Introduction

Students, write your response!

Pear Deck Interactive Slide
Do not remove this bar

Table of Contents

1

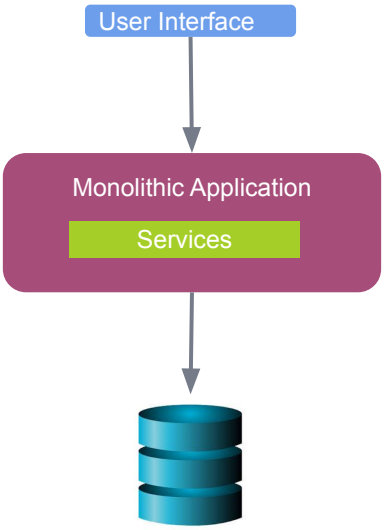
Monolith vs Microservices

Monolith vs Microservices

The word 'monolith' means 'one massive stone'. So we can describe monolithic as a large unified block.

Monolith vs Microservices

In software development, **monolithic architecture** is a traditional way to build an application as a single and indivisible unit.



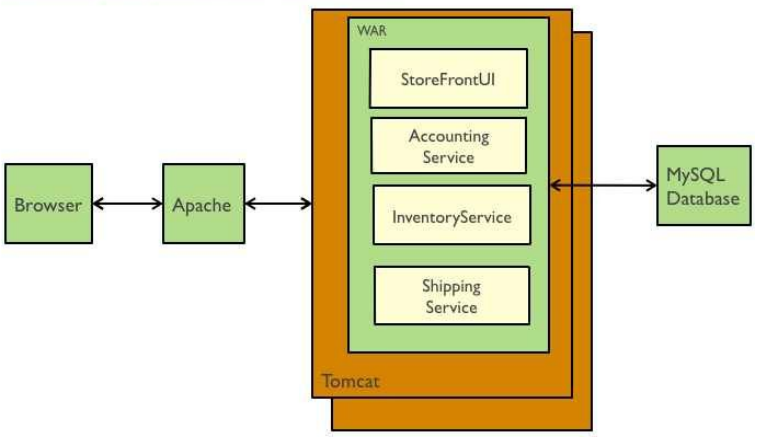
Monolith vs Microservices

Let's imagine that we are building an e-commerce application that takes orders from customers, verifies inventory and available credit, and ships them.

The application consists of several components including the StoreFrontUI, which implements the user interface, along with some backend services for checking credit, maintaining inventory and shipping orders.

Monolith vs Microservices

The application is deployed as a single monolithic application. For example, a Java web application consists of a single WAR file that runs on a web container such as Tomcat.



Monolith vs Microservices

Pros of monolithic architecture:

- **Easier to develop.** As long as the monolithic approach is a standard way of building applications, any engineering team has the right knowledge and capabilities to develop a monolithic application.
- **Easier to deploy.** You need to deploy your application only once instead of performing multiple deployments of different files.
- **Easier to test and debug.** Since a monolithic app is a single indivisible unit, you can run end-to-end testing much faster.

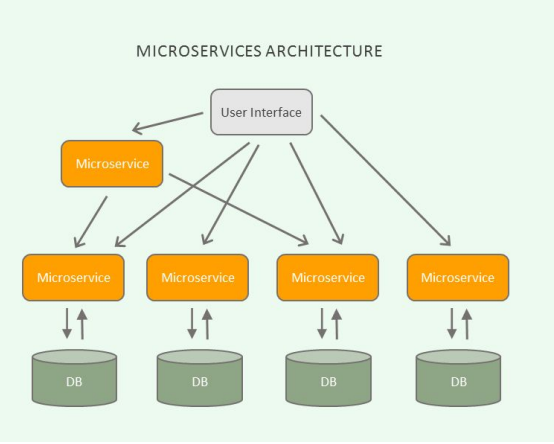
Monolith vs Microservices

Cons of monolithic architecture:

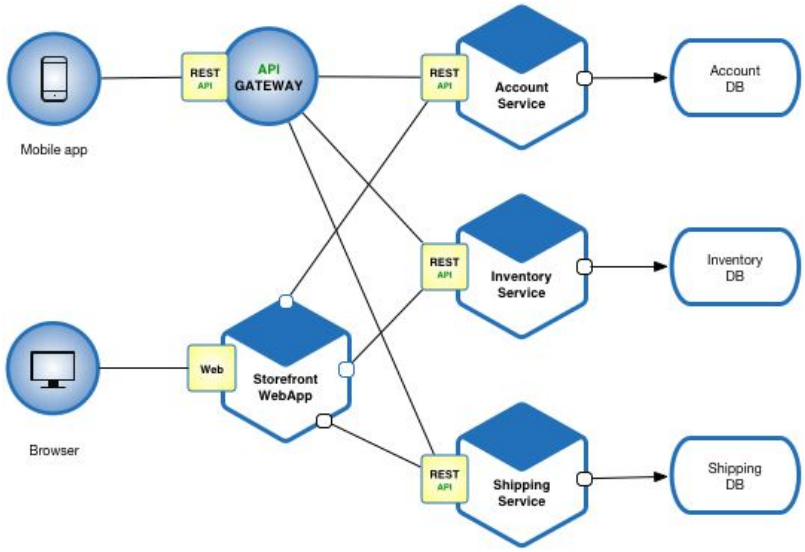
- **Understanding.** When a monolithic application scales up, it becomes too complicated to understand.
- **Making changes.** Any code change affects the whole system so it has to be thoroughly coordinated.
- **Scalability.** You cannot scale components independently, only the whole application.
- **New technology barriers.** It is extremely problematic to apply a new technology in a monolithic application because then the entire application has to be rewritten.

Monolith vs Microservices

While a monolithic application is a single unified unit, a **microservices architecture** breaks it down into a collection of smaller independent units.



Monolith vs Microservices



Monolith vs Microservices

Pros of microservices:

- **Independent components.**
  - All the services can be deployed and updated independently, which gives more flexibility.
  - A bug in one microservice has an impact only on a particular service and does not influence the entire application.
  - It is much easier to add new features to a microservice application than a monolithic one.

Monolith vs Microservices

Pros of microservices:

- **Easier understanding.** Split up into smaller and simpler components, a microservice application is easier to understand and manage.
- **Better scalability.** Each element can be scaled independently. So the entire process is more cost- and time-effective than with monoliths when the whole application has to be scaled even if there is no need in it.

Monolith vs Microservices

Pros of microservices:

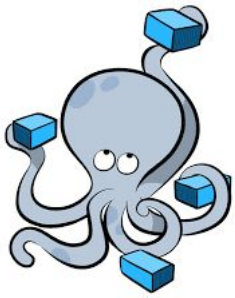
- **Flexibility in choosing the technology.** The engineering teams are not limited by the technology chosen from the start. They are free to apply various technologies and frameworks for each microservice.
- **The higher level of agility.** Any fault in a microservices application affects only a particular service and not the whole solution. So all the changes and experiments are implemented with lower risks and fewer errors.

Monolith vs Microservices

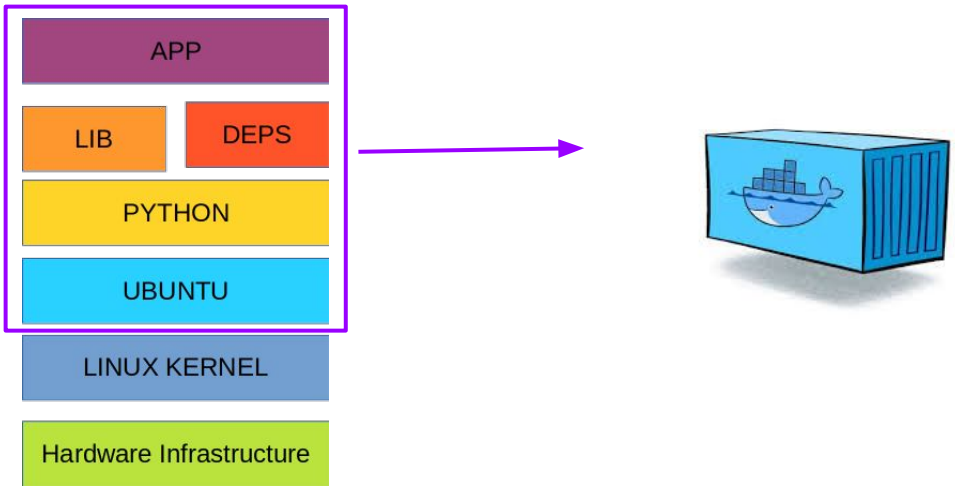
Cons of microservices:

- **Extra complexity.** Since a microservices architecture is a distributed system, you have to choose and set up the connections between all the modules and databases.
- **System distribution.** A microservices architecture is a complex system of multiple modules and databases so all the connections have to be handled carefully.
- **Testing.** A multitude of independently deployable components makes testing a microservices-based solution much harder.

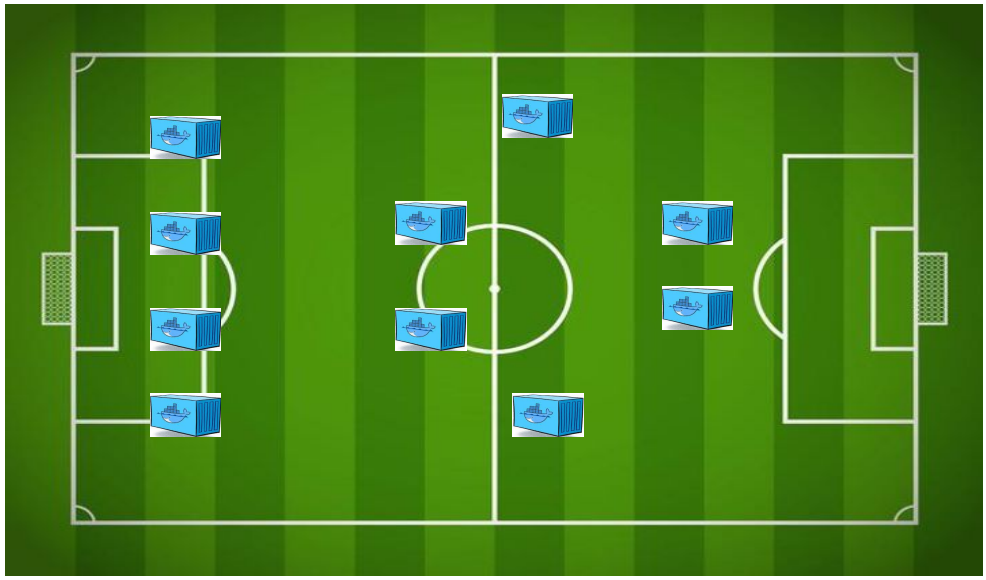
2 Orchestration



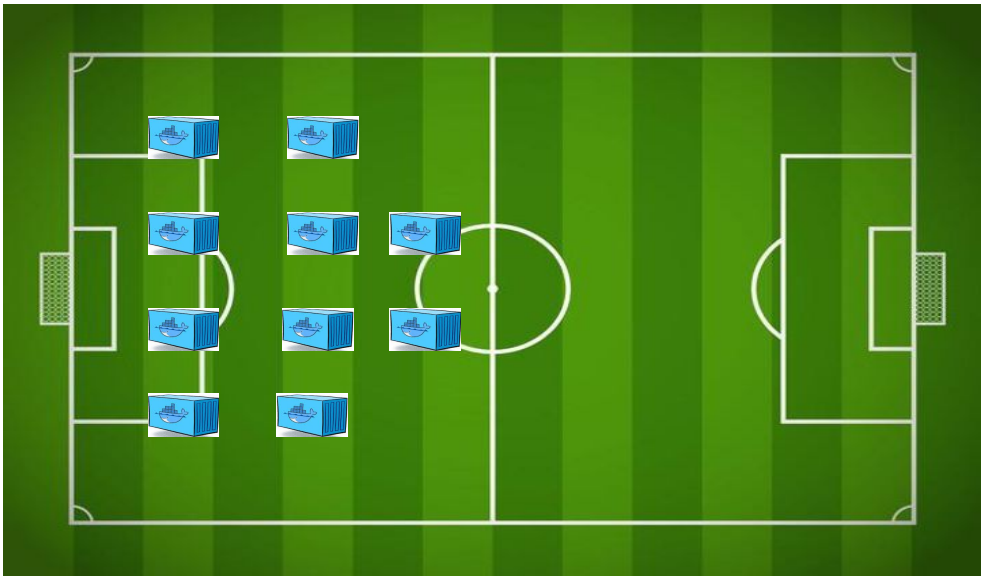
Orchestration



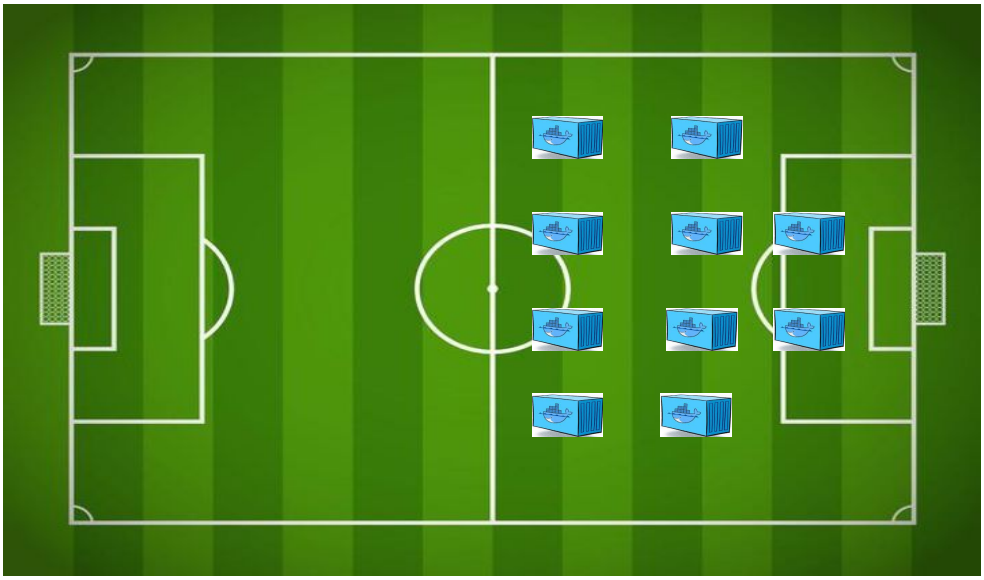
Orchestration



Orchestration

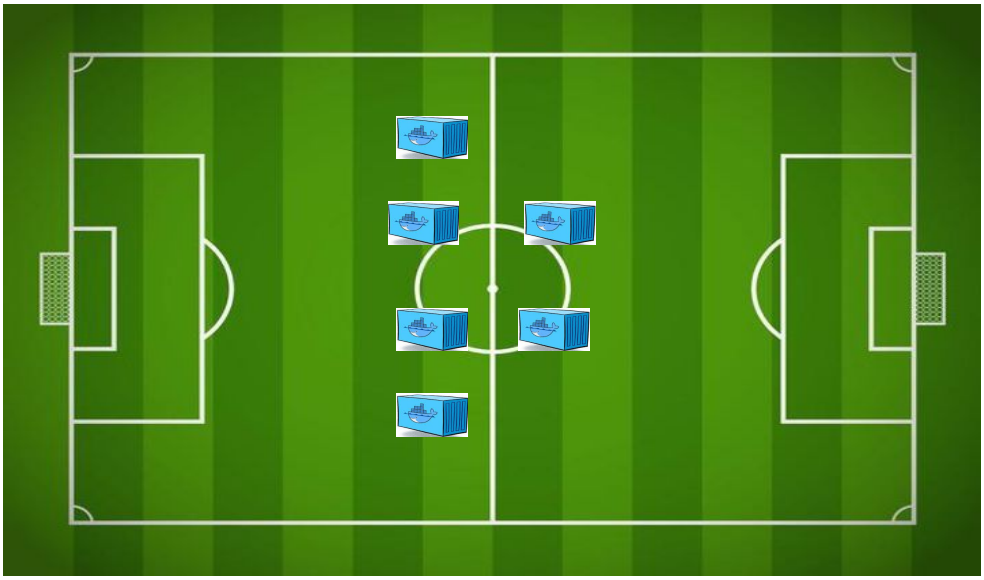


Orchestration

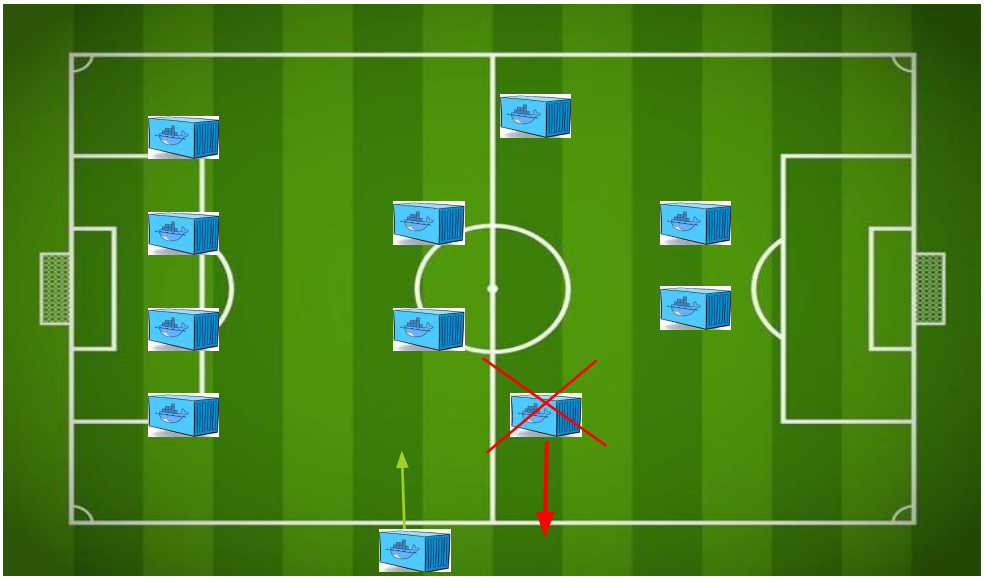


Orchestration

Orchestration

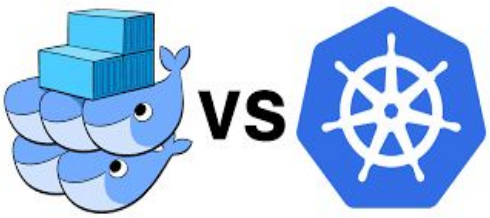


Orchestration



Orchestration

- Containers are great, but when you get lots of them running, at some point, you need them all working together in harmony to solve business problems.
- Tools to manage, scale, and maintain containerized applications are called orchestrators, and the most common examples of these are **Kubernetes** and **Docker Swarm**.



Orchestration

Container orchestration is used to automate the following tasks at scale:

- Provisioning and deployments of containers
- Availability of containers
- Load balancing, traffic routing and service discovery of containers

Orchestration

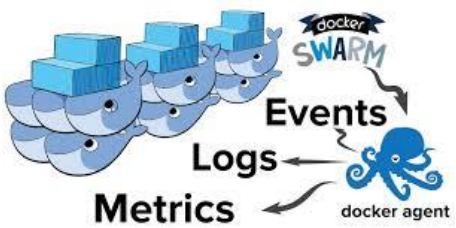
- Health monitoring of containers
- Securing the interactions between containers.
- Configuring and scheduling of containers
- Allocation of resources between containers

Container Orchestration Work

- Container orchestration works with tools like Kubernetes and Docker Swarm.
- Container orchestration tools work in any environment that runs containers.

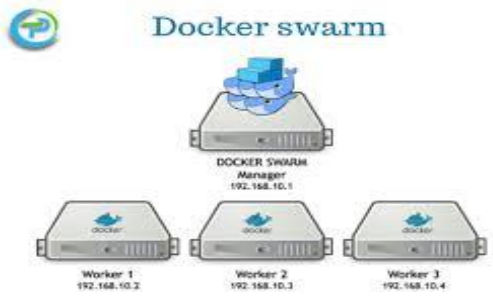
3

Docker Swarm



Docker Swarm

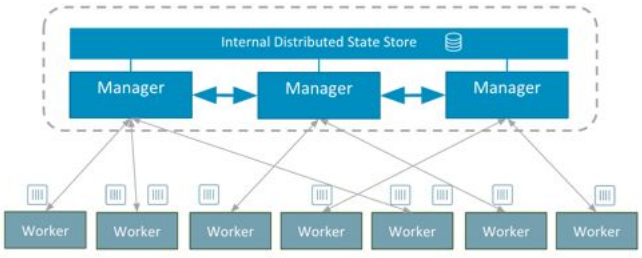
- **Docker Swarm** is native clustering for Docker.
  - Docker Swarm Mode comes integrated with Docker Platform.
  - Docker Swarm Mode is rightly integrated which means that you don't need to install anything outside to run Docker Swarm



Docker Swarm

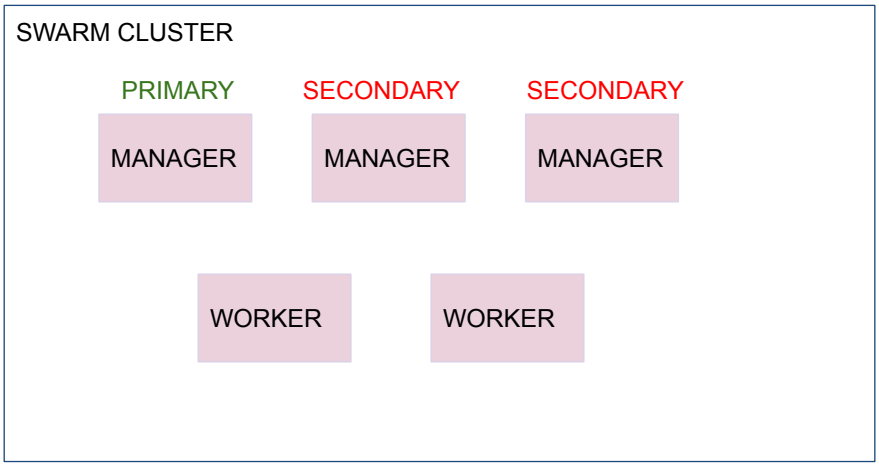
Manager Node:
The primary function of manager nodes is to assign tasks to worker nodes in the swarm.

Worker Node:
In a docker swarm with numerous hosts, each worker node functions by receiving and executing the tasks that are allocated to it by manager nodes.



Docker Swarm

Raft consensus in swarm mode



Docker Swarm

Raft consensus in swarm mode

- When the Docker Engine runs in swarm mode, manager nodes implement the Raft Consensus Algorithm to manage the global cluster state.

- The reason why Docker swarm mode is using a consensus algorithm is to make sure that all the manager nodes that are in charge of managing and scheduling tasks in the cluster, are storing the same consistent state.

Docker Swarm

Raft consensus in swarm mode

- Having the same consistent state across the cluster means that in case of a failure, any Manager node can pick up the tasks and restore the services to a stable state. For example, if the Leader Manager which is responsible for scheduling tasks in the cluster dies unexpectedly, any other Manager can pick up the task of scheduling and re-balance tasks to match the desired state.

- It is recommended to create clusters with an odd number of managers in Swarm, because a majority vote is needed between managers to agree on proposed management tasks according to `Raft Algorithm`.

4 Declarative vs Imperative

Declarative vs Imperative

Declarative and **imperative** approach is a DevOps paradigm or programmatic approach.

- While using an **imperative paradigm**, the user is responsible for defining exact steps which are necessary to achieve the end goal, such as instructions for software installation, configuration, database creation, etc.
- In **declarative paradigm**, instead of defining exact steps to be executed, the ultimate state is defined. The user declares how many machines will be deployed, will workloads be virtualised or containerised, which applications will be deployed, how will they be configured, etc.

Declarative vs Imperative

imperative focuses on **how** and declarative focuses on **what**.

Imperative approach:

1. Build the foundation
2. Put in the framework
3. Add the walls
4. Add the doors and windows

Declarative approach:

I want a tiny and cute house.

5

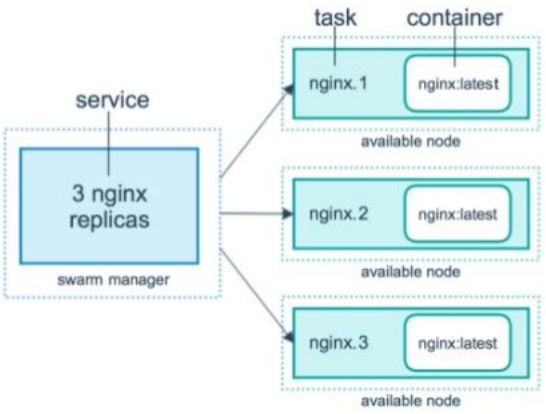
Services and Tasks

Services and Tasks

Services and Tasks:

- A **service** is the definition of the tasks to execute on the manager or worker nodes. It is the central structure of the swarm system and the primary root of user interaction with the swarm.

- A **task** carries a Docker container and the commands to run inside the container. It is the atomic scheduling unit of swarm.



THANKS!

Any questions?

You can find me at:

- alex.d@clarusway.com