Cloud Native Application Development – A New Computing Paradigm
IT Strategy and Architecture from Oracle

ORACLE WHITE PAPER | FEBRUARY 2017
Table of Contents

Introduction 1
What is Cloud Native? 1
Cloud Services 2
  Cloud-First 2
  Infrastructure As Code 3
The Software Delivery Process 4
  Agile Development 5
  DevOps 6
Application Architecture 7
  Environment Options 8
  Service-Based Architecture 13
  Twelve-Factor Applications 16
Conclusion 16
Appendix 17
  Cloud-Native Principles 17
“Six decades into the computer revolution, four decades since the invention of the microprocessor, and two decades into the rise of the modern Internet, all of the technology required to transform industries through software finally works and can be widely delivered at global scale.”

- MARC ANDREESSEN

WHY SOFTWARE IS EATING THE WORLD

Introduction

The process of delivering software is ever evolving. Mostly, that evolution is a result of technology shifts and advancements that make incremental improvements to the process. But occasionally a seismic shift occurs that transcends minor variations on the old routines. Born out of necessity for the rapid delivery of Internet-based services on a global scale, the Cloud Native approach to application delivery has emerged.

Shifting to a Cloud Native approach can provide enormous benefits to the business. A recent survey conducted by the Harvard Business Review has concluded that the success of an entire business may be determined by the speed with which it can capitalize on new technologies and adjust IT operations.¹ To become more responsive to business opportunities, changes such as moving workloads to the cloud and adopting more agile development processes were among the recommendations cited in the report. Such steps represent a starting point in a Cloud Native IT transformation.

Cloud Native application development is especially apropos for systems of differentiation and systems of innovation, as defined in Gartner’s Pace-Layered Application Strategy². These systems tend to require faster delivery times and more frequent updates in order to satisfy changing business practices and evolving market opportunities.

Many dominant technology-driven businesses have been benefiting from and exploiting a Cloud Native application strategy to their advantage for several years. In fact, the technologies and best practices available today are largely harvested from some of the most successful players in the industry.

What is Cloud Native?

Essentially, the gist of Cloud Native is that applications should be built in the cloud, for the cloud, maximizing the benefits of the cloud. More specifically, Cloud Native can mean different things to different people. Some definitions describe a very specific checklist of technologies, tools, and processes that, when used properly, should provide

meaningful results. However, businesses do not all operate under the same circumstances and environments. What works for some may not be beneficial or advisable for others. Rather than provide a very prescriptive definition, this paper will examine the larger goal of maximizing the benefits that cloud computing can provide by recalibrating certain aspects of IT.

To begin, we’ll focus on three key objectives that apply to most views on Cloud Native:

1. Capitalize on the Cloud: adopt a Cloud-First strategy and extract the greatest value from cloud services
2. Deliver value early and often: rapid, agile software delivery
3. Build applications to exploit the cloud: different than building applications that run in the cloud

These objectives prompt a review of various aspects of IT, including application architecture, the software delivery process, and the manner in which cloud services are integrated into the IT environment. The following sections will address each of these areas.

Cloud Services

There are many different types of services available today to support the development of cloud-based applications. Many services focus on providing an environment that is as identical as possible to what IT had on premises. The reason is simple – the move to cloud can be relatively easy and painless if nothing needs to change.

Take Infrastructure-as-a-Service (IaaS) for example. You can subscribe to a service that offers a virtual environment just like the virtual or physical environment you used on premises. You bring your own application platforms, frameworks, databases, integration platforms, etc. and set up environments the way you’ve been doing for years. However, the value in this is limited. You still need to purchase licenses for any licensed products; you still need to install, configure, patch, and maintain platforms; and you still need to do this for every application, environment, and deployment instance. You can go the route of creating and maintaining virtual images, however, this doesn’t eliminate your responsibilities – it only makes them somewhat easier to manage. Thus, the “lift-and-shift” approach will work – but it isn’t optimal. It works well when you can’t, or don’t, want to change what you have or what you are doing.

Cloud-First
An alternative to lift-and-shift is to port your current platform or software stack to a Platform-as-a-Service (PaaS) offering. This helps to eliminate the purchasing of licensed products as well as installation and maintenance responsibilities. Now you can more fully use services from the cloud and bring less of your own software with you. The goal in this scenario is to only bring what the cloud doesn’t offer. This goal, or principle, carries over into the use of Software-as-a-Service (SaaS) offerings as well. One should avoid building applications altogether when there are suitable alternatives already available in the cloud.

So the first principle for Cloud Native is “Cloud-First”: look for capabilities that are available in the cloud and use the greatest value services that suit your needs. Reduce the amount of software for which you are responsible to the lowest reasonable level. Leverage application platforms, databases, integration platforms, data analysis services, caching, load balancing, etc. and build your custom software around those services.

A drawback in following this principle is the potential for vendor lock-in, (or cloud lock-in). To avoid this, consider the trade-offs between proprietary products, open source products, and custom code. If a proprietary product is best suited for your needs, then either use a PaaS service that provides it or use an IaaS service and bring your own license. As long as the PaaS service is compatible with the licensed product, there is no more lock-in than what you had before. In fact, with a pay-as-you-go service contract, you are less committed to the PaaS service than you were to the licensed product.

If you find little added value in a proprietary product, then open source might be a better option. With open source you have more leeway to move between cloud vendors. Likewise, the costs associated with suitable cloud services are likely to be lower. You will find service offerings with open source software stacks available, or you can package open source software along with your custom code and deploy all of your software and dependencies yourself.

Oracle Cloud

The Oracle Cloud is designed to be a complete and integrated cloud environment. It consists of many cloud service offerings for SaaS, PaaS, and IaaS. These service offerings represent the next generation of applications, platforms, and infrastructure, now available as cloud services.

Oracle cloud services support a variety of application development and migration use cases, including:

- The development of modern Cloud Native applications
- The development of traditional and SOA applications
- The development of departmental applications and “low-code” or “no-code” web apps
- The development of mobile applications and backend APIs
- The migration of existing workloads to the cloud
- The integration and management of all of the workloads above and with on-premises applications

Despite offering many PaaS cloud services based on their industry-leading licensed products, Oracle is committed to providing services for popular open source technologies. In addition, IaaS offerings from Oracle make it possible for you to bring your own software stacks, licensed products, and virtualization technologies to the cloud.

Oracle cloud services are available in several data centers worldwide. And, with Oracle’s “Cloud at Customer” offering, you can also put the Oracle Cloud in your own data center, behind your corporate firewalls.

Infrastructure As Code

Cloud computing brings a service-based approach to computing infrastructure and platforms. That is, service instances can be created, modified, and destroyed via APIs as well as via a user interface. So, not only can you
bypass costly, time-consuming processes required to procure, install, and configure new hardware and software, you can also integrate the operational aspects of environment management with your own procurement processes.

There are several ways to benefit from this feature of cloud computing. For example, production environments can be far more agile in terms of managing resources, handling usage spikes, and setting up high availability and disaster recovery environments. Likewise, there is far more flexibility in creating temporary environments to handle application versioning, troubleshooting, and short-lived production applications. There are also tremendous benefits for application development and testing. Given the ratio of development and testing environments to production environments, the frequency with which such environments are altered, and the ability to create and destroy them on demand, it is no wonder that many companies begin using the cloud specifically for development and testing purposes.

API-based management enables the creation of service instances and entire environments to be orchestrated and automated using tools such as Chef and Puppet. Software that defines service instance creation, e.g. the “recipe” or manifest, can be managed the same way development teams manage the source code for their projects. This practice is known as “Infrastructure as Code”. Infrastructure as Code (IaC) is another key principle to Cloud Native computing. It allows environment infrastructure configurations to be managed and versioned together with the code that runs in the environment. It also supports environment versioning and duplication.

The Oracle Cloud supports IaC in two key ways. First, it offers APIs to manage cloud service instances. This makes it possible to integrate the management of services with process automation tools. Second, it provides services that provide and support process automation. The Oracle Developer Cloud Service supports automation with tools such as Chef and Puppet. This service also provides several tools for the dev/test environment including a version management system, issue tracking system, collaboration wiki, Agile project management, and IDE integration. The Oracle Orchestration Cloud Service provides task automation and workflow. It automates task execution by calling REST APIs, scripts (Chef recipes, Python, etc.), or 3rd party automation frameworks. It can apply automation on both on-premises and cloud infrastructure.

The Software Delivery Process

Now that development, test, and production environments can be provisioned in seconds or minutes as opposed to weeks or months, it makes sense to recalibrate the software delivery process to exploit this benefit. Besides, with many companies adopting a mobile-first application rollout strategy, frequent releases with incremental changes enable the business to more rapidly reach customers, provide value, adjust to market changes, and overcome issues.

The goal of software development should be to get the minimum viable product to market in the shortest possible time and then quickly deploy incremental releases to continuously improve the overall product. Defining a minimum viable product is mainly a business decision, whereas defining incremental releases involves a collaboration of business stakeholders and IT leadership.

In some cases, the minimum viable product may be an entire application – primarily if an existing application is being replaced. Users may not be cut over to the replacement application until all previous functionality is made available. An agile or iterative process will likely have minimal impact, other than to support user feedback and acceptance testing as features of the replacement application are being developed and critiqued. However, for new applications the calculus will likely be different. The minimum viable product may be a small portion of the overall application as it is currently envisioned. By releasing an initial product and adding features along the way, users may recognize tangible benefits very quickly. Likewise, as the product is released, the overall feature set and operations of the application can change according to user feedback and evolving business objectives.
Agile Development

Agile development is meant to break long release cycles into smaller, incremental releases. It takes an “inspect-and-adapt” approach by enabling end users and business stakeholders to test drive incremental feature releases and modify subsequent application requirements according to feedback and market forces. Agile isn’t a methodology per se, but rather an approach that eschews long delivery cycles. This philosophy is well described in the Agile Manifesto and the 12 Principles put forth by the Agile Alliance.

There are several methods that promote Agile principles including Scrum, Kanban, Lean, Extreme Programming (XP), Rapid Application Development (RAD), and Agile Unified Process (AUP). Of these methods, Scrum appears to be the most widely adopted. Scrum operates around time-boxed events called Sprints and Scrums. Sprints represent a period of time in which product features are completed – generally about one month in duration. Business and IT leaders are free to define their Sprint durations based on time frames that suit the project at hand. Scrums are brief daily events used to review accomplishments since the previous Scrum, plan the next daily objectives, and discuss impediments that may affect the ability to achieve the Sprint goals.

A key aspect of Scrum is that it pertains to the management of projects rather than the architectural approach used by developers to implement projects. This makes it flexible enough to be used on many different types of projects involving different technologies and implementation techniques. Scrum is a well defined methodology. The Scrum Alliance offers training and certifications for various types of participants defined by the methodology.

Water-Scrum-Fall

Agile development works best when the entire organization supports an agile delivery regimen. It starts with the way a business defines objectives and milestones, envisions projects, funds IT efforts, and manages initiatives. These activities feed into project requirements and delivery expectations, which in turn begin the software development process. For agile to work most effectively, the business should think in terms of incremental gains. If the business mindset is based on a quarterly or annual cadence, then there will likely be a mismatch with IT’s weekly or monthly cycle times. The full benefits of agility may be missed due to the inability of the business to inspect-and-adapt as rapidly as has been made possible.

---

Likewise, as shown in Figure 2, the tail end of the delivery process involves release coordination, operations planning, provisioning, and deployment into a production environment. Often, for reasons of quality control, production environment stability, and formalities that may pertain to manageability, resources, dependencies, security, and regulatory compliance, production releases are treated as major events rather than daily routines. Again, there can be a critical cadence mismatch with an agile development process. The value of agile development is greatly curtailed if software releases only reach production environments on an (at-best) quarterly deployment schedule.

The term Water-Scrum-Fall was coined to describe organizations that adopt agile development but continue to operate in a Waterfall approach both upstream and downstream from the software development portion of the delivery process. This state of affairs may be an undesired current reality due to an incomplete transition, or it may be a necessary compromise based on factors that are deemed of greater importance than a “fully” agile delivery process. Either way, it is important to consider all three phases of the process – pre-development, development, and post-development – and how well the organization as a whole can maximize the benefits of an agile approach. This describes the third principle for Cloud-Native application development.

DevOps

The tail end of the delivery process, e.g. the transition from testing phases led by developers on through the deployment of software into a production environment, presents a unique challenge. In order to be more agile and reduce delivery times, software needs to transition this phase more rapidly and more often. However, the groups that take responsibility for this part of the process, namely quality assurance (QA) and operations, generally have a different agenda. They aren’t usually motivated to move quickly, rather they are motivated to reduce defects and maintain a smooth running production environment. This motivation drives the processes and activities that they perform and the deliberation and rigidity with which they perform them.

DevOps is a movement that is meant to address this issue. As stated quite well on Wikipedia, “DevOps… emphasizes the collaboration and communication of both software developers and other information-technology (IT) professionals while automating the process of software delivery and infrastructure changes. It aims at establishing a culture and environment where building, testing, and releasing software can happen rapidly, frequently, and more reliably”.

By focusing on collaboration and communication, an organization can (to some extent) break down barriers between groups and work more effectively toward a common goal. The intent is to transition from a “throw it over the wall” approach where one group hands responsibility off to another, to an approach where everyone works as a team throughout the entire process. Development is made keenly aware of QA and operations’ needs, and vice versa.

DevOps also promotes process automation in an effort to streamline the delivery process. The most common forms of automation pertain to test automation, continuous integration, and continuous delivery.

Continuous Integration, Delivery, and Deployment

Continuous Integration (CI) involves automated code merging and building. It brings together (integrates) code changes that developers have made into a common build in order to quickly determine if one change has syntactically caused any problems, (i.e. “broken the build”). The advantage of continuous integration is that errors can be identified much quicker – often the same day; or overnight in the case of nightly build cycles. This is most beneficial for larger teams where frequent code changes are being made. CI helps to avoid problems that occur

when code paths diverge and become incompatible, often requiring a great deal of analysis and rework in order to merge everything together.

If the build is successful, the next step is to perform a battery of tests. In the past, testing was a very manual process. Not only did it take time and skill to set up environments for testing, many tests were executed by hand with users clicking through screens, tabs, and links, verifying results, and inspecting logs and database tables. A long testing cycle fit well with long release cycles and infrequent production deployments. However, the evolution of development and testing tools, the availability of cloud services, and best practices for managing releases has made a paradigm shift possible. It is now theoretically possible to not only perform a build every night, but also to release code into production as well.

The ability of configuration management (CM) tools to instantiate cloud environments, and the practice of Infrastructure as Code to manage environment configurations, makes it possible to move the build through a series of testing environments. CM tools can also coordinate the execution of automated testing in each environment. As a result, an error-free build can be tested automatically and made available for production release – at least to the extent that testing can be automated. This is the notion of Continuous Delivery – Continuous Integration, plus environment provisioning and configuration, plus automated testing. Automation of the delivery process is our fourth Cloud-Native principle.

It is quite likely that human intervention, or at least human approval(s), will be necessary at some point in the delivery process. Perhaps not all testing can be automated, perhaps a final sanity check by a QA representative is mandated, or perhaps the decision to deploy to production needs to be reviewed and approved. For these reasons the CM tools need to support having a human in the loop.

The trade-offs of human intervention obviously need to be weighed on a case by case basis. Some applications are necessarily more constrained than others. However, some organizations have taken the “fail fast” approach. That is, it is alright to risk failing in production if: a) errors will be spotted quickly, b) it is easy to roll back to the previous release, c) bug fixes / new releases will be available in a very short period of time, and d) the gains from frequent and rapid releases outweigh the downside of deploying a potentially error-prone release. This mindset embraces the notion of Continuous Deployment, which is essentially Continuous Delivery all the way into production environments.

Oracle Developer Cloud Service

The Oracle Developer Cloud Service is designed to support the Cloud Native software delivery process. To support agile software development it provides features for managing and tracking Sprints, Boards (containing issues), Story Points (work metrics associated with issues), and project reports. It supports CI/CD with automated build, test, and delivery frameworks. It also supports Webhooks that send notifications to registered remote services about internal events such as a Git push, an issue update, a merge request update, or a build completion. Oracle Developer Cloud Service also helps support DevOps via features such as an issue tracking system to manage tasks, defects, and features, and collaboration capabilities including live ATOM/RSS feeds to report on project activity and Wiki pages for collaboration and documentation sharing.

Application Architecture

The third objective of Cloud Native is to build applications to exploit the cloud. More precisely, build applications that can:

- Deploy quickly and easily
- Deploy to a variety of environments including on premises and cloud environments
• Deploy to secure operating environments with resource isolation
• Scale in and out as needed
• Achieve high performance characteristics
• Support high availability requirements
• Integrate easily and securely across cloud and on-premises environments
• Minimize cost
• Avoid vendor/cloud lock-in

The following sections address these objectives by outlining some of the options that are available.

Environment Options
To build applications for the cloud, one must consider the types of environments that are commonly available in the cloud. For the purpose of this paper, we’ll work with the following set of environment types:

• Application server environment
• Virtual machine environment
• Container environment
• “Serverless” environment
• “Low-Code” and “No-Code” environments
• Mobile environment

Application Server Environment Option
One example of an application server environment is Oracle’s Java Cloud Service (JCS). This service offering provides the Oracle WebLogic Server (WLS) platform as a cloud service. An application runtime environment (or application server), such as Oracle WebLogic Server, provides many features to support applications in a production environment. These include security, monitoring, management, transaction support, connection pooling, messaging, and resource management. In addition JCS provides options for load balancing, data caching, and clustering for high availability. The list of features reflects a mature product offering.

Figure 3: Java Cloud Service With and Without Domain Partitions
With this option, you are building applications and services that become hosted by the application server. Your deployment is limited to the custom code and frameworks that provide functionality beyond what is provided by the application server. This is highlighted in Figure 3 using red boxes for the software you provide and gray for what is provided for you.

Given that applications operate within the environment, the environment can be considered “bounded”. That is, the cluster environment to which applications are deployed can only run on servers or cloud service instances that have been pre-configured with the application server. In order to horizontally scale the application, the application server must first be provisioned in a new service instance. This characteristic will not affect most applications, however, it could be problematic for applications that are very dynamic and at least occasionally need to scale out to handle a very large number of concurrent requests.

In terms of isolation, JCS has two capabilities. First is the container-based approach that is a standard feature of Java EE application servers. Applications running on JCS are protected by security services provided by the container. Resource management is handled at the JCS service instance level. You can configure the amount of infrastructure resources you want to have allocated for each instance.

The second form of isolation, multi-tenancy, is a custom feature of JCS. It provides a means of partitioning tenant-specific application data, configuration, and runtime traffic within the Java EE domain; see Figure 3. JCS supports the ability to export and import domain partitions. Exporting and importing partitions enables you to move partitions from one domain to another, including the applications that are deployed to the partition. This feature is useful for replicating partitions across domains and for moving domains from a development to a production environment. You can use WLST scripting, a REST API, or an administrative console to export and import partitions.

In terms of vendor or cloud lock-in, a Java EE-based environment has both advantages and disadvantages. Since Java EE is defined by its specification, applications written for Java EE (should) run on any compliant Java EE implementation. Applications that make use of features beyond the specification will likely require some rework in order to port them to another vendor. Also, since WLS is available both as a licensed product and a cloud service (JCS), it is possible to deploy WLS on premises or on other vendors’ clouds, making it a viable option for virtually any environment. The cost associated with this option will likely be higher than with other options, however, it does have a very rich feature set.

Overall, using any application server in a cloud-first or Cloud Native initiative can be problematic due to the potential for vendor/cloud lock-in. However, given the ubiquity of Java, the ability to port from one Java EE implementation to another, the maturity of WebLogic Server, and the ease with which you can move between on-premises deployments and cloud services, there are reasons to consider a Java EE-based solution that conforms to the Java EE specification. This is particularly true for applications that are relatively stable in terms of load requirements and those that can benefit more from the Java EE feature set than from hypothetical scalability and agility characteristics.

**Virtual Machine Environment Option**

As illustrated in Figure 4, a virtual machine (VM) encapsulates an entire runtime environment within a portable virtual image. VMs, (provided by the subscriber), run on top of a hypervisor (which is generally provided and maintained by the cloud provider), which in turn runs on top of the host operating system, (also maintained by the provider). The hypervisor manages the VM runtime environments and provides isolation between environments as well as isolation between the VMs and the host operating system.
Oracle provides several environments for deploying virtual machines, including:

- Oracle Compute Cloud Service, in either a dedicated or shared environment
- Oracle Ravello Cloud Service, for migrating virtual environments "as-is" to the cloud
- Oracle Cloud Machine, for running cloud services on hardware installed in your private data center

There are also options for compute services where you can bring your own hypervisor, such as Oracle Bare Metal Cloud Compute Service and Oracle Dedicated Compute for SPARC Model 300.

The deployable unit for a VM is a VM image; (VM1-VM4 in Figure 4 above). The image includes its own operating system, referred to as the guest OS, as well as any platforms, frameworks, and custom code that are necessary to support the application. The image in its entirety can act as the unit of deployment for an application.

It is worth noting that an image can host a dynamic environment to which applications and services are deployed and redeployed. For example, a VM image can host an application server environment or a container-based environment. This will result in a nested deployment: a VM deployed to compute resources and applications or services independently deployed to the VM. For the purpose of this paper, we'll avoid detailing this option by considering it to be a virtualized implementation of the other two environment options. Instead, we'll focus on the deployment of static VM images that are configured for specific applications or services.

There are several advantages for using a VM-based deployment strategy, including:

- The inclusion of a guest OS provides you more flexibility to choose your OS type and version.
- The degree of isolation between environments is quite good. From a security standpoint, this option has been well tested and has demonstrated itself to be very secure.
- The ability to partition hardware resources for each VM is well supported.
- The ability to manage, migrate, and deploy images into different environments works well for supporting multiple data centers and cloud environments.

There are, however, certain disadvantages when viewed in the context of Cloud Native. First, is the size of a VM. Since a VM not only contains the application code, but also contains an OS and all of the other software necessary to run the application, it is rather large. It also takes time to boot up and bring the application online. Furthermore, if you run multiple instances for scalability and availability, and/or your application consists of more than one
independent module or service, then you will have many VMs running at once, which means many copies of the OS, platforms, and frameworks. Scalability and rapid deployment will be a challenge. Also, since there are different VM vendors and service providers to consider, cost and vendor lock-in can become a factor.

Given the advantages and disadvantages, VM technology is often perceived to be better suited for either application migration, (migrating applications as-is from on-premises systems to the cloud), or for setting up your own cloud infrastructure. For Cloud Native applications, one should consider a lighter-weight alternative, such as containers.

**Container-Based Environment Option**

A container is similar to a VM in that it represents an isolated runtime environment for an application. However, rather than layering a hypervisor and guest operating systems on top of the host OS, containers leverage isolation capabilities of the host OS. Linux provides such virtualization and isolation capabilities in its kernel; and Docker is a popular container technology that exploits those capabilities.

Given that Docker is an open-source technology, many organizations are using it. Compared to VMs, Docker provides a much smaller image to build and maintain. Shipping Docker images around is much more efficient. In addition, the technology behind Docker enables it to reuse common libraries across images, which means resource utilization is much more efficient. Since there is no guest OS to boot up, Docker processes can be started much faster than VMs.

Container-based environments are ideal in many ways for Cloud Native applications. They enable you to deploy applications quickly and easily to a variety of environments. They support resource isolation. They can be scaled easily and dispersed to provide high availability. With Docker, there is an open-source implementation, so cost and vendor lock-in are not an issue.

As illustrated in Figure 5, Oracle offers two options for container-based environments: the Oracle Container Cloud Service and the Oracle Application Container Cloud Service. The Oracle Container Cloud Service (CCS) allows you to compose, deploy, orchestrate, and manage Docker container-based applications on the Oracle Cloud. This service is meant for deployments where you want to maximize flexibility and control. It provides an environment where you can:

- Build Docker images that contain your application code
- Manage images in a registry
- Discover and pull images from the registry and load them into Docker containers
- Define pools of resources that represent collections of compute resources (hosts) for container runtime environments
- Deploy Docker containers to your resource pools
- Manage orchestration policies that determine the number of Docker containers to run, and how and where to deploy them
- Manage collections of containers that represent a multi-container based application (stack)
- Scale your environment in and out on demand

Oracle Application Container Cloud Service (ACCS) is the second option for container-based applications. ACCS is meant for lightweight applications where simple, rapid deployment and automation are desired. It supports development using Java SE, PHP, and Node runtime environments along with thousands of available libraries and application frameworks. It uses Docker container technology, however it does not expose container constructs to the service subscriber. Instead, it provides a UI for managing the environment as a collection of service instances.

With ACCS, the developer creates an archive that contains an application and all code dependencies. The service automatically creates a container image from the archive and deploys it. The automation provided by ACCS enables developers to deploy with fewer steps than would ordinarily be required.

“Serverless” Environment Options
“Serverless” environments are those that manage the server aspects of computing for you. They typically operate on a function or service basis where functions, often exposed as services, are executed on demand. The cloud provider is responsible for starting virtual machines or containers and executing the function. The provider also must ensure that sufficient resources are provided. The consumer of a serverless environment is usually billed according to the actual resources that are used. Given that the cloud provider may elect to spin down serverless processes that are not active, this form of computing is most useful for processes that can operate asynchronously. It is also useful for processes that are seldom used since the cost is based on actual usage rather than the resources that are reserved for an application in advance.

“Low-code” and “No-code” Environments
Platforms are also available for the “Citizen Developer”, i.e. folks within an organization with little or no coding skills that can benefit from designing and deploying web-based applications. These platforms provide drag-and-drop design tools that generally run within a standard web browser. The platform will translate pages that are created with the tool into actual executable code. The runtime environments are easy to set up and little is required in order to deploy an application.

In order to enable integration with other resources some of these platforms provide access to external resources such as APIs, databases, messaging systems, etc. In addition, some will provide the ability to inject custom code into the application in order to introduce custom logic or to customize the end user experience.

“Low-code” and “no-code” environments are great for departmental applications, online reports, and other efforts that would traditionally constitute a shadow IT effort. They can also be used by business users to extend the functionality of SaaS applications.

Oracle provides this form of environment with its Application Builder Cloud Service (ABCS). With ABCS, there is nothing to install or download – applications are built from the browser. You can add functionality with HTML5 and Javascript, you can integrate with your enterprise resources by invoking APIs via REST services, you can extend your Oracle SaaS applications, and you can even do single sign-on (SSO) with Oracle SaaS applications.
Mobile Environment

Given the ubiquity of smart, connected mobile devices today and the frequency with which customers are using mobile devices to access information online, many companies today are taking a “mobile-first” approach to their application development strategy. In many cases, the development of custom mobile apps either precedes or is done in parallel with browser-based applications.

While this strategy helps to achieve business goals, it can lead to problems if the two types of applications are developed in isolation. Common functionality, back end integration, security, and access to data may be duplicated. As new features are implemented, the two delivery channels may conflict with each other making for a confusing experience for the user community.

An ideal mobile cloud platform will provide the ability to easily and rapidly develop apps for mobile devices and to integrate with a common set of APIs that can be shared across delivery channels. Mobile application development should tie into software development infrastructure in order to take advantage of continuous integration and delivery capabilities. It must also provide a means to download and update mobile apps through an app store or a similar means of distribution.

The Oracle Mobile Cloud Service provides these capabilities, and more. It offers support for device-native, hybrid, and no-code app development. It also includes capabilities for push notifications, data synchronization, user authentication, location awareness, device management, and user management. Mobile applications can easily and securely integrate with back end APIs, which can be used to access common enterprise services, applications, and data.

Service-Based Architecture

Service-Based Architecture (SBA) is an umbrella term that refers to any architecture that applies service-oriented principles including service-oriented integration, service-oriented architecture, and micro-services architecture. SBA, in its various forms, helps to address Cloud Native objectives pertaining to integration and agility. As such, it becomes our fifth Cloud-Native principle. The various forms of SBA are introduced below.

Service-Oriented Integration

Cloud computing extends the notion of distributed computing beyond the boundaries of the corporate datacenters. It now becomes commonplace to have applications scattered about, hosted by SaaS providers as well as providers of PaaS and IaaS services that support custom applications. In an effort to support application integration, either within the datacenter or across datacenters, service-oriented integration is a fundamental principle.

![Service-Oriented Integration Example](image)

Service-oriented integration (SOI) is the practice of exposing application functionality as consumable services and providing a means to loosely couple the provider of a service from the consumers of a service. Services are remotely accessed via well-defined interfaces. The service interface is a technical construct, also referred to as an
application program interface (API). While there are no constraints on which technologies to use, open standards such as REST and/or SOAP over HTTP(s) are highly recommended.

Loose coupling is necessary to allow the service provider and service consumer to evolve independently. It enables location independence – the ability for the service provider to get deployed in different places without affecting the consumers. An intermediary is required to provide loose coupling. The consumers connect directly to the intermediary, and the intermediary relays requests to the provider.

The intermediary can perform other tasks such as technology mediation, routing, security checks, etc. However, the intermediary should not introduce business logic or change the context of the request in any way. This is referred to as “smart endpoints, dumb pipes”. The intermediary is part of the pipe connecting a consumer to a provider, not an intelligent actor affecting the behavior of the service. All business logic should remain within the service implementation, which resides behind the service interface.

The Oracle Cloud supports service-orientation, either in the form of SOI or SOA, in a number of ways. First, many of the cloud services promote the development and management of APIs. APIs are used to integrate enterprise and mobile applications and they are used as an integration mechanism for accessing both Oracle and non-Oracle SaaS applications. Second, the Oracle SOA Cloud Service and API Platform Cloud Service are designed to facilitate and manage SOI and SOA assets. They provide capabilities such as mediation, discovery, management, and security that are essential to service-oriented environments. Also, service instances in the Oracle Cloud are managed as services via APIs. Service-orientation is a common theme throughout the Oracle Cloud.

Service-Oriented Architecture

Service-oriented architecture (SOA) applies service-orientation to the architecture of the application. Not only are functions exposed as services, the code that comprises services is built as a separate, independent entities. Services may be logically separated, e.g. having well defined boundaries within the code base but packaged and deployed along with the application, or physically separated – packaged and deployed independently. The degree of separation must be determined by the development organization.

Figure 7: Service-Oriented Architecture Example

The ideal process for building a SOA application is to identify services during the initial stage of application design. Services are commonly identified for their reuse potential. The initial application will generally be the first consumer of services and other applications may also use them as well. In order to support reuse, services should be well defined. Therefore, in addition to an interface specification, the service provider is encouraged to write a service contract. The contract is meant to describe the service in plain language in terms of what it does and what qualities of service it provides. The interface and contract are then made available for other development groups to discover and leverage for their needs.

In addition to the integration objectives of Cloud Native computing, SOA helps address objectives related to agility and scalability. Services, particularly those that are independently packaged and deployed, can be versioned much more rapidly than larger monolithic applications. They can also be deployed multiple times according to load
requirements. For instance, a service with heavy load could have more instances running than one that is not used as heavily.

There is one important factor that can limit the agility of SOA: dependencies. Dependencies, such as shared resources, create an environment where it becomes difficult to manage and make changes to one service without affecting others. The most common issues arise from shared packaging and shared data sources. While shared packaging is fairly easy to address, shared data sources present a larger problem – especially when applications and services share a common database and/or schema. Unless dependencies are eliminated, agility will be compromised.

**Microservices Architecture**

Microservices architecture (MSA) is a recently emerging architectural approach, designed to maximize agility. It addresses the issue of dependencies by specifically architecting services to eliminate them. In addition, it maximizes autonomy by limiting the size of services and by limiting the unit of deployment to just one service. It also seeks to eliminate dependencies between services as well as synchronous interactions between the service and any of the resources it uses.

Unlike SOA, MSA is not focused on enterprise-level reuse. In fact, reuse across applications can be considered a detractor to the extent that it creates a system of dependencies. Rather, Microservices are quite simply small (micro) portions of an application that provide functionality autonomously via APIs; (services). As such, publication and mediation for the sake of reuse are of less concern, while greater emphasis is placed on avoiding dependencies, such as sharing common data sources.

![Microservices Example](image)

**Figure 8: Microservices Example**

Following a Microservices architecture, the development of an application is organized as the prioritized development of a group of services. The intent is to develop the core set of services — those that comprise the minimum viable product, as quickly as possible. Additional services are then developed to provide greater functionality. Existing services can be rewritten whenever a change needs to be made to existing functionality. Since services are small, rewriting them is not as time consuming as revising larger and more complex services and monolithic applications. Likewise, since services are often accessed via REST APIs, and any one service is easily rewritten, it is not so important to standardize on specific languages and frameworks. Speed to delivery is often viewed as a priority, which means that services are written in whatever language the developer chooses in an effort to expedite delivery.

From a Cloud Native perspective, MSA is the pinnacle in terms of agility, rapid deployment, and scalability. Each service is completely autonomous and can be revised, rewritten, deployed, and scaled (in or out) to a degree beyond what any prior approach has been able to achieve. Since service instances can be instantiated and eliminated easily, a dynamic service discovery capability is needed in order to help locate instances of services that are currently available.
It is easy to see why Microservices are often deployed in containers such as Docker. Containers are small, lightweight, isolated environments that are perfect for small, autonomous application services. Although Microservices can be deployed to any environment, they are a more perfect match for container-based environment. This is why the Oracle Container Cloud Service and the Application Container Cloud Service are both a natural fit for Microservices applications.

**Twelve-Factor Applications**

Just as best practices and design patterns exist for application development, best practices also exist for building Cloud-Native applications. One such collection of best practices is referred to as the Twelve-Factor App. This is a methodology for building cloud-based applications to promote portability and enable build/test automation, continuous deployment, and scalability. It is the result of lessons learned from building hundreds of applications for the cloud. It can be applied (to some extent) for building applications with any programming language, platform, and architecture style.

As the name suggests, the methodology consists of twelve factors that describe how one should approach the development of applications for the cloud. Each factor is well documented to explain its meaning and purpose. The factors cover topics such as:

- How code should be tracked, built, and released
- How dependencies and backing services should be handled
- How configuration and port binding should occur
- How to handle scalability and disposability
- How to approach logging and administration

If you are building applications for the cloud - especially if you are looking to maximize the agility, scalability, and speed to market that MSA provides, then this methodology is worth considering. In fact, it is called out as our sixth principle for Cloud-Native application development.

Oracle supports the development of Twelve-Factor Applications using the Developer Cloud Service along with either the Application Container Cloud Service or the Container Cloud Service. The Developer Cloud Service supports factors pertaining to maintaining a common codebase, such as: supporting builds for multiple environments, defining dependencies for builds, enabling continuous integration and deployment, and implementing the Twelve-Factor build, release, and run stages.

The Application Container Cloud Service and the Container Cloud Service support factors pertaining to the runtime environments, such as: requiring all dependencies to be packaged with the deployed application, strictly separating configuration from application code, supporting the use of environment variables for configuration, providing a declarative way to attach backing services as resources, running applications as Docker container processes, providing a means to store session state in external storage and caching services, supporting external port bindings, scale-out concurrency, statelessness, disposability, and logging to stdout.

**Conclusion**

The Cloud Native computing paradigm represents an evolutionary step in custom application development. It goes beyond migrating and porting applications to run in the cloud. Rather, it focuses on how to exploit cloud computing to maximize its benefits. In order to achieve this, IT organizations need to focus on how they use cloud services, how they architect applications for the cloud, and how they develop those applications.
The Oracle Cloud offers many services that can help you achieve your Cloud Native goals. There is a wide variety of services available that span many aspects of IT including application development, data management, analytics, integration, application management, and security. Application development services are available to support traditional application development efforts, SOA-based applications, modern Twelve-Factor Microservices applications, and the secure integration of all of those types of applications that comprise a hybrid IT environment.

For more information about Cloud Native and Microservices, please contact your Oracle account team. For more information on IT strategy and cloud computing, please visit the Hybrid IT ETS page. For specific questions or to provide feedback on this whitepaper, please send an email to its_feedback_ww@oracle.com.

Appendix

Cloud-Native Principles

The following tables capture the Cloud-Native principles mentioned in this paper.

Cloud-First Principle

<table>
<thead>
<tr>
<th>Principle</th>
<th>Cloud-First</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement</td>
<td>When evaluating solution options, first look for capabilities that are available in the cloud and use the greatest value services that suit your needs.</td>
</tr>
<tr>
<td>Rationale</td>
<td>Reduce the amount of software for which you are responsible to the lowest reasonable level. Leverage application platforms, databases, integration platforms, data analysis services, caching, load balancing, etc. and build your custom software around those services.</td>
</tr>
</tbody>
</table>
| Implications    | • Solutions will be deployed in a cloud environment unless there are prohibiting circumstances, e.g. security / compliance issues, unsupported technologies or integration requirements, unreasonable data movement or network usage patterns.  
                  • SaaS applications should be used in situations where the application is "good enough" or can be adequately configured or extended; especially when little advantage is gained via a proprietary solution.  
                  • Trade-offs must be considered for the use of proprietary over open source technologies |

Infrastructure as Code Principle

<table>
<thead>
<tr>
<th>Principle</th>
<th>Infrastructure as Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement</td>
<td>Manage infrastructure configurations and provisioning workflow definitions in the same manner you handle the application code.</td>
</tr>
<tr>
<td>Rationale</td>
<td>Given the ability to provision environments via APIs, and the tools to manage and execute provisioning workflows, it is now possible to treat environment provisioning as a function of software. By managing environment provisioning code along with application code, you can achieve a better overall configuration management experience. The entire runtime environment can be versioned as a single entity and executed in a repeatable manner.</td>
</tr>
</tbody>
</table>
| Implications    | • Tools that support IaC will need to be used  
                  • Processes or scripts will need to be written and tested for each environment and application  
                  • Application releases may require changes to provisioning processes or scripts  
                  • No manual steps should be required to provision or configure an environment |
### Agile Delivery Principle

#### Statement
Strive for agility in all phases of the delivery process including pre-development project initiation and planning, and post-development release management and operations activities.

#### Rationale
While an agile software development process generally enables faster time-to-market, agility can be compromised if pre-development and/or post-development activities are too rigid. Strive to be agile across all three phases in order to maximize benefits.

#### Implications
- Pre-development planning should accommodate agile development by identifying project iterations that map well to development delivery cycles
- A minimum viable product should be identified in order to set an initial delivery objective
- It should be expected that solution objectives and requirements will change from one release to another as business objectives evolve
- The development team should work closely with operations and release management in order to enable frequent releases. A DevOps philosophy is advised.
- A “fail fast” approach should be considered in order to avoid long testing and burdensome QA cycles, particularly for solutions that are deemed low risk

### Delivery Automation Principle

#### Statement
Strive to automate all aspects of solution delivery from nightly builds, up to, and including production deployment.

#### Rationale
There are definite advantages in terms of efficiency and time-to-market that can be realized with the ability to automate many tasks related to software builds, environment provisioning, testing, and deployment.

#### Implications
- This principle builds on the principle of Infrastructure as Code.
- Automated testing tools will be required.
- A “fail fast” approach to production releases will be necessary in order to expedite and automate production deployments
- A plan for rolling updates can enable production deployments to occur for specific servers and sites as opposed to updating all production servers at once
- A monitoring system and rollback plan should be devised in order to detect and back out problematic releases quickly and without waiting on bug fixes

### Service-Based Architecture Principle

#### Statement
Service-based architecture (SBA), in its various forms, must be followed according to defined project objectives and desired characteristics.

#### Rationale
All forms of service-based architecture have advantages that should be exploited. Although there are trade-offs to consider when choosing one form over another, each should be considered and evaluated in order to determine the most appropriate architecture approaches for a given solution.

#### Implications
- Some analysis will be necessary early in the software development life cycle in order to determine the most appropriate application of service-based architecture
- All forms of SBA will require the specification and development of APIs
- An API-First development strategy should be adopted
- SOI and SOA will require some form of intermediary in order to provide loose coupling between service consumers and providers
- The creation of a human readable service specification, or service contract, is advisable for services that are meant to be shared with consumers that are not part of the initial solution or solution delivery team
Appropriate security measures need to be taken in order to protect services or APIs that are exposed to untrusted consumers or over untrusted networks.

### Twelve-Factor Applications Principle

<table>
<thead>
<tr>
<th>Principle</th>
<th>Twelve-Factor Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement</td>
<td>Follow proven best practices, such as Twelve Factor Applications principles, for the development of Cloud-Native applications.</td>
</tr>
<tr>
<td>Rationale</td>
<td>Some organizations have been developing Cloud-Native applications for years and have begun to document best practices. It is best to learn from others’ missteps and adopt their best practices where appropriate.</td>
</tr>
</tbody>
</table>
| Implications | - Development and testing guidelines may need to be reviewed and updated in order to achieve certain objectives  
- Build processes, release processes, and configuration management practices may be affected by certain best practices  
- Some best practices will affect the way applications are deployed and managed, therefore it may be necessary to review best practices with Operations team members |
Integrated Cloud Applications & Platform Services

Copyright © 2017, Oracle and/or its affiliates. All rights reserved. This document is provided for information purposes only, and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document, and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group. 0116

Cloud Native Application Development – A New Computing Paradigm
February 2017
Author: Dave Chappelle