure Service Fabric), but the one that has been rapidly adopted is Kubernetes. Kubernetes is also a CNCF project.

Kubernetes

On the Kubernetes website, Kubernetes is described as "a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation."

Following are the main features of Kubernetes:

Native support for containers

Kubernetes manages the deployment and execution of containers. Containers can be grouped together in "pods," which are simple execution units that are managed as a single unit, connected without extra configuration (see Figure 3). The master node in the cluster is responsible for managing the nodes in the clusters where pods are deployed. Along with scheduling pods, it determines which node should host pods when they are deployed. The master node includes an API server and exposes the APIs that are used by the kubctl command line or other tools. Etcd is a key-value store used by various subsystems in the cluster.

Figure 3. Kubernetes architecture
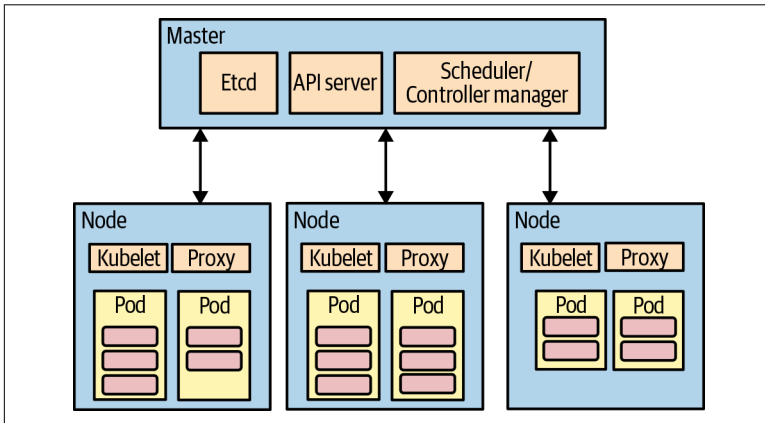
Self-healing

Kubernetes monitors the execution of pods and ensures that they are always running. A Kubernetes configuration can be used to specify, for example, the number of instances of a specific service that should always be running. If one or more instances fail, Kubernetes will handle the process of restarting them to ensure that the system returns to its desirable state as

soon as possible. Kubernetes has a built-in mechanism to monitor the health of services as well.

Autoscaling

Adding or removing instances of a service can happen automatically according to simple rules (e.g., when the CPU utilization of a service goes over a certain threshold, new instances of the service are created). Services can also be scaled down, but scaling down stateless services can be done more easily. For stateful services, the process of scaling down needs to be handled in the service itself to ensure that state is not lost.

Domain Name System management

Kubernetes enables different services to connect to one another using only their names. The association between the name of a service and a specific network address is managed by Kubernetes. Kubernetes also manages ingress: a public endpoint that allows external services or client applications to connect to services managed in the Kubernetes cluster.

Load balancing

Kubernetes creates a load balancer when a service is defined. When multiple instances of the service are required, new pods are created, and the load balancer takes care of distributing requests across all instances.

Rolling updates or rollback

When a new version of a service needs to be deployed—for instance, because there is a new version of a container—Kubernetes stops and restarts each instance of the service in such a way that there is no interruption of service. The process continues until all instances of the running pods are updated. Kubernetes also supports rolling back a service to its previous version.

Resource monitoring and logging

Kubernetes can monitor resources via health endpoints (used for self-healing). It also supports the collection of log files and other runtime metrics that you can use to monitor the health of the entire system.

Running existing applications

After an application is packaged in a container, it can be easily managed by the orchestrator to ensure its availability. Though it

requires more effort, stateful services such as databases can also be containerized as highly available services.

Kubernetes offers some additional advantages. First, it is a portable open source project that is supported by all major cloud providers and that can run on-premises. Second, automation servers such as Jenkins and Oracle Visual Builder Studio have plug-ins to build pipelines integrated with containers. Third, Kubernetes offers namespaces as a way to partition and isolate services (even though it is not a security feature). For instance, using namespaces, a Kubernetes cluster can host two separate environments, such as dev and test. Fourth, Kubernetes supports containers, so pretty much all languages and frameworks can be used; hence, it is not necessary to commit to a specific stack (language, framework, deployment methodology). Fifth, Kubernetes has a comprehensive set of APIs that can be used to manage the cluster, control deployments, and even invoke Kubernetes functionalities within a running service. Finally, Docker now supports Kubernetes, so a developer can have a Kubernetes cluster running in their development machine.

For the adoption of Kubernetes, enterprises have three options:

Self-managed deployment
Kubernetes is an open source project, so you can install it on any infrastructure: cloud or on-premises. However, the reliability of the services running on the cluster depends on Kubernetes being "healthy," so solving production issues, such as a cluster running out of resources, requires a deeper understanding of the platform.

Public cloud services
Oracle, Azure, Google, and other vendors offer managed Kubernetes solutions. Because the health of the cluster is guaranteed by the cloud provider, customers can focus on managing the size of the cluster by setting some scalability rules to determine how the cluster should add or remove resources. DevOps cost is lower and a deep understanding of how Kubernetes works is not required.

Third-party curated solutions
Kubernetes' solutions for on-premises scenarios are similar to the cloud provider solutions because some aspects of

Kubernetes management are automated to provide an experience that is similar to what Cloud Providers offer with their managed Kubernetes solutions.

Managing the cloud environment: Infrastructure as Code

One of the advantages of using a cloud environment is its ephemerality: resources can be created, used, and then deleted in a very simple way. Without fixed assets, one way to optimize costs is to allocate resources when they are needed. Development teams can create, for instance, a set of virtual machines (VMs), deploy some code on them, run some tests, and then delete the virtual machines. This ability to create and delete resources quickly is a big advantage of using public clouds. But it requires a repeatable process for creating complex environments, such as dev, test, and production areas, in a sustainable way. Infrastructure as Code (IaC) is the answer to this set of requirements.

IaC is an evolution of the old shell scripts and automation tools that were used to manage machine configuration. It is a set of tools that can help developers define the configuration of a complex solution (VM configuration, security settings, network configuration, load balancers, storage infrastructure, and so on) via a set of scripts that, after they're interpreted and executed, can create, configure, and connect all of its components.

IaC solutions are feature-rich, with a learning curve that is relatively steep. However, IaC is a critical component in the cloud native tool set and developers will quickly appreciate the role it plays in the DevOps workflow.

In addition to the creation and management of compute, storage, networking, and application, IaC can use cloud management APIs to create accounts, users, and other cloud assets and to manage the entire deployment process with the ability to serialize/parallelize operations.

IaC can also optimize the deployment process by determining configuration changes and apply only required changes to an existing deployment. Most IaC solutions also support modularization and parametrization: scripts can be used on their own or combined to create complex configurations. Like any source code file, IaC configuration files can be managed using a source code management system to track versions and changes.

IaC technologies include Terraform, Ansible, Chef, and Puppet. These IaC solutions support all major cloud providers. Some cloud providers have developed their own solutions, which are specifically designed to take advantage of their cloud infrastructure, such as AWS CloudFormation and Azure Resource Manager. Other providers, such as Oracle Resource Manager, use existing IaC solutions that are already well established (Terraform and Ansible). The end goal of embracing the DevOps culture is to put an initially small but growing set of services in production. Then the bar is higher, and the Ops side of DevOps needs to ensure that the service is performing as expected and respects the service-level agreement. To have a resilient system, a production environment needs to be able to react quickly to changes in workload. If the number of requests grows over time (e.g., peak versus off-peak time), the system needs to be able to increase the available resources so that the system can still respect the service-level agreement in terms of responsiveness, availability, and user experience.

Observability

To ensure that incidents can be quickly investigated and fixed, developers need access to data such as log files, but they also need to collect data on how services are called. Because cloud native solutions are based on microservices, and small(er) components interact to provide the desired capability, developers need access to data about service calls: data that crosses the boundaries of a single service. For cloud native solutions, observability can be divided into two main components:

Intra-service runtime data
> This is collected via log files or through add-on services. Fluentd is an open source technology that can be used for logging. It provides various data sources that can be used to collect different types of runtime data, such as plug-ins for Java applications or MySQL. Fluentd can also send data outputs to public cloud logging services such as Oracle Cloud Infrastructure Logging, data warehouses such as Oracle Autonomous Data Warehouse, and publisher/subscriber queue systems such as AMQP.

Distributed tracing
> In a microservice scenario, the ability to monitor and troubleshoot transactions across services is important. Jaeger is a distributed tracing tool that is integrated with the CNCF stack.

Jaeger and similar distributed tracing technologies offer the ability to do the following:

Monitor distributed transactions
> Track calls across services and "follow" the transaction/request as it is processed across services.

Optimize performance and latency
> Measure time spent in processing a request within a service and the latency introduced by remotely calling other services.

Analyze root causes
> Follow how a request/transaction moves across services. This can be used to identify the root cause of an issue that might manifest on another service.

Analyze service dependencies
> By collecting real-time data on service calls, distributed tracing can be used to build a dependency map that includes multiple levels; for example, Service A calls Service B, which then can call Services C and D.

Prometheus is an open source monitoring and alerting solution used to monitor services by recording events in a time series. Events can be visualized and analyzed with a power query language; with the same query language, users can define alerts that are managed by a built-in Alert Manager. Jaeger is also integrated with Prometheus to provide a powerful metrics/visualization and alerting platform. Oracle Cloud supports Prometheus through the Grafana plug-in.

Scalability

The ability to react to changes in workload is an important aspect of making the service resilient. Requests to the service can vary over time, so the service needs to be able to react to spikes and still guarantee a predefined quality level. To handle increased workload, a service can either *scale up* or *scale out* (see Figure 4).

Scaling up is the process of increasing the resources available to the service (such as memory) so that it can handle a bigger workload. The problem with this approach is that it typically requires restarting the process; hence, it cannot be done without downtime. The

other downside is the constraints placed by the underlying system, such as the memory of the VM or bare-metal server.

Scaling out is the model used with cloud native solutions. A service can increase the workload it can handle by adding more instances of the service with a load balancer that distributes the traffic across all available instances. The load balancer can use different policies to distribute traffic, including round robin and weighted response time. Scaling out can be done manually by increasing the number of instances running—for instance, by using APIs and the Kubernetes management console—but it is not a sustainable model. However, in most cases it is not possible to foresee peak traffic.

Autoscaling is the process of automatically increasing the number of instances of a service based on the change in workload (as illustrated in Figure 5). Kubernetes offers the ability to set autoscaling policies based on metrics such as CPU or memory usage as well as custom metrics such as transactions per second. You can find more information about Kubernetes autoscaling on the Kubernetes documentation website.

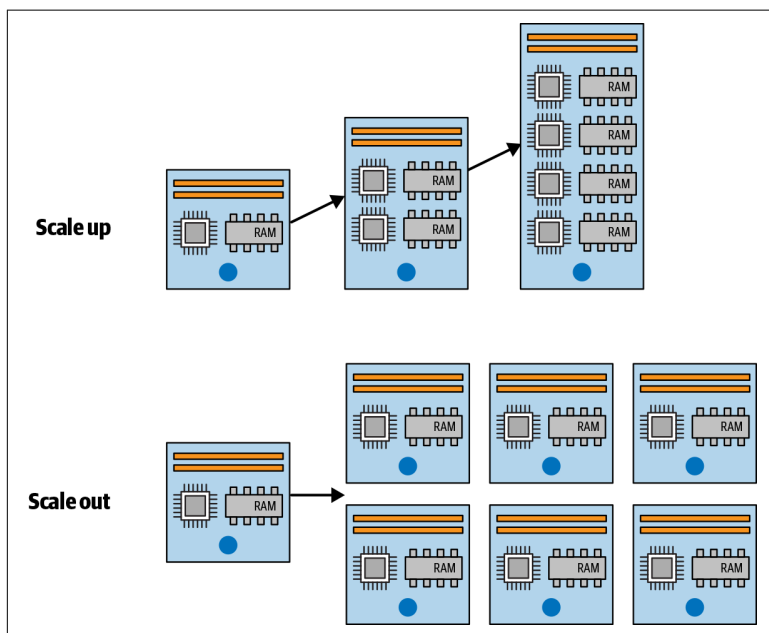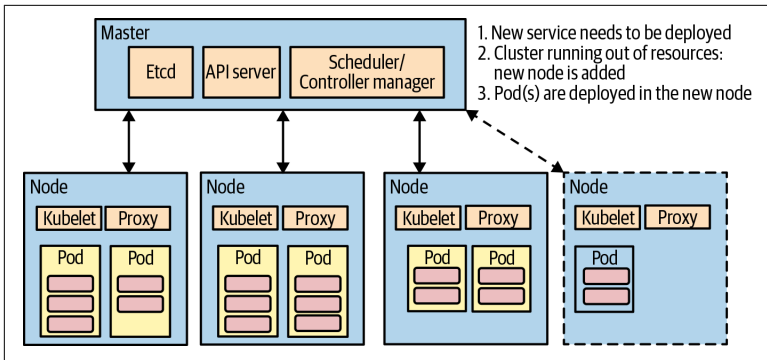*Figure 4. Scaling up versus scaling out*

Figure 5. Kubernetes autoscaling

Implementing Phase 3: Service Proxy and Service Mesh

The more you buy into the cloud native design approach, the more moving parts you need to deal with. You'll want to decompose your architecture into independent and manageable components using microservices or functions to make updates and scalability more efficient. This creates the need to manage communication between services. Phase 3 focuses on the service mesh.

Service proxy and discovery

The ability to associate running code with a specific endpoint and make that endpoint discoverable, so it can be used by other services, is known as service proxy and discovery. It is functionality built into Kubernetes (for example, CoreDNS is the default cluster Domain Name System service installed in Kubernetes), so developers can become familiar with it as soon as they begin deploying their services on a Kubernetes cluster. But service proxy and discovery are not sufficient for scenarios in which, for instance, traffic needs to be managed dynamically or policies need to be defined to route traffic through services. This aspect is managed by service mesh solutions.

Service mesh

Service mesh introduces an additional level of control over communication across services. It is based on a small agent that is deployed within the pods and that takes control of all incoming and outgoing traffic. A control plane is also installed, through which it is possible to define policies and monitor traffic across services using a "sidecar," which is a container that is added to the microservice's pod.

At a high level, the service mesh enables you to control how traffic is managed. For instance, if you have two versions of the same service, you can redirect traffic between them: 50% on one version, 50% on the other. You can route all the traffic from one version to the other, route traffic from specific clients to one version only, and so on. Service mesh provides the ability to create circuit breakers to prevent your service from stalling while it is waiting for an upstream service to respond. By allowing the service to fail faster, you can avoid a cascade of wait/retry cycles across potentially multiple services in your application.

With *ingress/egress*, service mesh also allows you to control incoming and outgoing traffic to the system. Along with policies defined by firewalls and other cloud infrastructure components, you can use ingress/egress to control subsystems of the architecture. For example, if the system is composed of subsystems managed by different teams, each team can define an ingress/egress to control access to the services it manages.

Tracing and logging is another feature that service mesh offers. Communication across services is logged, and tracing information can be used to analyze connection patterns across services. This feature complements the monitoring services that cloud providers offer to, for instance, monitor network traffic.

Istio is one of the most-used service mesh technologies. Linkerd and Envoy are other examples of service mesh that are currently incubating and graduated projects, respectively, in CNCF.

Implementing Phase 4: Distributed Databases and Storage

With three-tier applications (see Figure 6), access to data was granted directly to a database via a connection string that contained the information required to establish a connection with the database. In general, it required a limited number of connections to the database. In the cloud world, there is a new generation of storage systems designed to be distributed across multiple machines or even different regions and datacenters. These distributed databases are designed to handle a high number of connections and provide relatively low-latency access to data in such a way that they can be used directly from any service that might need to access their data. Examples of the new generation of distributed databases include Amazon

Aurora, Oracle Autonomous Database, Oracle Database Cloud Service, Azure Cosmos DB, and traditional open source applications such as Postgres and Cassandra offered as managed services.
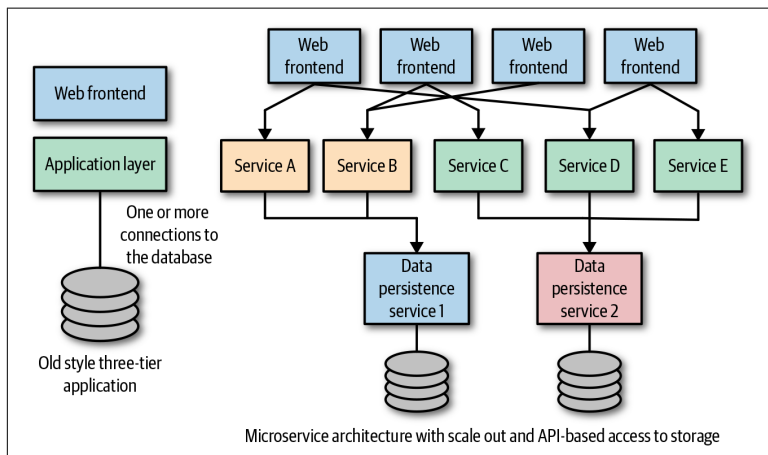

Figure 6. Three-tier versus microservice architecture

In the microservices world, a common practice is to abstract the data access with a service that exposes a set of APIs instead. Access to the database is isolated within the service. One of the benefits of this solution is that you can change the database schema without affecting other services. This is a good practice when using modern distributed storage services because it offers a more robust solution.

Cloud native applications that are distributed by design need an additional layer of storage to improve overall performance. There are two cases for which caching can improve performance:

Hot versus cold data
> You can use in-memory databases such as Redis to provide low-latency access to hot data. *Hot data* is the data being used by the application that needs to be readily accessible. Examples include session states and other temporary data that needs to be available between requests. *Cold data* is data for which access can be less immediate but that needs to be stored permanently on a database such as Oracle Database or MySQL.

Data sharing in scale-out scenarios
> Cloud native applications use scale out to handle increased workload. With multiple instances, state needs to be shared, as

requests are not guaranteed to be processed by the same instance every time.

Implementing Phase 5: Messaging, Serverless, and Streaming

Phase 5 represents the transition to a different paradigm for designing and implementing services. So far, we have assumed that an application can be decomposed into multiple independent units (services) that expose their functionality through a set of APIs; for example, REST APIs.

Messaging

An evolution of this method of designing a distributed application is to make the invocation completely asynchronous by using messages instead of invoking REST APIs. In a messaging-based model, a centralized messaging brokering system offers a publisher/subscriber model in which services post messages that other services receive and process. A messaging-based system has the advantage of making the system more resilient and potentially more scalable, but at the same time, it introduces a higher level of complexity in how services are implemented. All major cloud providers offer messaging services; examples include Oracle Integration Cloud, Oracle Data Integration, Azure messaging services, and Amazon Simple Queue Service. Other options include deploying an open source messaging app such as Apache Kafka or RabbitMQ via containers on an orchestrator such as Kubernetes. This can be a more flexible solution, but it requires additional operations costs.

Serverless

Serverless is becoming a mainstream technology in the cloud native world. In the early days of AWS Lambda, serverless was a new way to deploy simple operations (functions) that were invoked by a set of events such as, a change in a database or a new element in a stream. Now serverless is more pervasive and offers a lot more flexibility and richness of features, which makes it a solution for a wider range of scenarios.

There is no doubt that serverless is going to have an important role in how services will be designed in the future; it offers some significant benefits with respect to existing solutions. Serverless is based

on a pay-per-use pricing model, so, in theory, it offers a big economic advantage. Serverless services also offer a very simple deployment model; there is no operations cost and the application is autoscaled based on the workload.

Some see serverless as the silver bullet that is going to make it very easy to design, deploy, and manage complex distributed applications. Unfortunately, we are not there yet, but we can expect its relevance to increase in the near future.

Vendor lock-in is a potential risk to consider as most serverless solutions are based on a proprietary infrastructure that makes the code less portable. The code of each function has some degree of portability, but it can have so many dependencies on the infrastructure that porting it to another provider can become quite expensive. There are open source solutions such as OpenFaaS and Fn Project that can reduce the impact of vendor lock-in. Oracle Functions is a managed service that implements Fn Project. Fn Project can run anywhere—on premises and on public clouds.

Streaming

There are scenarios for which the ability to analyze data in real time is important, such as telemetry from Internet of Things (IoT) devices, website clickstreams, or application log files. In all these cases, data needs to be processed as it becomes available instead of using traditional batch processing. All cloud providers offer services designed to enable stream processing (such as Amazon Kinesis and Oracle Streaming), typically with an ingestion service that can handle a high volume of continuous data that can be processed as a set of messages or events. There are also open source solutions such as Apache Kafka that can be used on both cloud and on-premises scenarios. Cloud providers also offer integration with Kafka, an example is Oracle's Streaming support for Kafka.

Whereas the ingestion part of the process is handled by high-volume, highly scalable services from by the cloud provider, the downstream processing needs to be designed to handle the workload (in terms of number of events per second) that the ingestion pipeline can generate. In most scenarios, the downstream services need to be able to process the events in a timely fashion and keep up with incoming events to avoid ingestion queues growing over time. Designing any streaming application requires careful consideration

of expected throughput and the amount of time that can be spent in processing each request if real-time processing is also a requirement.

To avoid the complexity of designing an ad hoc system, cloud providers offer complementary services that can be used to create an end-to-end streaming application. Serverless can be a good solution for streaming applications because it provides the ability to scale automatically as the workload increases or decreases; hence, it reduces the complexity and can simplify the design.

We have walked through the five phases of adopting cloud native applications. The five phases represent one way to think about the migration to cloud native technologies, though there are clearly more that could be considered. For enterprise customers, adopting cloud native does not always mean a full migration to the cloud. There are scenarios for which public cloud providers need to be used along with existing on-premises IT Infrastructure.

Hybrid Scenarios

An enterprise might want to keep some services or applications on its on-premises infrastructure if, for example, it might not be possible to port a legacy application because of the technologies it uses. In other cases, privacy or governance requirements might require on-premises infrastructure along with the public cloud.

There are a lot of nuances, and the adoption of a hybrid model depends on the specific requirements of each enterprise; for instance, timing (short-term versus long-term strategy), security requirements, and technologies used for existing services.

On-Premises and Cloud Native

For scenarios in which some services must run on-premises over the long term, start by sharing the same hosting infrastructure (e.g., Kubernetes) so that the programming model is the same for both on-premises and public cloud services. A service designed and packaged in such a way can, potentially, be deployed without significant changes to both on-premises and public cloud infrastructure. With this built-in flexibility, it is possible to move workload from or to the public cloud based on specific criteria. In this scenario the on-premises infrastructure needs to be modernized to adopt all of the cloud native technologies that are required to mimic the features

available in the public cloud. In most cases this means running an orchestrator such as Kubernetes with a container registry.

The storage infrastructure used by stateful services requires additional thinking: storage services available on public cloud providers are unlikely to be available for on-premises deployment, and the latency involved when an on-premises service connects to a remote storage service in the cloud is probably not acceptable. A way to avoid the need to modify the code if the application needs to run on-premises is to design access to storage through a set of APIs that abstract the underlying storage infrastructure. The on-premises storage infrastructure can be replaced with an existing solution without changing the code on the services that need to access it.

In other cases, legacy applications or services cannot be easily ported to the cloud. In this case, the goal is to create a good interface between the new services designed and deployed for a public cloud environment and the legacy application running on-premises. The legacy applications can be "exposed" via a modern interface that will make them "look like" modern services. Kubernetes's Service Catalog can help to address this type of scenario. The major downside of this approach is the additional latency introduced by the fact that not all services are running on the same hosted environment.

Examples of on-premises cloud native software include Oracle Linux Cloud Native Environment, Azure Stack, and AWS Outposts.

Multicloud Scenarios

For some customers, betting on a single cloud provider might not be an option. If this is the case for you, a simple strategy is to partition the services/applications and data in such a way that they can be hosted on a single cloud provider. Another option could be to create a uniform infrastructure on all cloud providers so that the workload can be moved from one cloud provider to another. The best way to ensure this is to use those vendors that use unmodified/unforked open source software. In the case of Kubernetes, using CNCF-conformant distributions can provide portability of your container workloads to other clouds. In the case of IaC, using vendors that support multiple clouds, such as Terraform, can provide flexibility in deployment and management across clouds. It is always a good idea to keep any dependency on the cloud infrastructure isolated.

Data has a high level of stickiness on cloud infrastructure. If the amount of data is significant, moving it across two different cloud providers can be costly and time-consuming. One way to reduce potential dependencies is to use storage technologies that work across cloud providers, such as those based on Object Storage Service (OSS). The disadvantage is that you cannot benefit from fully integrated and managed solutions provided by the cloud provider, so managing a storage solution can be costly.

Some cloud providers offer interoperability across clouds. One example is the Oracle-Azure partnership, which provides interoperability by interconnecting those two public cloud vendors, which provides interoperability between these two public cloud vendors with low-latency connectivity, unified single signon (SSO), and a shared support model.

Future Opportunities

Cloud native and DevOps culture are transforming how applications and services are built and managed. They provide a way to take advantage of the cloud infrastructure to create scalable and reliable services.

Whereas in the past adopting cloud native technologies was seen as optional, now it has become a mandatory step for enterprises that need a solid IT infrastructure in order to be competitive. Cloud is now a mature technology, with all cloud providers offering a wide range of very sophisticated services that were inconceivable not long ago. Databases that can support massive scaling and different data models or AI services are examples of the competitive advantages that cloud technologies can provide.

Adopting DevOps is only the first step of the process; it clearly has a cost in terms of time and resources, but the return can be significant. Cloud native is the stepping stone to fully utilizing the wide range of technology that the cloud can provide. With the services being offered by all major cloud providers increasing quickly, enterprises have an opportunity to incrementally develop their services and reduce their time to market considerably.

About the Author

Claudio Caldato is a technical program manager on Facebook's infrastructure team, working on various projects related to cloud infrastructure optimization and reliability. Prior to Facebook, he was a senior director of product strategy with the Oracle cloud team, where he worked on the grand unified theory of cloud native development, which aimed to empower developers to build the next generation of cloud native applications. Before joining Oracle, Claudio worked in the Azure Hyperscale and IoT teams at Microsoft and was one of the founding members of the team that pioneered OSS there. He also worked on various OSS projects and some of Microsoft's key products and technologies, such as the .NET Common Language Runtime, Office, and Visual Studio.