Software Architecture: Assignment 2

Dockerization, group 6

Subject:

Software Architecture

Teacher:

José Luis Assadi

Students:

Manuel Bentjerodt

Nicolás Gebauer

Jorge Plaza de los Reyes

Delivery date:

August 22

1. HISTORY AND USE CASES OF VIRTUALIZATION TECHNOLOGY

Virtualization is the usage of software to create abstraction layers over the main computer by using its hardware components. Making it able to be divided into multiple (virtual) computers, with different(or same) OS, these are called Virtual Machines/VM. This technique takes us back to the 1960's, but its main usage began in the 90's, where single OS machines presented issues for third-party/legacy apps to run besides the main frame and the server, making this the number one solution for these issues due to the cost reduction and efficiency. In terms of functionality, the key to virtualization is the Hypervisor, the one in charge of sharing shared resources for the virtual machines. These can be divided into two categories, Type 1 and Type 2, a Type 1 Hypervisor is built on top of the main hardware of the machine, without a main OS running. Type 2 Hypervisors are built on top of an operating OS.

Fast forward to current days, virtualization is still being used for the same purposes, mainly on server implementations in enterprise data centers for cost reductions on maintainability, energy consumption, floor space and portability/reliability. Some examples of these applications are:

- Type 1: A company needs to save on the implementation of server functionality. So, in the space of a server it divides it into three independent systems, one for the applications, another on the server and the third one for the data. The company also has a backup server, open to the transfer of these functionalities in case of failure. Saving in maintainability and reliability too.

- Type 2: A student is asked to develop a web application on Linux, but his laptop operates on Windows OS. So he installs VM applications and downloads a Linux image to use on his computer. This approach will be more costly in terms of resources.

2. HISTORY AND USE CASES OF CONTAINERIZATION TECHNOLOGY

Containerization of applications or services arises for various reasons, but the main one is the need for consistency between environments. Who hasn't heard the phrase "but it works on my computer" from a developer? This problem typically arises due to differences between the environments where the application runs: locally, on the server, and others. For this reason, the container concept emerged, which is a set of packages including the application and everything necessary to run it, regardless of where the container is deployed.

Other factors lead developers to containerize their applications, such as portability. Software becomes easily transferable, and there's simpler control over its dependencies. Also, it allows for quick deployment, facilitating scalability.

The beginnings of containerization date back decades. In 1979, operations like chroot in Unix operating systems allowed processes to run in different directories. In 2000, the FreeBSD operating system introduced the concept of "JAILS", advancing the chroot idea by offering independent virtual environments in network and file systems, yet sharing the operating system's kernel. In 2005, Solaris implemented "zones" which, although previously called containers, separated the kernel from these isolated environments through virtualization.

More recently, in 2008, LXC (Linux Containers) emerged. Instead of fully virtualizing hardware, it virtualized the operating system, sharing hardware-level resources with other containers. Docker, in 2013, globally popularized the concept of application containerization.

Containerization encapsulates an application in an environment with everything necessary for its operation. However, with the rise of microservices architectures, it's common to containerize not only the complete application but also its independent services. These services, despite being independent, are interrelated. Hence, the need to orchestrate these containers arises.

There are tools like Docker Compose, structured in YAML language, ideal for local environments. For more scalable solutions, options like Kubernetes or AWS Elastic Container Service (ECS) are available.

3. COMPARISON BETWEEN VIRTUALIZATION AND CONTAINERIZATION

Virtualization is a technology that creates virtual machines (VMs) on a physical server, allowing multiple operating systems and applications to run independently on a single physical machine. This is done through the hypervisor, which manages and allocates hardware resources to the VMs.

Next, containerization is a technology that encapsulates applications and their dependencies in units called containers that share the same underlying operating system. Container engines such as Docker make it easy to create, deploy and manage containers in a variety of environments.

So, virtualization and containerization are two fundamental technologies that are influencing the IT infrastructure landscape. Both enable efficient resource management, isolation and scalability, but their methodologies and use cases differ significantly. In this section, we compare these two paradigms in detail.

- Architecture
    - A hypervisor is used to run numerous virtual machines (VMs), each with its own operating system (OS). This process is known as virtualization. In this case, separate instances of the operating system generate a resource overhead.
    - The creation of containers enables application isolation and dependency management. Because containers share the host operating system, the configuration is portable and lightweight.
- Isolation:.
    - As a result of the individual configuration of the operating system for each instance, virtualization provides a high level of isolation between virtual machines. For sensitive workloads, higher security limits are available.
    - By sharing the operating system kernel, it provides effective isolation at the application level. It is suitable for most use cases, although not as strict as VM isolation.
- Resource efficiency:

    - Because each VM needs a complete operating system, virtualization is resource intensive. Sub-optimal resource utilization can be due to the overhead of running multiple OS instances.
    - By using containers, you can efficiently share resources while using the host operating system. Utilization is improved and resource waste is reduced with this strategy.
- Performance:
    - Due to hypervisor overhead and multiple OS instances, virtualization results in slightly reduced performance.
    - As a result of the direct interaction of containers with the host OS kernel, containerization offers excellent performance. Startup times are shorter and productivity is higher overall.
- Portability:
    - Different hypervisor formats and dependencies can make virtual machines (VMs) less portable.
    - By encapsulating dependencies and applications, containers excel in portability and ensure consistent behavior across multiple environments.
- Deployment time:
    - Because a full operating system must be booted prior to VM deployment, it takes longer.
    - Due to the portability and light weight of containers, fast deployment. This agility is useful for microservices and frequent scalability.
- Use cases:
    - Perfect for situations requiring strong isolation and running multiple OS environments on a single physical machine.
    - With a focus on rapid development and scalability, this technology is ideal for microservices, continuous integration and delivery (CI/CD) workflows and cloud-native applications.

- Overhead:
    - Due to the requirement of different OS instances in each virtual machine, virtualization generates higher overhead.

    - Minimal overhead because it uses the host OS kernel and improves resource utilization.
- Management:
    - Requires more administration work, including updating and patching each virtual machine's operating system.
    - Kubernetes and other container orchestration tools simplify management and enable automated scaling and upgrades.

The decision between virtualization and containerization ultimately comes down to your specific needs. Virtualization is good for strict isolation and legacy applications, while containerization is ideal for agility, resource efficiency and scalability.

4. AWS ELASTIC CONTAINER SERVICES (ECS)

AWS Elastic Container Services (ECS) is one of the advanced services that Amazon offers to developers for container orchestration. It goes beyond simple tools like docker-compose and provides additional capabilities, including easy scaling, seamless integration with other AWS services, and load balancer deployment.

To properly understand ECS, it's essential to first familiarize yourself with the AWS ecosystem. A key tool in this context is Elastic Container Registry (ECR). ECR serves as a container registry service that allows developers to store, manage, and deploy Docker images. Repositories in ECR can be both public and private and shared with other users. One of ECR's distinctive advantages is that it can automatically scan images for vulnerabilities, and also allows for image encryption using the Key Management Service (KMS).

After establishing a repository in ECR, the next step is to upload the images of our applications. AWS provides an intuitive interface and specific commands to facilitate this process. Once the image is uploaded, ECR displays relevant information about it, such as the upload date, size, URI, and any detected vulnerabilities.

In ECS, one of the central concepts is the Cluster. A cluster is a logical set of resources where services can be deployed. These services consist of tasks, which in turn encapsulate our Docker images. When deploying, you can opt to use Amazon EC2 (Elastic Compute Cloud) instances, which are virtual machines with configurable capabilities. If you choose this option, you can use the Amazon Machine Image (AMI) specific to ECS, which already has the ECS agent pre-installed, simplifying integration. Alternatively, you can use AWS Fargate, a serverless solution that handles the underlying infrastructure, eliminating the need to manage it directly.

When establishing a cluster in ECS, you must select an existing Virtual Private Cloud (VPC) in your AWS account. During this process, you will decide whether to use EC2 or Fargate-based infrastructure. Later, when defining a new task, you will have options to configure aspects like the internet protocol (commonly TCP), the container port, the image URI to use, among other details. You can also define aspects such as CPU, RAM, storage, and log settings, with variations depending on whether you are using EC2 or Fargate.

Once a cluster is established in AWS ECS, whether you've recently created it or had it in operation, the next step is to deploy a service or task, depending on your needs. In this process, AWS offers various configuration options. For example, you can set up a load balancer to distribute incoming traffic among your containers, or configure specific network details for your application. Once this step is completed, the service will be active in your cluster, allowing you to monitor the status of your containers and access them through their respective public and private IP addresses.

Suppose that after some time, you decide to update your application. The first step would be to upload the new image of your application to ECR, AWS's container registry. Later, in ECS, you can go to the definition of your existing task and create a new revision. This process will present an interface similar to the initial setup, but now you can modify the URI to point to the new image you have uploaded. Once you've made the necessary changes, in your cluster you can update the service to use this new task revision. What's remarkable here is that AWS handles a smooth transition: the container with the previous version of your application won't stop until the container with the new image is fully operational, ensuring constant availability of your service.

In summary, to understand all this, we can use the kitchen analogy.

The image of our application is your recipe, the container is the dish cooked according to that recipe, the task is how you serve that dish (for example, with a specific side), the service is making sure to always have three dishes ready on the table, and the cluster is the kitchen where everything is prepared.