Introduction to Cloud-native applications

First, I will try to provide a layman language.

Cloud Native applications are the software applications that are

designed and developed specifically to leverage the cloud computing principles

and

take the full advantage of cloud native technologies and services,

which means these applications

are built and optimized to run inside any cloud environments

by utilizing the cloud advantages like scalability, elasticity, and flexibility.

In simple words, cloud native applications are built for cloud environments

so that

the organizations can take complete advantage of cloud provider services and technologies.

Now let us try to understand the official definition of cloud native applications from the Cloud Native

Computing Foundation (CNCF).

So, if you see here, cloud native technologies empower organizations to build and run scalable applications in modern dynamic environments such as public, private and hybrid clouds.

So, the cloud environment can be anything, it can be it can be your own data centre, it can be GCP,

azure, or it can be a hybrid combination of private cloud and public cloud.

Regardless of what type of cloud computing you are trying to use, your cloud native applications will

work.

So how this is possible is when we are trying to build cloud native applications, we will try to leverage

technologies like containers, service meshes, microservices, immutable infrastructure and declarative

APIs.

So, all these technologies make your applications to run on any cloud so you will not get vendor lock in with any cloud environment.

On top of these cloud advantages.

These techniques also enable developing loosely coupled systems that are resilient, manageable and

observable.

Resilient means they can withstand any failures manageable you already know they are easy to manage and observable means we'll get to know everything about our applications, how it is working, are there any issues?

So, if anyone asks you what is the definition of cloud native application inside an interview or inside

any project, first try to understand whether the opposite person who is asking is a technical person

or a non-technical person.

If he is a non-technical person, please share this layman definition.

Whereas if the person is a technical person, you can share the official definition from the Cloud Native Computing Foundation.

Important characteristics of cloud-native applications

The very first main character of Cloud Native application is **microservice**.

When you are building microservice based applications, which are like loosely coupled and smaller in

nature, then that gives a flexibility to you to develop them parallelly and deploy and scale independently.

After building microservices like separating your business logic, you will obviously **containerize** your

applications with the help of Docker or any other containerization software.

So what are these containers we already discussed?

These are typically packaged and deployed using Docker containers.

These containers provide a lightweight and consistent environment for running applications and making them highly portable across different cloud platforms and infrastructure.

With the help of these containers, only the code will work very similarly regardless of which cloud

environment you are trying to deploy. You take this container, you try to deploy inside your local

system and you deploy inside the AWS, GCP, Azure Cloud in all the places it is going to work in very

similar manner.

Whereas this is not the scenario with a monolithic application.

Your monolithic application won't give such kind of flexibility.

You need to put a lot of efforts to bring that consistency across all the cloud platforms.

The next character of cloud native applications is they provide **scalability and elasticity**.

Since you are building your applications based upon microservices and with the help of containers,

they can be easily scaled horizontally allowing them to handle any kind of traffic that comes towards

your applications.

So adding more instances of services is going to be super easy and this can be achieved automatically

with the help of container orchestration platforms like Kubernetes.

Moving on to the next principle, these cloud native applications, they follow **DevOps practices**, by

embracing all these DevOps principles.

They promote a collaboration culture between the development and operations teams.

Whenever these cloud native applications are being built, they will not be any blame game between the developers and operations team when they are building cloud native application because they are following these DevOps practices and with the help of these DevOps practices, they will incorporate continuous integration, continuous delivery and automated deployment pipelines to streamline the software development and deployment process.

So these cloud native applications, they will give complete flexibility to the organizations, whether

they want to do continuous integration only or whether they want to go with the continuous delivery

or if they want to go with the continuous deployment as well.

The next character of native applications is, they are **resilient and fault tolerance in nature**.

Whatever applications that we are going to develop with the help of cloud native principles, they can

withstand any kind of failures which will make them resilient and they will utilize techniques such

as distributed architecture, load balancing and automated fail recovery to ensure high availability

and fault tolerance.

One example that I can give here is, think like you have a microservice by following the cloud native

application principles, you will deploy this microservice in multiple locations.

Even if one of the location has a downtime due to some power outage or due to some internet issue,

the microservice will continue to work from the other location that you have deployed.

And at the same time, for some reason, if one of your microservice instance is not working platforms

like Kubernetes, they can automatically shut down that microservice instance and bring up a new instance.

This way you are ensuring the failure recovery automatically and bringing fault tolerance inside your

applications.

Moving on to the next character of cloud Native applications, which is they utilize **cloud native services** to a great extent.

Like I said, cloud native applications means they are developed to leverage the cloud environments

to a great extent.

When an organization or an application uses these cloud native services provided they don't have to

focus on infrastructure because all these services will be monitored and maintained by the cloud platform provider itself.

With that, the developers and the organizations they can simply focus more on the application logic and enhancing the business logic with a very less focus on the infrastructure components.

Like you can see these are all the important characters of cloud native applications. Inside this course, we are going to touch all of these characters in the various sections.

Whenever you see a application is following all these characters, then you can simply assume that is

an cloud native application.

Deep dive on 15 factor methodology – Part 1

Like you can see here, the very first principle are guideline that we have from the 15 factor methodology is one code base for one application.

So, this is going to be super, super simple.

So, let us try to understand details about it.

As per these guidelines, we need to make sure there is one to one correspondence between an application and its code base, which means each application are a microservice should have its own dedicated code base.

Inside this course we build three different microservice.

But in real world we need to make sure every microservice is having its own GitHub repo or its own code base inside the versioning system.

If there is any code which is common for all the microservice, then such common code should be manage separately as a library.

We can also deploy all these common code as a separate standalone service which will

serve as a backing service for the other applications.

Since it is possible to track each code base in its own code repository, this is going to provide a

lot of flexibility and brings organization of your code in a cleaner manner.

Once you have separate code base for all your microservice, please make sure, regardless of how many deployments you are doing in different, different environment, you should always make sure you are building and converting your code base into your Docker container or a Docker package or any other artifact only once.

As per these guidelines, it is unnecessary to rebuild the code base for each environment deployment

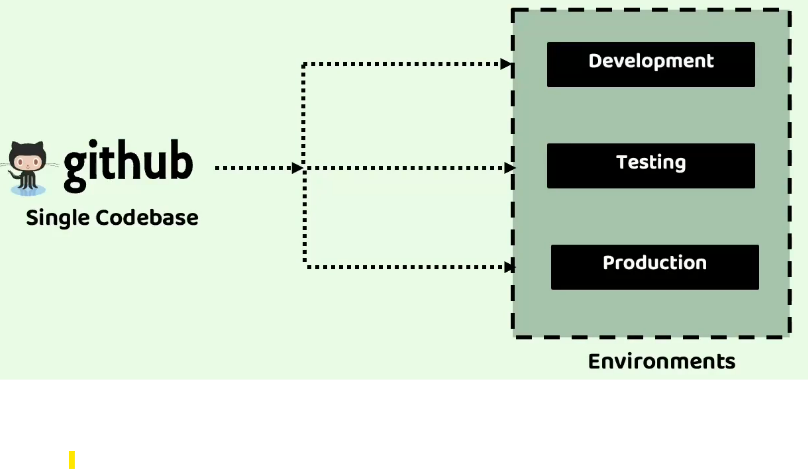
that you are going to make.

Here you may have a question like you have some configurations that may vary from environment to environment, like your database details or any other details which will vary from development to QA to production.

So, all such configurations as per this guideline, we need to maintain outside of your code base so

that they can be injected externally during the deployments into your application code base.

So, if you try to visualize this guideline, you can see there is a single code base for a single microservice application and the same code base must be packaged only once and once the packaging and the build is completed, the same packaged artifact or the Docker container or the Docker image should be deployed into the development testing and production.



Now moving on to the second guideline, which is always think like APIs first, like we saw inside microservices everything we are going to develop as REST APIs.

So even this cloud native guideline, which is API first, it is also recommend you to always adopt

and have a mindset of API first, which means right from design of your cloud native applications,

you should always think like to write as much logic as possible with the help of APIs.

Only if you design and develop your business logic with the help of APIs.

This will give flexibility that most of your business logic can be invoked by the other APIs or by the

other microservices as a backing service, and this also provide you an advantage like different, different teams. They can work on different, different APIs.

Apart from these advantages, when you follow these API first approach, it is also going to give flexibility like inside your deployment pipeline you can write some testable integrations with other systems, and if the integration and testing is working, then only you can do the deployment.

On top of that, since you are going to separate all your services with the help of APIs, you can always do some internal modifications behind your API implementation without impacting other applications or teams that rely on your APIs.

That is why please, please, while you are developing cloud native services or applications, always

have a mindset that API first.

The next guideline that we have here is dependency management principle.

As per these guidelines, it is important or crucial to explicitly declare all your dependencies of

an application inside a single manifest file and post that we should also ensure that these dependencies are accessible to a dependency manager which can download all of them from a central repository.

In case of java applications like we know we have two robust tools like Maven and Gradle that follow the same standard.

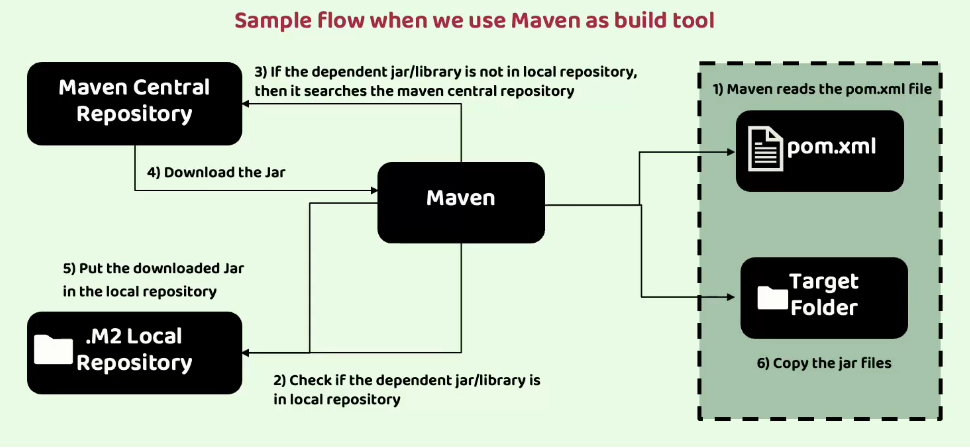
We as a developers, we only provide our dependencies inside the pom.xml or build.gradle and behind the scenes my build manager like maven and gradle.

It is going to pull all the dependencies from a centralized repository.

Once this download is completed during the packaging of our application as a Docker image or a Docker container, all the dependency libraries are going to package as a single artifact.

By following this approach, we maintain a clear and controlled dependency management process for our application.

That is why always follow this dependency management principle.



Let us try to visualize how this is going to work.

Like you can see here on the right-hand side, in the top there is a pom.xml.

**Step-1**: In the very first step, the developer defines all his dependencies inside the pom.xml and the build tool like Maven is going to read this pom.xml.

**Step-2:** You can see it is going to check if the dependent jars or libraries are present inside the local Maven repository.

**Step-3:** If they are not already downloaded and if they are not present in the local repository, then it is going to check if the same dependency jar is present inside the Maven central repository.

So, it is going to search in that.

**Step-4:** And if there is a library, the Maven is going to download the jar and put the same downloaded jar inside the local repository like you can see in the step five.

**Step-5:** And at last during the build process or during the Docker image generation process, all the dependent libraries will be clubbed as a single fat jar or a single docker image by following the step, which is copying all the required jars into the target folder.

This guideline name is design, build, release, run.

Inside these guidelines, what is the recommendation is your code base must progress from design to production by following the following stages.

The very first stage is design stage.

Inside this design stage, we need to determine all the required technologies and dependencies and tools for a specific application.

If you are building a microservice during the design stage itself, you need to determine all the technologies, dependencies and tools needed for your microservice.

So here inside these design stage, it also includes the development and unit testing.

So once the design, development and testing is completed next, we need to move on to the build stage.

The next stage is build stage.

Inside this build stage, we need to compile and package the code base with the required dependencies.

By creating an immutable artifact, every build artifact should have its own unique identification number, just like how we maintain versions like 1.02.03.0.

Very similarly, each of your build stage should have its own unique identification and at the same

time, whatever code base package you have generated, it should be immutable.

They should not be a scenario where someone is trying to change the content of the package code base

manually that is not recommended.

The next stage is release stage

After this build stage, is the release stage.

Inside this release stage, we need to combine the code base package with the deployment configurations based upon the environment.

Suppose if you are trying to release this build package into a production environment, you

should combine your code base with the production related configurations like database credentials,

folder structure, any server related properties or microservice related properties.

So, all those details you need to club and you need to prepare an immutable release component with

its own unique identifier, just like how we maintain inside versioning like 6.1.5.

So, inside a central repository you should store all your codebase artifacts along with the deployment

configurations so that in future, if you are looking for a rollback to a specific version, it is going

to make your life super, super easy.

The next stage is run stage

And at last, once this release stage is completed, our application along with the configuration,

is ready to be deployed and run as an microservice or as an application.

So that is why inside the last stage, which is run stage, we need to run the application in the designated environment using a specific release.

kind of behavior. So the reproducibility is going to be super, super easy if you follow these principle.

Now moving on to the next principle that we have, which is configuration credentials and code.

So, what is a configuration according to the 15-factor methodology?

Configuration are the elements that are prone to change between the deployments.

If there is a property that is going to differ from one environment to environment, we call all these

kind of properties or elements as configurations.

So as per these 15-factor methodology, we should never club these configurations with our code base.

We should have an ability to modify application configurations independently or without the need

to rebuild the application for every environment.

The examples of these configurations are like your database properties or your message system properties, credentials for accessing third party APIs are feature flags, so all these kinds of configurations, they are confidential in nature and at the same time they are going to be differ from one environment to other environment.

So, if you take your database username and password, it is a confidential and at the same time the database credentials for dev, environment and QA environment and production environment is not going to be same.

So, for all such type of configurations, we need to make sure we are maintaining all of them in a separate codebase and at the same time we need to make sure we are not exposing any sensitive information while maintaining all these configurations in a separate codebase.

If needed, we need to provide all the sensitive configurations by following some externalization standards which we are going to discuss inside this course.

In simple words, to follow this principle, your configuration should not be embedded within

the code or tracked in the same code base except for the default configurations.

So always the default configurations can be bundled within the application.

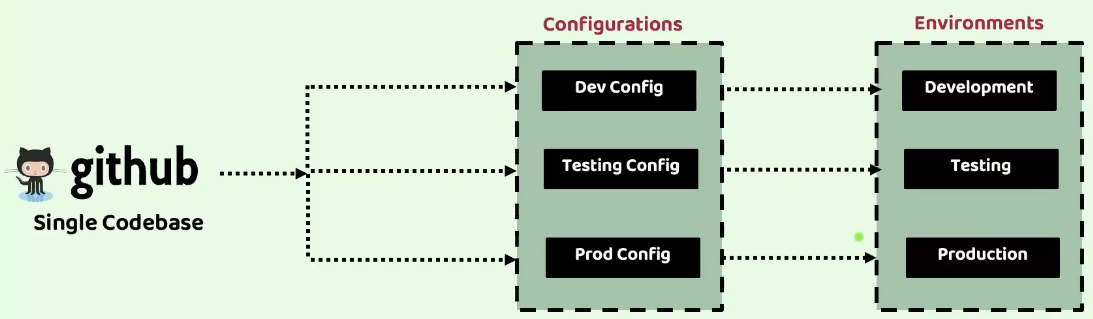
But anything that is going to change from environment to environment, we should not embed within

our code base.

If you try to embed it along with the code, then you need to generate the Docker image or the code

base package for every environment which is against these 15 factor methodologies.

Like you can see here, first there will be a single code base which is going to hold all the business logic, which is not going to change from environment to environment.



Using this single code base, during the build stage, we are going to generate a software package or a Docker image to this software package and Docker image while deploying to a specific environment like development, testing and production.

We need to make sure at runtime we are injecting the configurations related to the specific environment.

Suppose if I am trying to deploy my microservice into a production environment at runtime or at deployment stage, I need to provide the production related configurations.

This is one of the super important principles because since you are going to have hundreds of microservices and cloud native applications, it is always recommended to maintain the configurations related to them, which are going to change from environment to environment in a separate location.

In the spring ecosystem, there is a separate project with the name Spring Cloud config Server, which

is going to help us in implementing this guideline inside our microservice.

Deep dive on 15 factor methodology – Part 2

The next principle that we have inside the 15-factor methodology is **logs**.

Inside a traditional application are a monolithic application, how the logs are handled.

The application will write the logs into a file and folder location of your server.

So, if there are any issue comes inside your monolithic application, developers will go to that logs

location.

They will try to open the logs related to a specific date and timestamp and they will try to look at

the logs what happened, what is the issue, what is the exception?

But do you think this is going to work very similarly inside the microservice or cloud native applications?

Off course not, because you will have 100 different of microservice.

If there is any issue, are you going to look and search in all the 100 different servers or in all

the 100 different locations where your applications are going to write the logs?

That is not going to be a feasible option.

That is why as per this methodology, it is recommended that all the log routing and storage are not

the applications concern or cloud native applications are microservices they are not going to write the logs inside any log folder or inside any log location.

Instead, the application will simply redirect the logs to the standard output, treating them as a

sequentially ordered events based upon the time.

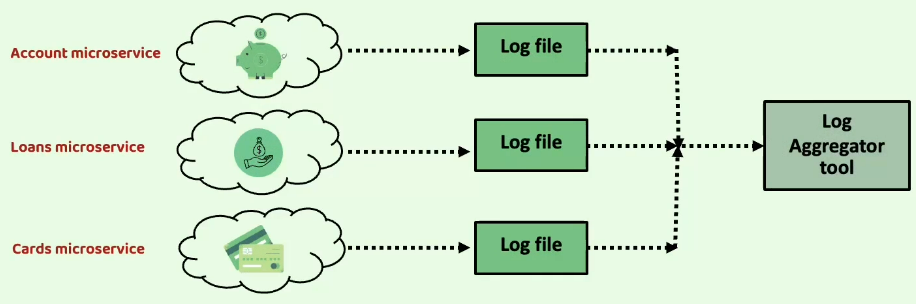
The responsibility of the log storage and the rotation should be shifted to an external tool called

log aggregator.

This tool retrieves, gathers and provides access to the logs for the debugging purpose.

So, if you see here, there is an accounts microservice, loans microservice and cards microservice.

They will simply print the logs to a standard output based upon the log statements and log framework that they use.



And behind the scenes, there will be a log aggregator tool that will keep on pulling for the logs,

and all these logs will be aggregated in a single location so that the developer or operations team,

they can search all the logs of all the microservice with the help of this log aggregator tool.

So, this is one of the guiding principles.

There is a separate section inside this course where with the help of a log aggregator tool, we will

feed all the logs from all the microservice into that tool and we are going to search all the logs of

all the microservice from a single UI log aggregator tool.

Now moving on to the next methodology, which is disposability inside a traditional monolithic application, making sure that a single monolithic application is always running is a top priority, and there is no room for this monolithic application to terminate or to get stopped.

However, inside a cloud environment or inside cloud native applications or inside microservices, it

is not going to super, super critical and necessary as well because you will have hundreds of microservices and they will be having multiple instances running.

It is not possible to manually monitor and making sure and all these instances and microservices

are running always.

That is why applications in the cloud are always considered as ephemeral, meaning that if a particular microservice or a cloud native application becomes unresponsive, it can be terminated and replaced with a new instance by platforms like Kubernetes automatically and at the same time, during high load periods or during high load traffic, additional instances of the applications can be spin up to handle the increased workload.

This concept of shutting down and creating new instances automatically is called application disposability, where the applications can be started or stopped as needed.

But to make our applications disposable and manage our applications in the dynamic environments

like cloud, it is crucial to design them for quick startup when new instances are required and for

graceful shutdown when they are no longer needed.

This fast startup enables systems elasticity and ensuring robustness and resilience.

Without these fast capabilities, performance and availability issues may arise like we discussed before, with the help of Spring boot framework and Docker containers, any microservice we can create and destroy within seconds.

Whereas with virtual machines and monolithic applications, it is going to take at least 10 to 15 minutes.

So that is why we are following Docker containers along with the useful framework like Spring Boot.

And whenever a graceful shutdown is being involved for an application, the application should be capable of not accepting any new request and at the same time any ongoing requests should be processed successfully and then only it should exit.

So, this process is going to be straightforward for a web application.

Now moving on to the next principle, which is backing services.

Your microservices may have dependency on many other external resources like database Smtp servers, FTP servers, caching systems, message brokers.

So, all these external resource dependencies, we call them as backing resources.

So, we should always treat these backing resources as attached resources so that we can modify or replace them without needing to make any changes inside your application code.

For example, consider the use case of a database throughout your software development life cycle.

Typically, different database is used in different stages, such as development, testing and production.

We should always treat these databases as an attached resource so that you can easily switch to a different service depending on the environment.

How this attachment is going to work through resource bonding, which involves providing necessary

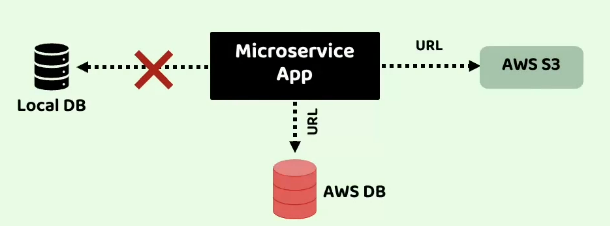
information like URL, username, and password of the database through externalized configurations.

So, once you generate a Docker image, you should not be generating the Docker image again and again whenever you want to change to your different database, instead to the same code base, you need to provide these URL information, user information and password information from an external configuration.

Like you can see here, there is an app which is using local database.

If needed, I can easily switch to an AWS database or any other external backing service by changing

these URL details and connection details.



Now let us go to the next principle that we have, which is environment parity.

As per this guiding principle, it is recommended to minimize the differences between various environments of your application and avoid any costly shortcuts.

If your environments they look very similar, then your application also is going to work in a very

similar manner.

Just to fix some issue, if you are making some changes inside an environment by using shortcuts, then the behaviour is going to be different for different, different environments.

That is why this environment parity recommends to make sure all your environments are looking exactly as much as possible.

When we use this environment parity, we are going to address three different types of gaps.

The very first gap is time gap.

Usually, the time it takes for a code change to be deployed can be significant, but this methodology

encourages automation like adopting CI/CD pipelines and perform continuous deployment to reduce the time between the code development and production deployment.

This will make sure always your environments are looking similar and it is going to make developers

life easy to perform any debugging.

The next gap it is going to address is people gap.

Usually developers, they create applications while platform operators they handle the deployment in production.

To bridge this gap, a DevOps culture should be followed.

And this DevOps culture, it promotes collaboration between developers and operators, fostering the

philosophy, which is you build it, you run it.

When the people gap is reduced, then there will be more coordination and collaboration, which will

help to make environments look very similar.

And with that, the number of issues are going to be less.

And even if an issue comes since the environment are looking same, debugging is going to be easy.

The last gap that this factor is going to address is tools gap.

Like we discussed previously, applications can always use backing services and sometimes they can differ from environment to environment.

For a microservice, a developer might use an H2 database locally, but in production they might be

using PostgreSQL.

This is not recommended because to achieve the environment parity we need to make sure we are using the same type of tool and the same version of the backing service across all environment.

The next factor at the principle that we have is administrative process.

Many times, we will be having many management tasks required to support application such as database migration, any batch jobs to clean the data or to update the data.

So, all these maintenance tasks and management tasks, they should be treated as isolated process.

Like application process, the code for these administrative or management tasks should be version controlled and packaged along with the application and executed within the same environment.

Many times, developers, they will try to skip these administrative tasks running in the local dev or QA.

For example, they do not run these batch jobs or data migrations, any maintenance tasks just to save

some time and so that they can deploy the code into production quickly.

But that will sometimes bring surprising results in production.

That is why we should make sure these management and administrative processes are also equally important and they are treated as a separate isolated process and they are also properly tracked using versioning system and packaged along with the application and deployed in each environment where the application is being deployed.

It is always advisable to consider these administrative tasks as independent microservices where they are executed only once and when they are not needed.

We can discard them, if you try to put this administrative process inside your business logic, then you are unnecessarily carrying all these administrative tasks inside your microservice.

Instead, if you keep your microservice business logic and these administrative tasks separately, once

you are done with these administrative tasks; you can discard them so that your microservice alone

can run continuously and serve its clients.

Alternatively, these administrative tasks, they can also be integrated directly into the application,

which can be activated by calling a designated endpoint.

But it is always better to deploy them as an independent microservice.

Deep dive on 15 factor methodology – Part 3

Like you can see here, the next guideline that we have is port binding.

As per this guideline, all the cloud native applications should be self-contained and expose their

services through port binding.

When I say self-contained, an application should not rely on an external server within the execution

environment.

For instance, a traditional Java web application might typically run within a server container like

Tomcat, Jetty or Undertow.

We as a developer should manually deploy our Java code into these servers.

In contrast to this, a cloud native application does not depend on the presence of a Tomcat server

in the environment.

It manages a server as a dependency within itself.

We are already using these inside our microservices, for example using spring boot, we are enabling

the usage of an embedded server where an application incorporates the server within itself instead of relying on its availability in the execution environment.

Once the application is started using these self-contained servers, each application is going to map

to its own server compared to traditional approach where deploying multiple applications in a single

server.

In simple words, we should not deploy multiple applications in a single server.

Instead, every application should be deployed in a separate server and even that server also should

be self-contained but not external to the application.

Once the application is started with its own self-contained server, the applications should expose

its services to the outside world through port binding.

When we are trying to start our Docker image as a Docker container with the help of docker run command, we use the port forwarding or port mapping.

So, using that port binding, we are exposing the microservice to the external network.

Once this application is exposed at a specific port, then any other service or any other clients they

can invoke my microservice and these microservices are cloud native application is going to act as a

backing service for another application.

So, this is very common practice within cloud native systems because many applications are many microservices they want to communicate with each other.

In such scenarios we should make sure we are properly exposing our microservice by following these self-contained and port binding standards.

Never ever develop cloud native applications, which depends on an external Tomcat server or an external server. With such scenarios, you will end up with managing these external servers manually in all the locations where your cloud native applications and microservices are being deployed.

The next principle that we have here is stateless process.

We know that the cloud native applications are the microservices are developed with high scalability

in mind.

One of these key principles to achieve scalability is to design our applications as stateless process

and adopting a shared nothing architecture.

Like you can see here, there is a single microservice where it has multiple instances running.

To make it simpler.

Let us try to assume you have your accounts microservice.

If it is receiving a lot of traffic, what you will do, you will scale your accounts microservice by

onboarding multiple new instances of same accounts microservice. So, all these multiple instances of accounts, microservice they should follow stateless process and they should not share anything between themselves.

This is important because when the traffic is low or when an instance is not working properly, we are

going to destroy it or we are going to recreate it.

If the instance of a microservice is going to hold some data, there will be some data loss would occur

and the business logic will be impacted.

That is why our application should be strictly stateless.

However, sometimes there might be a requirement where our accounts microservice or a specific cloud native service, it has to store some data or it has to store some user data.

So, for all such scenarios, these instances are accounts microservice or cloud native service,

they should use a backing service like a data store, for example you can use a database or you can use a Redis cache to store all the caching related information.

This will remain your application stateless and whenever they want to store something, they can store inside the database or any other data store.

Even a particular instance of accounts microservice is shut down and in future, after few minutes

if a new instance Is coming up, there is no loss of information.

Everything is stored inside a data store from where my new instance can read and execute the business logic.

So that is why always make sure you are not storing anything inside your instance, such as user session related information or caching information, because you are going to lose all that information as soon as a particular instance is shut down.

That is why please make sure always every information is being stored inside the storage system and making your instances and cloud native applications as true stateless applications.

Now let us move on to the next principle that we have, which is concurrency.

We discussed that by implementing stateless applications we can achieve the scalability.

But scalability cannot be achieved solely by creating stateless applications.

While statelessness is important, scalability also requires the ability to serve large number of users.

This means your applications should support concurrent processing to handle multiple users simultaneously.

This means if you have ten different microservice instances are running inside your microservice network, they should not accept only ten different requests and a process sequentially one by one.

Instead, they should be capable of processing lot of requests parallelly simultaneously to achieve

this, according to the 15-factor methodology process play a very crucial role in application design.

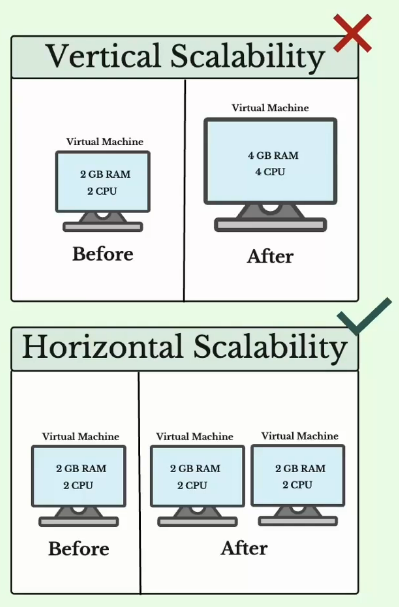
Whenever we are getting a greater number of traffic, which means we need a greater number of processes to handle such traffic.

So, in such scenarios we can horizontally scale our processes to distribute the workload across multiple processors on different machines.

Inside Java we already have this concurrency inbuilt developed when we built our applications with the help of Java and JVM, concurrency is typically managed by the program itself with the help of multiple threads which are available from the thread pools.

And whenever you are trying to scale your application, we should never follow the vertical scalability

instead, we should follow the horizontal scalability.



You can see vertical scalability means we are going to increase the Ram memory and CPU processes for a mission.

To some extent, this vertical scalability may work, but once you reach to a maximum CPU processor

and maximum Ram, then you cannot scale your applications vertically.

That is why we should always follow the horizontal scalability inside the horizontal scalability.

We are going to create multiple virtual machines with the same configurations like two GB Ram and two CPU. And inside these virtual machines we can create containers or we can create processes based upon our requirements.

And there is no limit for our horizontal scalability.

You can create any number of virtual machines and you can deploy any number of containers inside them.

And usually, these processes which handles the traffic of the application, they are categorized based

upon their respective types.

For example, all the web processes responsible for handling the Http requests.

And just like web processes, we also have worker processes that execute some scheduled background jobs.

So, by classifying all these processes and optimizing their concurrency, applications can effectively

scale and handle increased workloads without having this concurrency ability inside your application

and inside your programming language that you are going to use, your cloud Native applications or microservices, they are not going to be easily scaled.

Now let us move on to the next principle or guideline that we have, which is telemetry.

Inside monolithic application we will have very limited number of applications, like 1 or 2 monitoring them, understanding their logs, metrics, performance related information is very easy because you have to monitor only one server or one application or two applications or two servers.

Whereas with cloud native applications or microservices, you will be having multiple containers running, multiple services running, multiple servers will be running inside your organization.

So how are you going to monitor them. For the same we have a concept called observability.

So, this observability is a fundamental characteristic of cloud native applications.

Since we are going to have multiple applications inside the cloud it becomes essential to have access

to the accurate and comprehensive data from each component of the system.

If you have 100 different microservices, you should be able to access the accurate information and

comprehensive information about all these microservices in a single place. So that it can enable remote monitoring of the systems behaviour and facilitate effective management, what kind of data we need for this effective management.

Telemetry data such as logs, metrics, traces, health status and events.

All this plays a very vital role in providing this visibility into your cloud Native applications.

If you try to understand how this telemetry is derived in Kevin Hoffman's analogy, he emphasizes the

significance of telemetry by comparing with the applications to the space probes.

So, he is trying to compare our cloud native applications or microservices to the space probes.

Space probes are nothing but the satellites are the space rockets that we send into the space for

research. Space organizations like NASA, Isro.

They will use this telemetry data of the space probe to monitor and control remotely.

The same concept applies to the cloud native applications as well. To effectively monitor and control

applications remotely, we need various types of telemetry data, and these telemetry data are like

detailed logs for troubleshooting, metrics to measure performance, traces to understand request flows, health status to access system well-being and events to capture significant occurrences.

By gathering and utilizing all these type of telemetry data, we can gain valuable insights about our applications and microservices and make informed decisions to manage them effectively from a remote location.

So, what this guideline is recommending is please make sure your applications are feeding all this telemetry information to a centralized component.

And from that centralized component, we should be able to monitor and control their behaviour.

Now, moving on to the last guiding principle, which is authentication and authorization.

So, security is a very critical aspect of any software system.

But many times, we see that this security is not receiving necessary emphasis it deserves.

So, what this guiding principle is saying is, we need to follow a zero-trust approach and we need to make sure every communication and every interaction within the system and within the microservice network or cloud native systems is happening by following the security standards.

When you talk about security, there are many things involved apart from authentication and authorization, which is the responsibility of the developer.

Apart from these authentication and authorization platforms team, they can follow Https protocol,

they can have some SSL certificates; they can have some firewall protection.

So, all these standards also our operations team, they are going to follow.

But from the development perspective, following these authentication and authorization is very important.

So, what is an authentication, authentication enable us to track and identify who is the user trying to

access our application. Once the authentication is completed, like with the help of username and password, we can then proceed to evaluating their permissions and determine if they have necessary authorization to perform a specific action.

So inside authentication will only check the identity of the end user, whereas inside the authorization

that is going to happen after the authentication, we are going to validate if a specific end user has

enough privileges to perform a specific action inside a application.

So, implementing these authentication and authorization is very important for cloud native applications and microservices.