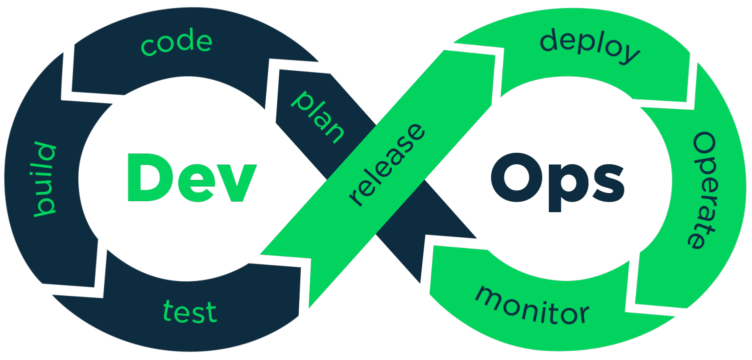
B5 - Advanced DevOps

B-DOP-500

Whanos

Automatically deploy (nearly) anything with a snap



Whanos

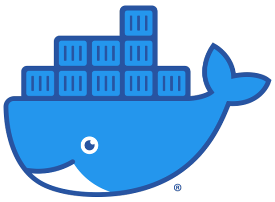
- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.

DevOps is a collection of great notions and technologies that allows oneself to have a truly great control over the lifecycle of an application.
Mastering one notion or technology is like owning an Infinity Stone: you become very powerful.
Now, imagine owning several Infinity Stones at the same time, what a powerful programmer you would become!

This dream is about to come true. You have started your DevOps journey last year, and up to now, have collected 4 Infinity Stones:

- the **Containerization Stone** with Docker;
- the **Task Automation Stone** with Jenkins;
- the **Configuration Management Stone** with Ansible;
- and the **Orchestration Stone** with Kubernetes.

You started your journey with Moby Dock, the Docker whale.



And you are going to finish it with another one, very knowledgeable about Infinity Stones: *Whanos*.

Mastering individual DevOps notions and technologies is a good thing, and it is even better mastering them in combination with each other.

This is what this project is about: you will have to combine your knowledge of the four technologies previously seen to set up an infrastructure using all of them.

You will go further into each technologies, with new features you (probably) did not see during the projects their respective technologies were each introduced in.

GENERAL DESCRIPTION

As you might have guessed, Whanos is a powerful being, and wants you to set up an as-powerful *Whanos infrastructure* that allows any developer to automatically deploy an application into a cluster, just by a push to their Git repository.
Following a push to a Whanos-compatible repository, it will perform the following steps:

1. Fetches the Git repository.
2. Analyses its content to determine its technology.
3. Containerizes the application into an image, based on both a *Whanos image*, and an eventual user-provided image customization.
4. Pushes the image into a Docker registry.
5. If a valid `whanos.yml` file exists, deploys the image into a cluster following the given configuration.

More information on these points is given below.

To tackle on this great task, you will use the 4 Infinity Stones you acquired so far:

- **Docker** as the containerization technology;
- **Kubernetes** as the clustering/orchestration technology;
- **Jenkins** as the task automation technology, linking the previous two;
- and **Ansible** as the configuration management technology, to deploy your infrastructure.

If particular settings or elements are not specified or addressed in the subject, you are free to do as you please with them. **However, if in any doubt, ask your local teacher.**

Whanos-compatible repository specifications

In order for a repository to be usable within the Whanos infrastructure, it must contain a single application (written in a supported language), whose source code and resources are in an `app` directory placed at the root of the repository.

This section details the structure of the repositories that are considered compatible with the infrastructure to be created, not the delivery repository in which you will have to turn your project in.

Supported languages

The following languages **must** be supported by your Whanos infrastructure:

- C;
- Java;
- JavaScript;
- Python;
- Befunge.

The commands below are **all** expected to be executed from the root of the repository.

C

- **Default compiler**: *GNU Compiler Collection 11.2*.
- **Criteria of detection**: Has a `Makefile` file at the root of the repository.
- **Dependency system**: None.
- **Compilation**: Using "`make`".
- **Execution**: With the resulting compiled binary.
- **Base image name**: `whanos-c`.

Whanos-compatible C applications are expected to be compiled into an executable `compiled-app` file, placed at the root of the repository.

Java

- **Default version**: *Java SE 17*.
- **Criteria of detection**: Has a `pom.xml` file in the `app` repository.
- **Dependency system**: *Maven*.
- **Compilation**: Using “`mvn package`”.
- **Execution**: Using “`java -jar app.jar`”.
- **Base image name**: `whanos-java`.

Whanos-compatible Java applications are expected to be compiled into an executable `app.jar` file, placed in a `target` subdirectory; the use of the `target` directory is a well-established convention when using Maven.

JavaScript

- **Default version**: *Node.js 14.17.5*.
- **Criteria of detection**: Has a `package.json` file at the root of the repository.
- **Dependency system**: *npm*.
- **Compilation**: Not applicable.
- **Execution**: Using “`node .`”.
- **Base image name**: `whanos-javascript`.

Python

- **Default version**: *3.10*.
- **Criteria of detection**: Has a `requirements.txt` file at the root of the repository.
- **Dependency system**: *pip*.
- **Compilation**: Not applicable.
- **Execution**: Using “`python -m app`”.
- **Base image name**: `whanos-python`.

Befunge

- **Default version**: *Befunge-93*.
- **Criteria of detection**: Has a single `main.bf` file in the `app` directory.
- **Dependency system**: Not applicable.
- **Compilation**: Free choice.
- **Execution**: Free choice, from the root of the repository.
- **Base image name**: `whanos-befunge`.

A single Whanos-compatible repository cannot match multiple detection criterion at the same time.

When encountering a repository that is not, regardless of the reason, Whanos-compatible, the handling is left free.
Triggering a build error is a recommended way to proceed.
It is not your duty to make a repository Whanos-compatible if it is not in the first place.

Whanos images specifications

The *Whanos images* are the images on which all application running on the Whanos infrastructure are based on.
They must have the following characteristics:

- themselves based on an official image;
- use the Bourne-Again shell for **all** image build instructions;
- work in an `app` directory at the root of the image;
- (if applicable) copy the dependency file(s) and install the dependencies;
- copy (re)sources;
- (if applicable) compile the application and get rid of the sources and other unnecessary files, in order to only leave what is needed for execution;
- sets the execution command based on the one defined for the language (in the previous section).

An image is only considered official if **marked as such** on Docker Hub.
`esolang/befunge93` is not an official image. ;)

Make sure to leave in the resulting image only what is needed for the execution of the application. For example, source files are not at all needed when a C application has been compiled.

Image types

Two different types of Whanos images have to be created for each language, corresponding to two different types of use.

Standalone images

Applications that do not need to configure their runtime environment further than the default configuration above will use *standalone images*.

These images must allow for the application to run smoothly without external modification.

Such applications will have no `Dockerfile` at the root of their repository.

Standalone images must be named like the base images, suffixed by `-standalone` (e.g.: if the base image name is `whanos-haskell`, its standalone counterpart will be named `whanos-haskell-standalone`).

Base images

Some applications will need to configure their runtime environment further than the default configuration described above, but they will still be *based* on Whanos images; as such, they will be based on another type of image: the *base images*.

These images must set the environment up just like the standalone images, but you have to keep in mind that they will be referenced by the `FROM` instruction of a custom `Dockerfile`.

Applications that use base images instead of standalone images will have their own `Dockerfile` at the root of their repository, starting with a `FROM` instruction referencing the image they want to base them on.

The base images must be able to be built on their own, without any application code, resource, or dependancy information available.
Such elements will only be available when building the custom image based on the Whanos base image, **you must keep this in mind.**

Take a look at the example applications, you can gather valuable information about the way the Whanos infrastructure is expected to work.

JENKINS INSTANCE

In order for the applications to be automatically containerized and deployed, you will use Jenkins.

The Jenkins Configuration as Code approach, which you used in the *my_marvin* project, is not mandatory for this project, but is still a great approach for you to use, since having an entire configuration just within a file greatly helps in setting up and deploying the Jenkins instance.

The Jenkins instance must meet the specifications detailled below.

USERS

Signing up must be disallowed.

A user named *Admin* must be created and must have:
- an id `admin`;
- all the rights.

Further users can be created if desired.

FOLDERS

WHANOS BASE IMAGES

- Is named *Whanos base images*.
- Is at root.

PROJECTS

- Is named *Projects*.
- Is at root.

JOBS

Each of the following jobs is expected to be enabled and to be a freestyle job.

WHANOS BASE IMAGES BUILD JOBS

For each supported language, a job must be created, and must:

- be named like the language's base image name (e.g.: `whanos-haskell`);
- be in the *Whanos base images* folder;
- build the corresponding base image so that it is available for the Jenkins instance host to use.

Build all base images

- Is named *Build all base images*.
- Is located in the *Whanos base images* folder.
- When executed, triggers all base images build jobs.

link-project

- Is named *link-project*.
- Is at root.
- Has the necessary parameters to perform its task.
- When executed, links the specified project in the parameters to the Whanos infrastructure by creating a job with the specifications listed below.

Jobs created by the `link-project` *job*

The following specifications apply to all the jobs created by the `link-project` job.

- Checks every minute for changes in the repository.
- When a change is detected:
  - containerizes the respository's application according to the specifications described in the *Whanos images specifications* section;
  - if applicable, deploys the application into a Kubernetes cluster (see below).

For the project, only the default branch of the repository is to be considered. Handling other branches is considered as a bonus.

Kubernetes cluster

Automatically containerizing applications is nice, but automatically deploying them is even better! ;)

If a repository has a `whanos.yml` file at its root, and that this file contains a `deployment` root property, it is the sign that the application has to be deployed into a Kubernetes cluster.

`whanos.yml` contents

The `whanos.yml` file can contain a `deployment` root property, which itself can contain:
- `replicas` -> number of replicas to have (default: 1; 2 replicas means that 2 instances of the resulting pod must be running at the same time in the cluster);
- `resources` -> resource needs, corresponding to Kubernetes' own resource specifications (default: no specifications; the syntax expected here is the same as the given link);
- `ports` -> an integer list of ports needed by the container to be forwarded to it (default: no ports forwarded).

Accessing the application from the outside world

If the `whanos.yml` file defines ports, they must be accessible from the outside world (not necessarily on the same port; e.g.: if an application needs to have its port 80 accessible, it is not necessarily needed for the host machine that the application is going to be accessed through to bind it to its own 80 port).

This last part is the one with which you have the most freedom in regards to the way to proceed.

The methods are left free, but **they must be documented.**

There must be at least 2 nodes in the cluster.

Deploying your infrastructure

Your instance must be deployed online, and as much as possible by using Ansible.

The ideal situation is to only have to fill in the necessary environment variables and launch your playbook, in order to then have a completely deployed and working infrastructure.

If a part of your infrastructure is not deployed with Ansible, you will be asked for justification.

The usual Ansible good practices (playbook idempotency, using modules as much as possible, etc.) are expected to be followed.

You are allowed to use Ansible Galaxy and community-made modules.

Evaluation

The project will be evaluated **during a defence**.

The **functional aspect** will obviously be taken into account, but the **respect of good practices** and the **cleanliness of your work** will also weight into your mark.

The infrastructure you will set up will **need to be deployed online and accessible by the evaluator** during the defense.

Previously seen notions, such as “redeployability” (i.e.: the ability to easily redeploy an infrastructure), are obviously going to be taken into account.

Delivery repository structure

You will have to place your Dockerfiles in an `images` directory, itself placed at the root of your repository.

Each language’s Dockerfiles will be named `Dockerfile.base` and `Dockerfile.standalone`, respectively for the base and the standalone images, placed in a directory bearing the name of the language, in the following fashion:

```
~/B-DOP-500> ls images
befunge  c  java  javascript  python
~/B-DOP-500> ls images/javascript
Dockerfile.base  Dockerfile.standalone
```

Documentation must be placed in a `docs` directory, itself placed at the root of your repository.

If you have any **usable** files related to Jenkins, Kubernetes, or Ansible (e.g.: helper scripts, configuration files for easy redeployment of the infrastructure), they need to be placed in your repository. You are free to decide the file structure.

Any file not present in the repository will be considered to be the equivalent of “hard-coded” into your infrastructure.

Some files (such as secrets, or generated configuration files that are specific to each instance) will obviously not, and **should not**, be in the repository, but other files (such as eventual deployment templates or Jenkins Configuration as Code files) should be in your repository if you want to benefit from them during the defence.

Tips and tricks

You will have to go further into each of the technologies you have seen until now in order to succeed, especially Docker and Jenkins; so immerse yourself in their documentation to find the magic features you want!

Doing the project with only Docker, Jenkins, Ansible, and Kubernetes is difficult; in order to help you, you may need to write helper scripts, as well as setting up other elements, such as a Docker registry. The choice is yours!

Docker provides an official image of a Docker registry. Handy, is it not?

Want to go further?

Whanos loves that! If you want to be blessed with the power of DevOps, you can for example:

- support more languages (e.g.: C++, Go, Rust, or Brainfuck);
- handle other branches than the default one;
- display a Whanos ASCII art when launching a job;
- extend the configuration capabilities.