## Definitions

**Microservices** - [software architecture style](https://en.wikipedia.org/wiki/Architectural_pattern) in which complex [applications](https://en.wikipedia.org/wiki/Software_application) are composed of small, independent [processes](https://en.wikipedia.org/wiki/Process_%28computing%29) communicating with each other using [language-agnostic](https://en.wikipedia.org/wiki/Language-independent_specification) [APIs](https://en.wikipedia.org/wiki/Application_programming_interface).[[1]](https://en.wikipedia.org/wiki/Microservices#cite_note-martinfowler-1) These [services](https://en.wikipedia.org/wiki/Service_%28systems_architecture%29) are small, highly [decoupled](https://en.wikipedia.org/wiki/Coupling_%28computer_programming%29) and focus on doing a small task,[[2]](https://en.wikipedia.org/wiki/Microservices#cite_note-2)[[3]](https://en.wikipedia.org/wiki/Microservices#cite_note-3)[[4]](https://en.wikipedia.org/wiki/Microservices#cite_note-4) facilitating a [modular](https://en.wikipedia.org/wiki/Modularity) approach to [system](https://en.wikipedia.org/wiki/System)-building.[[5]](https://en.wikipedia.org/wiki/Microservices#cite_note-5) Services are organized around capabilities, e.g., [user interface](https://en.wikipedia.org/wiki/User_interface) front-end, recommendation, logistics, billing, etc.   
<https://en.wikipedia.org/wiki/Microservices>  
http://martinfowler.com/articles/microservices.html  
<https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing>  
<https://rclayton.silvrback.com/failing-at-microservices>

• An application built around the database—dependent on the database—will succeed or fail based on how it uses the database. As a corollary to this—all applications are built around databases; I can’t think of a single useful application that doesn’t store data persistently somewhere.

• Applications come, applications go. The *data*, however, lives forever. It is not about building applications; it really is about the data underneath these applications.

• A development team needs at its heart a core of database-savvy coders who are responsible for ensuring the database logic is sound and the system is built to perform from day one. Tuning after the fact—tuning after deployment—means you did not build it that way.

Thomas Kyte - Expert Oracle Database Architecture, Oracle Database 9*i*, 10*g*, and 11*g -*Programming Techniques and Solutions, Second Edition

## Why not Docker?

<http://sirupsen.com/production-docker>  
<https://valdhaus.co/writings/docker-misconceptions/>

## Docker for local development

For local development, you are using containers as a host for your application, but optimizing for developer productivity and debugging. For example, you typically volume mount your local source code, turn on full tracing or debugging features, and include un-minified JavaScript to simplify debugging.

## Docker for Production Validation

In contrast to local development, for production validation, developers are building a container with the fully optimized code and disabling development only features. Instead of volume mounting the source code, production images more typically include Dockerfile commands to copy any source code directly into the image. This helps developers test that the same exact image they built in development will work exactly the same way in staging and production.

## Docker as a Build/Test host

Some development teams use Docker containers not as a host for their application in production, but rather as an on-demand isolated environment for compiling and testing. A developer can easily spin up a build inside a container, specify what tests to run and run it in a container instead of on their local box. One reason this is handy is if a developer needs to install and validate different dependencies like shared libraries or pre-release software that you may not want to install on your local box. After the build compiles and unit tests pass, the container is then discarded. A more common use case for build and test validation with Docker is not on the local machine, but rather as part of a continuous integration workflow, which we’ll discuss in [Chapter 6](https://www.safaribooksonline.com/library/view/microservices-with-docker/9780134218229/ch06.html#ch06).

## Local Development

In this configuration, developers have a virtualized operating system like VirtualBox or Hyper-V that they develop and test Docker containers on. Developers can run multiple containers, like other microservices or application infrastructure like databases or caching services, all from the comfort of their local dev environment. Developers can also use Docker’s local volume mounting (covered later in the chapter) for even faster local development.

## Local and Cloud

In this configuration, you get all of the benefits of the local environment discussed above, but you also develop and deploy against a shared dev environment. This enables you to test your microservice using orchestration tools like Mesos or Kubernetes hosted across multiple Docker hosts. The difference here is that instead of everything working on a single host, this helps you test for service latency, cluster configuration issues and in general a more real-world environment that mimics production.

## Cloud only

In this configuration, development teams have two options. They can develop and test natively on the machine without using Docker at all, and instead push to a cloud environment as part of their CI process. Or they can develop directly against a “local” cloud environment, meaning a lightweight virtual machine assigned to each developer. Like the other scenarios, you still have a shared dev environment to integrate microservices. Developers sometimes need to choose this because their local PCs aren’t capable of running virtualization or a corporate policy doesn’t allow them to run virtualization on their PCs. To work around this, development teams provision a bare-bones virtual machine per developer in the cloud for development and test. This allows developers to have a “local” environment in the cloud they can develop with before pushing to a shared development environment. Developers with a Microsoft MSDN subscription can use their Azure benefit which provides a free credit of $50-$150 a month that can be used to cover the cost of each developer’s virtual machine.

Let’s review these tags as you’ll find many of them on other Docker Hub images:

• **latest** – The latest version of the image. If you do not specify a tag when pulling an image, this is the default.

• **slim –** A minimalist image that doesn’t include common packages or utilities that are included in the default image. The only reason to use this is if size is a constraint as the slim image is often times half as small as a full image.

• **jessie/wheezy/sid/stretch**– These tags represent codenames for either versions or branches of the Debian operating system, named after characters from the Toy Story movie. **jessie** and **wheezy** represent Debian OS versions 6.0 and 7.0 respectively, while **sid** is the codename for the unstable trunk and **stretch** is the codename for the testing branch.

• **precise/trusty/vivid/wily** – These tags represent codenames for versions 12.04 LTS, 14.04 LTS, 15.04, and 15.10 of the Ubuntu operating system.

• **onbuild** – The onbuild tag represents a version of the image that includes onbuild dockerfile commands which are designed to be used as a base image for dev and test. In normal dockerfile commands, the execution happens when the image is created. Onbuild dockerfile commands are different in that execution is deferred and instead executed with the downstream build.

## Notes

Installing docker on Oracle Linux is same as for CentOS. But before installing it, check the yum repo file:

/etc/yum.repos.d/public-yum-ol7.repo

By default, on Oracle Linux, group [ol7\_addons] is disabled.