Cloud-Native Apps and 15-factor methodology

# Cloud-Native Technologies and Applications

* A good resource: <https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/definition>

## According to the cloud native computing foundation:

Cloud-native technologies empower organizations to build and run **scalable** applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

These techniques enable loosely coupled systems that are **resilient**, **manageable**, and **observable**. **Combined with robust automation**, they allow engineers to make high-impact changes frequently and predictably with minimal toil.

Cloud-native applications are designed to leverage cloud computing principles and take full advantage of cloud-native technologies. They are designed to run on cloud environments, utilizing the scalability, elasticity and flexibility offered by the cloud.

## Characteristics of Cloud-Native applications

### Microservices

They are usually built with the microservices architecture where the application is broken to smaller, loosely coupled services that can be independently developed, deployed and scaled.

### Containers

They’re usually packaged and deployed as containers which provide a light-weight and consistent environment for running applications making them highly portable across different cloud platforms.

### Scalability & Elasticity

They’re designed to scale horizontally allowing them to handle increased loads by adding more instances of the services

They can also automatically scale up and down based on the demand thanks to cloud-native orchestration platforms like Kubernetes.

### Automation

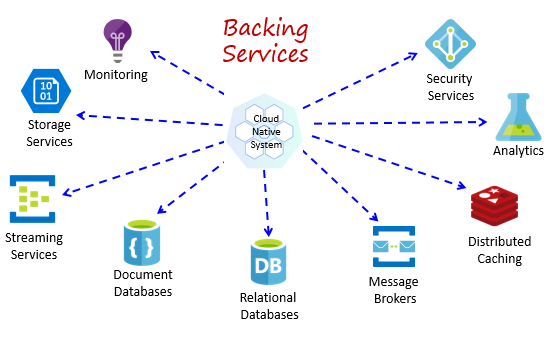
They embrace Dev-ops practices and incorporate CI/CD pipelines to streamline development and deployment process

### Resilience and Fault Tolerance

Utilizing techniques like distributed architecture, load balancing and automated failure recovery to ensure high availability and fault tolerance

### Backing Services

They take advantage of many resources like:



# Differences Between Cloud-Native and Traditional Enterprise Apps

Read this section with a grain of salt.

## Tracking Issues

Tracking issues in microservices is more straightforward than a monolithic application where you have the entire code base to find the bug.

## Dependency on the Environment

Cloud-native apps are not dependent on the environment and it makes the development env close the actual production env.

## Size and dependency

Cloud-native microservices are loosely coupled and rather independent from each other while having the size needed for that specific service. But in a monolithic application, different parts of the app are probably more dependent on each other while having a bulk-sized application carrying all the services.

## Continuous Delivery

The changes in microservices can be easily deployed as each microservice can be independently deployed.

## Recovery and Scalability

In case of containers, they can be easily and rapidly scale up and down and turned off and on and restarted but it’s not the case with monolithic apps.

# Development Principles of Cloud-Native Applications: 12-factor and 15-factor methodology

The 15-factor methodology is a revised and extended version of the 12-factor methodology that was introduced earlier and is apparently the main reference nowadays.

## One Code Base, One Application

There is only one code base(repository) per app(microservice). This implies 2 features:

* Shared codes should be packaged into libraries to be used across different microservices or into a separate service acting as a backing service for other microservices.
* One code base should be deployed across different environments like QA and production and configurations should be maintained outside the application code base and injected when deploying the application.

**A deploy in this context is a running instance of the same code base. There can be multiple deploys of a code base across different environments.**

## + API First

Make everything a service. Assume your code will be consumed by a front-end client, gateway, or another service.

Having a clearly defined and I add non-changing API will help rely on your service through the public API and implementation details can change without the users even noticing it.

Having these APIs is critical for performing integration tests.

## Dependencies

**A twelve-factor app never relies on implicit existence of system-wide packages.** It declares all dependencies, completely and exactly, via a **dependency declaration manifest** (POM file in our case).

Each microservice isolates and packages its own dependencies. And other people will **only** need the **dependency manager** and the **language run time** to run the application.

## Build, Release, Run

A [codebase](https://12factor.net/codebase) is transformed into a (non-development) **deploy** through three stages:

* The build stage is a transform which converts a code repo into an executable bundle known as a build. Using a version of the code at a commit specified by the deployment process, the build stage fetches vendors [dependencies](https://12factor.net/dependencies) and compiles binaries and assets.
* The release stage takes the build produced by the build stage and combines it with **the deploy’s current**[**config**](https://12factor.net/config). **The resulting release contains both the build and the config and is ready for immediate execution** in the execution environment.
* The run stage (also known as “runtime”) runs the app in the execution environment, by launching some set of the app’s [processes](https://12factor.net/processes) against a selected release.
* **The twelve-factor app uses strict separation between the build, release, and run stages.** For example, it is impossible to make changes to the code at runtime, since there is no way to propagate those changes back to the build stage.
* Every release should always have a unique release ID, such as a timestamp of the release (such as 2011-04-06-20:32:17) or an incrementing number (such as v100). The releases must support the ability to roll back. Releases are an append-only ledger and **a release cannot be mutated once it is created. Any change must create a new release.**

## Configuration

**An app’s config is everything that is likely to vary between**[**deploys**](https://12factor.net/codebase) (staging, production, developer environments, etc.). This includes

* Resource handles to the database, Memcached, and other [backing services](https://12factor.net/backing-services)
* Credentials to external services such as Amazon S3 or Twitter
* Per-deploy values such as the canonical hostname for the deploy?

Configuration information is moved out of the microservice. Storing configs as constants in the code base violates this principle.

A litmus test for whether an app has all config correctly factored out of the code is whether the codebase could be made open source at any moment, without compromising any credentials.

**The twelve-factor app stores config in environment variables.** Env vars are easy to change between deploys without changing any code; unlike config files, there is little chance of them being checked into the code repo accidentally; and unlike custom config files, or other config mechanisms such as Java System Properties, they are a language- and OS-agnostic standard.

## Logs

In Cloud-native applications, routing and storage of logs are not of concern of the application. They direct the logs to standard out put and storing logs for inspection is the job of an external tool called log aggregator.

## Disposability

**Disposability in 15-factor apps means they can be started or stopped at a moment’s notice.** This facilitates **fast elastic scaling**, **rapid deployment** of [code](https://12factor.net/codebase) or [config](https://12factor.net/config) changes, and robustness of production deploys. In case a service is not responding that instance can be shut down and a new instance replaces it. In increased loads new instances can be added to respond to the new load. So, the services should be disposable with minimum time.

When the instances are no longer needed, they should be gracefully shut down by sending a SIGTERM to the process. This for web processes means that they should cease listening on their port and complete the ongoing requests. Implicit in this model is that HTTP requests are short (no more than a few seconds), or in the case of long polling, the client should seamlessly attempt to reconnect when the connection is lost.

For worker processes however this involves returning any pending jobs to the work queue before being shut down.

Using Docker containers along with Kubernetes clusters will guarantee this principle.

## Backing Services

A backing service is any service the app consumes over the network as part of its normal operation. Examples include datastores (such as [MySQL](http://dev.mysql.com/) or [CouchDB](http://couchdb.apache.org/)), messaging/queueing systems (such as [RabbitMQ](http://www.rabbitmq.com/) or [Beanstalkd](https://beanstalkd.github.io/)), SMTP services for outbound email (such as [Postfix](http://www.postfix.org/)),Restful Services, FTP servers, and caching systems (such as [Memcached](http://memcached.org/)).

**The code for a fifteen-factor app makes no distinction between local and third-party services.** To the app, both are attached resources, accessed via a URL or other locator/credentials stored in the [config](https://12factor.net/config). A [deploy](https://12factor.net/codebase) of the fifteen-factor app should be able to swap out a local MySQL database with one managed by a third party (such as [Amazon RDS](http://aws.amazon.com/rds/)) without any changes to the app’s code. Likewise, a local SMTP server could be swapped with a third-party SMTP service (such as Postmark) without code changes. In both cases, only the resource handle in the config needs to change.

## Dev/Prod Parity (env parity)

It means you have to keep the dev/staging/prod environments as similar as possible.

Historically, there have been substantial gaps between development (a developer making live edits to a local [deploy](https://12factor.net/codebase) of the app) and production (a running deploy of the app accessed by end users). These gaps manifest in three areas:

* **The time gap**: A developer may work on code that takes days, weeks, or even months to go into production. The gap between code changes and their deployment.
* **The personnel gap**: Developers write code, ops engineers deploy it.
* **The tools gap**: Developers may be using a stack like Nginx, SQLite, and OS X, while the production deploy uses Apache, MySQL, and Linux.

**The twelve-factor app is designed for**[**continuous deployment**](http://avc.com/2011/02/continuous-deployment/)**by keeping the gap between development and production small.** Looking at the three gaps described above:

* Make the time gap small: a developer may write code and have it deployed hours or even just minutes later (using CI/CD pipelines?).
* Make the personnel gap small: developers who wrote code are closely involved in deploying it and watching its behavior in production (again using CI/CD pipelines?).
* Make the tools gap small: keep development and production as similar as possible.

## Administrative Processes

Run admin/management tasks as one-off processes

The Admin Processes factor advocates for treating administrative or management tasks(like database migration) as separate, one-off processes rather than bundling them with the main application. They should be version controlled and packaged alongside the application and executed within the same environment.

It’s advised to package the admin processes into independent microservices that are used once and thrown out. Or as functions in a stateless platform which respond to specific events.