1 Practical Phase at scVenus

1.1 Introduction

At Atos, we have the possibility to choose the department in which we want to perform our practical phase. For this purpose, a lot of departments introduce themselves and what they do, to the students. That is how I found my department, science+computing, and more specific scVenus.

The overarching project I worked in, is called PeekabooAV. This is an Anti-Virus software, which can receive a file, for example from an email client, and pass it through its own rule engine, which intern may use other programs, such as behavioral analysis, to determine the risk of the file.

1.1.1 Motivation

PeekabooAV is written to have a high degree of configurability and extendability. A major problem with this approach is that it is not trivial for a user to try out PeekabooAV in an efficient manner.

Imagine this scenario:
You are a System Administrator, with an on-premise email service. You are looking through GitHub, or searching articles for better Anti-Virus, to cut down the manual work you have to do. You stumble upon PeekabooAV, which sounds promising to you. But to know if it is the right fit for your needs, you would need to set up a whole test environment, with at least spam filtering and an email service. This process is cumbersome, and would likely stop you from even testing out PeekabooAV.

Although it is possible to test out PeekabooAV on its own, there is the definite need to test it with a full environment. This hurdle of setting up an environment, to test a software, is a major problem for adoption.

The motivation behind my work with the PeekabooAV Installer repository, is to provide a faster way to get a full pipeline environment, to foster the adoption of PeekabooAV.

1.1.2 Assignment

My specific task was to containerize the bleeding-edge version of PeekabooAV, to make a showcase pipeline, and further ease the future deployment of PeekabooAV. This is a sentence to pad out the page for dev. The pipeline, orchestrated with docker compose, is to include the following services:

1.1.3 Workflow

Throughout the phase, my workflow was predominantly guided by a meeting twice a week. In these meetings, I presented the current state of my work, and discussed the next steps with my two colleagues in this project. My workday changed a lot during the duration of my involvement in the project. This is due to the fact that I had to make myself comfortable with the new tools and the already quite mature codebase of PeekabooAV. After I acquainted myself with the Docker tools and the overall structure of the PeekabooAV codebase, I incrementally created each part of the pipeline. This process started with PeekabooAV itself, and culminated with the 3 other services together with their respective containers.

Throughout the phase, my workflow was predominantly guided by a meeting twice a week. In these meetings, I presented the current state of my work, and discussed the next steps with my two colleagues in this project. My workday changed a lot during the duration of my involvement in the project. This is due to the fact that I had to make myself comfortable with the new tools and the already quite mature codebase of PeekabooAV. After I acquainted myself with the Docker tools and the overall structure of the PeekabooAV codebase, I incrementally

1.1.4 Open Source

Open source is, at its core, a way to develop something. In many cases software. It does not rely on companies paying developers to produce a piece of software, instead it is community driven. Interested developers can contribute to the source code of the software, which is openly visible to anyone. To ensure the correct usage of the code, there are specific open source licenses, which dictate the way open source software can be used. Some often used examples include: The MIT License, which allows anyone to use the software for commercial and private use, and allows modification and redistribution of the software under any terms. A license that verges more towards the open source community, is the GNU General Public License v3.0 (often abbreviated as GPLv3). This license also allows use for private and commercial use, aswell as modification and redistribution, but only under the same license. This is supposed to guarantee that further progress stays in the open source realm. There are many other open source licenses, not only for use with software and source code, but for example regarding media. One of these open source media licenses is the Creative Commons Attribution-ShareAlike 4.0 International License. This license allows for the use of the media, and modification and redistribution of the media under the same license, similar to the GNU General Public License v3.0.

The licenses mostly do not cover how the development of software is done, but rather how the software is used. That induces the need to govern the development of open source software. The most common way to do this is by using tools like GitHub, which provides workflows opportunities for both open source and closed source software development. GitHubs role in the development process of open source software will be further discussed under the `Used Technologies > GitHub` section.

1.2 Used Technologies

1.2.5 GitHub

Millions of developers and companies build, ship, and maintain their software on GitHub - the largest and most advanced development platform in the world. ~GitHub

Above is how GitHub describes itself. For this paper, we will concentrate on the ’building’ and ’maintaining’ parts.

open source version control software. Version control, also called revision control, is a tool for managing changes of information. Today's version control software, is able to track changes of information, keep the history of changes available, and log who made which changes.

Above is how GitHub describes itself. For this paper, we will concentrate on the 'building' and 'maintaining' parts. GitHubs name is derived from the git software. Git is an

The open source workflow for GitHub starts by forking a repository you want to contribute to. This essentially creates your own copy of the repository, which you can then modify. The next concept are branches.

Branches are used to diverge from the main branch, or trunk, in order to not affect the work of other developers. With the use of branching, we also gain the possibility to make a pull request. Pull requests (often abbreviated as PR) are a way to propose to merge your own branch into another one, often the main branch.

As pull requests are the primary way that new code is contributed to open source software on GitHub, we wanted to have the results of my work in a pull request at the end of the phase. That includes the branching workflow, and also the review process once a pull request has been opened. In summary, a review is done by another developer, who looks over all your changes. They then either suggest changes, or approve your changes by closing the review. This process has the benefit of being transparent, and also allows for multiple reviewers. When a reviewer proposes changes, they can do so in form of a conversation directly on top of the code. In these conversations, anybody can discuss the proposal and the conversation can be marked as resolved.

The last part of the pull request flow, is merging. Merging describes how the changes from the pull request will be combined with the main branch. There are multiple ways to merge pull requests, each with its own advantages and disadvantages. Merging with a merge commit, or a true merge, creates a new commit with both branches as the parent, to then include the other wanted commits on top of that. This way of merging is often avoided in more active projects, because the extra merge commits it creates, do not include any useful information and thus can be seen as spam by some developers. Another way is squash merging. This method squashes the changes from all commits into just one commit, and pushes that commit on the main branch. This does not clutter up the history, but when too many commits are squashes into one, it can be hard finding what exactly has been changed, or which part is the origin of an error. The third way is rebase merging. This way of merging is often used in more active projects, because it does not create any extra clutter, but instead rewrites the history of the main branch. As the commits are patched onto the main branch one after the other, there could occur conflicts. Conflicts are blocks of a file where git can not resolve how to change the file with the current patch. This needs to be handled manually.

We used rebase merging in our project, because we did not as we developed alongside the bleeding-edge version of PeekabooAV.

We used rebase merging in our project, because we did not want the spam merge commits, and there were no conflicts

1.2.6 Docker and docker compose

Docker is another tool, that is used industry-wide, for easier deployment of software, through containerization. Docker can, through virtualization, create containers which can then run software. Virtualization is a concept to create an isolated computer system, that runs on top of the host computer system. This means that the virtualized system believes it can communicate with the hardware directly, but the communication is routed and managed through the host system. In the case we discuss here, virtualization is used to create a smaller system for a specific task on top of the host system. Another way virtualization is used in the industry, to split up the whole host system with many other smaller systems, so the host has minimal ressources left over, after lending each sub system an amount. These can either be ones made for a specific task, or more general purpose systems.

Another advantage of using virtualization, is the isolating effect it has. By default, the virtualized systems do not know anything about each other, they are fully isolated. By the virtue of being achieved in software, there is the possibility to create a network of virtualized systems, which can communicate with each other.

In the case of Docker this is used to create another layer of abstractions, containers. The specialized virtualized system mentioned before, is what is often referred to as a container. Docker is the software tool to run and manage these containers, which are build with another tool, in dockers case Mobyproject. To create a container with docker, you need a `Dockerfile` that describes the desired system. This text file consists of the commands that are to be executed to get the system to the wanted state. It is not needed to start from scratch with every `Dockerfile`, you can start from any other image. Images are what results from building a `Dockerfile`, this image can then be made into a container by the Docker engine. So the making of said `Dockerfiles` is recursive at its core.

If, for example, you want a simple container to capture the network traffic in a network, you begin from an image like `alpine` and install a tool like `tcpdump`, or if available you start from another small container that has `tcpdump` already installed.

There are other technologies around docker containers, with which they interact with the host or with each other. There are volumes, to create a shared directory or file between the host and the container. Additionally, there are Networks, which are used to create a virtual network between the containers, that can also be accessed by the host if needed.

Another tool made by Docker is docker compose (or docker-compose). This makes it easy to manage multi-container apps, which use above-mentioned

file (to spec named `compose.yaml`), in which you fully define all the needed containers, volumes for those containers, and networks to connect them.

Another tool made by Docker is docker compose (or docker-compose). This makes it easy to manage multi-container apps, which use above-mentioned technologies. With docker compose you need another text

To make clear how those containers are described, we will continue the example of capturing network traffic between containers.

Above code represents a `compose.yaml` file, setting up a microservice backend with an according database, and a container that uses tcpdump to capture network traffic. This file follows the version three of the docker compose file format. One creates a service for each container, that is described mainly by its name, for example `microservice` on line four, and the image used, for example in line five. In this example, the use of volumes and networks is also illustrated. There are two volumes with different responsibilities, one for the database, and one for the tcpdump container. The `db_data` volume is needed to make the database persistent. Due to the reproducability of these containers, no data on the virtualized system is persistent after runtime. To achieve persistence, the `db_data` directory, from the host system, is mounted inside the container, and is thus not a direct part of the virtualized system. On the other hand, the `dump` volume is for easy access to the network traffic capture files produced by the tcpdump container. If this volume would not exist, one would have to have another way to access the capture files, most likely by using a lengthy command to copy the file from the container to the host system. With the volume, one can directly access the files from the host system.

After all wanted services are defined in the `compose.yaml` file, one can start all elements with a single command, `docker compose up`. This command will start all containers, if they are not already running, and if applicable restarts containers with changes made to them, and creates the networks.

I will cover more specific features Docker and docker compose offer, where needed at a later point in this paper.

1.2.7 MTA

Generally, MTA is an abbreviation for **M**essage **T**ransfer **A** gent, but for our use case the **M** stands for **M**ail. The name is reasonably self-explanatory, this software can either **t**ransfer emails to another MTA or to an MDA, **M**ail **D** elivery **A**gent. Or it can reject or block the email for a number of reasons.

The process of rejecting an email is the important part to this project. An email can be rejected for a variety of reasons, for example by a connected spam filtering system, as described in the next section. When an email is rejected, it is not forwared to the next MTA/MDA, but a notification of the rejection is sent back to the sender. The sender can then decide what to do next. In most cases the sender is also an

sent again when the queue is flushed.
The process of rejecting an email is the important part to this project. An email can be rejected for a variety of reasons, for example by a connected spam filtering system, as described in the next section. When an email is rejected, it is not forwared to the next MTA/MDA, but a notification of the rejection is sent back to the sender. The sender can then decide what to do next. In most cases the sender is also an
MTA, and it puts the email back in its queue, so it can be

PeekabooAV uses this mechanism to reject an email while it is still being analyzed, and once the same email is sent again the cached result will be used to determine if it is finally rejected or accepted.

1.2.8 Spam Filtering System

Spam filtering system are the systems used to decide if an otherwise valid email is unwanted or not. An email can be unwanted due to being an unsolicited advertisement, containing a phishing link, having a virus attached, or other reasons. Most of these spam filtering system make their descision after running the email, or parts of the email, through possibly hundreds or thousands of rules of varying importance. They then combine the resulsts to a heuristic value which is used to decide if the email is classified as spam. These rules can use a variety of techniques, for example scanning the text content for certain words often used in scams, or looking at the sender email address in the header of an email, or now with PeekabooAV test the behavior of attachments.

To achieve this, most spam filtering systems are configured in the MTA as a milter. Milter stands for Mailfilter, which is a module that is registered as a step in the processing of an email. In our specific case this means that, rspamd receives emails as a milter from postfix, and uses PeekabooAV as a custom module to analyze the email.

1.2.9 Email spec

The email spec, or more precise, the Internet Message Format (IMF), is specified by the Internet Engineering Task Force (IETF). The specification takes place with RFCs, Request for Comment, which are documents that can be discussed by other members. The most important RFCs for the Email Framework are RFC 2822 and RFC 5321. Those two RFCs are updated by other later RFCs like RFC6854. The details of these RFCs do not fall into the scope of this paper, therefore I will discuss only parts that were of importance in the context of this project.

One notable aspect of these RFC is that they are not concrete standards that are followed. They are better described as a syntax or language for authoring Email messages. Due to this, two emails, being visually and contentually the same, can be different in mulitple points, if sent from different Email clients. Due to these points, no concrete standard, and the separated nature of the important RFCs, validating and parsing an Email message is a hard problem with a lot of possible edge cases.

I experienced this first hand, while working on this project.

attachment is stored in the Email. To not go into specifics in this chapter, this problem,and the solution to it, will be discussed in the rspamd chapter under Services.

I experienced this first hand, while working on this project. The problematic feature was how the meta information of an

1.3 Pipeline

1.3.10 Services

In the next section I will discuss which services we used in the showcase and the reasoning behind them, aswell as challenges or problem I have come across.

1.3.10.1 MTA - Postfixes

As mentioned above, we need an MTA to receive the email and use a spam filtering system. Additionally, we need something to ensure the email is sent again, some time after it was rejected. To make this as easy and realistic as possible, we use another instance of an MTA, which uses a queue to achieve this task.

For both MTA instances we use Postfix, which is a widely used open-source MTA. Postfix supports all UNIX-like system and still receives multiple updates a year, at the time of writing. For both postfix containers we use our own Dockerfile.

For this container we start from an alpine image, which is a very minimal and small version of linux that only takes up about 5.5 MB. We then install all needed programs, especially postfix and swaks in line 3 to 6. Afterwards we configure postfix to our needs:

This script converts specially formatted environment variables to postfix configuration commands and executes them. These variables need to take the form of either `POSTFIX_MAIN_CF` with the option name appended, or `POSTFIX_VIRTUAL`. The value of the variable is then used to set the option. These will be explained in more detail below. Finally, the script starts postfix in the foreground.

1.3.10.2 Sending MTA

As mentioned above, we use our own Docker image for this postfix container. And configure it accordingly with environment variables.

The code above is an excerpt from the `compose.yaml` where we define the `postfix_tx` service. This is the service used to send emails. The image we used is called `peekabooav_postfix`, which corresponds to the build directory `postfix`. The `Dockerfile` shown in the previous chapter is inside this directory. Docker Compose always checks if the image is present, otherwise it will build it with the build directory just discussed. Next, we set the hostname to `postfix_tx`, which corresponds to the name the service has in the network. The most important part of the service declaration are the environment variables. Here we set some trivial options, for example:

Finally, we set the ports so that we can actually access the service from the network.

1.3.10.3 Receiving MTA

Fundamentally this service is very similar to the `postfix_tx` service.

This code is an excerpt from the `compose.yaml` aswell. This `postfix_tx` service is used to receive emails. And the declaration is mostly similiar to the `postfix_tx` service. Configuring this service was more complicated due to the lacking documentation specific to our problem. That results in a config which is likely to be more verbose and extensive than it needs to be. Nevertheless, this is not a problem as these configurations do not impair the performance in any meaningful way.

The most important configurations, except the ones explained in the previous section, are:

The last part if this excerpt sets the service dependencies. By specifying the `rspamd` service as a dependency, we ensure that the `postfix_rx` container will only be started after the `rspamd` container. Additionally, we set the `condition` to `service_healthy` to ensure that the `rspamd` container is running as we expect it, before the `postfix_rx` container is started. The *healthiness* of a service is discussed more in the Spam Filtering System - rspamd chapter.

1.3.10.4 Spam Filtering System - rspamd

As explained above, we need a spam filtering system that communicates with PeekabooAV and acts as a milter for postfix. For this we chose rspamd, which is a widely used open source spam filtering system. An important feature of rspamd is the comprehensive Lua API, which allows us to write scripts to extend the functionality of the system. [@RspamdRspamd2022] We use this API to create a module that uses PeekabooAV to filter. Similar to the postfix containers, we created our own Dockerfile.

The structure of this Dockerfile is very similar to postfixes Dockerfile. We start from the same version of alpine linux and then install needed packages. The block of `COPY` commands copy over some configuration files and the custom Lua module that utilizes PeekabooAV.

Afterwards we apply a patch file which concludes the integration of PeekabooAV into rspamd, and some other configurations.

The `entrypoint.sh` we include is in one part similar to the postfix `entrypoint.sh`, in the sense that rspamd options can be set by using environment variables starting with `RSPAMD_OPTIONS_`. The other part of the entrypoint disables all rspamd modules, except the ones that are set in the `RSPAMD_ENABLED_MODULES` environment variable. This would not be wanted behaviour in a deployed environment, but it is useful in a showcase.

This excerpt defines the `rspamd` service. We set `image` to the image we created earlier, and `build` to the path to the directory where the rspamd Dockerfile is located. The `RSPAMD_ENABLED_MODULES` is the list of modules that we

This excerpt defines the `rspamd` service. We set `image` to the image we created earlier, and `build` to the path to the directory where the rspamd Dockerfile is located. The `RSPAMD_ENABLED_MODULES` is the list of modules that we want to enable. The `RSPAMD_OPTION_FILTERS` is the list of filters that we set to be empty. Furthermore we add a dependency on the `peekabooav` service, and add a healthcheck.

A healthcheck runs the `test` command with the specified options. `interval` is the time between each check, `timeout` is the time to wait for the command to finish before assuming failure, and `retries` is the number of times to retry the command before giving up. `start_period` is the time to wait before the first check. The service has a health status that can be used in the `depends_on` section of other services, like we did with `postfix_rx`.

While creating this service a few problems came up. At the time of my practical phase the old `docker-compose` command was still the default and standard. For most features that was not a problem, but that command was not in line with the latest compose specification. Importantly the `healthcheck` section was not supported by the old command. Due to this we needed to use Compose V2 which is the direct implementation of the compose specification. As the goal of this project was to ease adoption, needing a non-default installation of a tool was a regress.

The other problem was with how the email spec is handled by different clients. Specifically how exactly the filename is sent in the content-type and/or content-disposition headers. With some email clients the filename is sent with the content-type, with others it is only with the content-disposition.

These are examples of email excerpts. They were taken at the time of writing and presumably looked slightly different at the time of the project

As you can see, even if the filename is included in the same header, the quotation can be different. Due to this discrepancy I discovered a bug in the Lua module code. This bug was fixed by an external developer who also wrote the Lua module for the project.

1.3.10.5 PeekabooAV

The Dockerfile for PeekabooAV has to be more complex than the previous Dockerfiles. This is primarily due to PeekabooAV not being available for installation with a package manager like apt. Instead, we need to build the application from source, and still keep the final image size small.

To achieve this we used a special Dockerfile feature called multi-stage builds. [@UseMultistageBuilds2022]With this feature we can define a regular Docker image that includes all the needed tools to build an application, and afterwards we define the actual application image, where we can include specific files or directories from the previous stage. This feature is useful to keep the image size small, because a lot of tools that are needed for building an application are not needed for running it.

To achieve this we used a special Dockerfile feature called multi-stage builds. [@UseMultistageBuilds2022]With this feature we can define a regular Docker image that includes all the needed tools to build an application, and afterwards we define the actual application image, where we can include specific files or directories from the previous stage. This feature is useful to keep the image size small, because a lot of tools that are needed for building an application are

This excerpt is the first stage of the PeekabooAV Dockerfile, which is named `build`. The build stage installs all the needed dependencies and fully sets up PeekabooAV inside the `/opt/peekaboo` directory. This closely follows the installation instructions found in the PeekabooAV documentation, by installing some development tools, configuring a python environment, and creating needed configuration files.

This is the final stage of the Dockerfile, which will be used as the actual image. Here we start from a more extensive base image, the linux distribution Debian. From there we copy the `/opt/peekaboo/` directory from the build stage, this is the only interaction with the previous stage. Afterwards it is a regular Dockerfile similar to the ones before. We create a peekaboo user and install python, plus the needed libraries.

The entrypoint.sh is again set up to configure the service with environment variables.

Once more the service definition is slightly different. We use another condition in depends_on, service_completed_ successfully, which waits until the given service exits without an error. Another important bit is the stop_grace_ period. This sets the amount of time docker waits for the application to close itself after receiving the corresponding signal. If it does not do so in the specified time, the service is forcefully closed.

Furthermore, we do not set the environment variables directly inside the compose.yaml, but in an extra file called compose.env. This was done because the needed environment variables for most other services are already known by many developers as those are the industry standards. That is not the case with PeekabooAV, therefore we wanted more structure to the way the variables are input.

We also use this file in other services, such as MariaDB. Which is an industry standard small-mid scale SQL database that is used by PeekabooAV. Both PeekabooAV and MariaDB are configured with environment variables, and some variables, for example the database password, need to be the same in both services. Due to this it is logical to put these settings in a singular file.

The service definition for MariaDB does not include any new components, as the only important parts are the env_ file, which also points to the compose.env file. The other important bit is another healthcheck, to know when MariaDB is fully running.

1.3.10.6 Behavioural Analysis - Cortex

detecting if an attachment is malicious or not. We use Cortex, a project by The Hive Project. With a very modular system, you can analyze different observables, such as IPs, domains, files, and more. Cortex can use many tools to achieve this task, and it can be fully run via an API.

The behavioural analysis service does the heavy lifting of To go through an example, imagine file called `cat.png.bat` as an email attachment, with the following content:

A malicious file similiar to this is a common attack vector targeting Windows systems. As Windows does not show the real file extension by default, an unsuspecting user would see the file as just `cat.png` and would likely not think about trying to open it. But in reality, the file is a script which downloads an executable file from a remote server. Here nothing more happens, but it would be trivial for a malicious actor to further compromise the system from this point. We used this file in the development of PeekabooAV as demo malware, to test functionality.

With a behaviour analysis engine, we can analyze this file with different analyzers. Some analyzers might try to detect the mime-type of the file to specify it as malicious or not. Others try to execute the file in a sandbox and keep track of all changes to the filesystem. The latter could detect that the supposed `cat.png` is actually downloading an executable file. There are more than 100 analyzers that come Cortex, these also include well known tools such as VirusTotal and Google Safe Browsing. Although for this showcase, we only enabled a single analyzer called FileInfo.

The service definition for Cortex has no fundamental differences to the previous ones. Two notable things are the use of a premade image by The Hive Project, and the `docker.sock` volume. Internally Cortex uses docker to run each analyzer, in our use case this would mean running a docker instance inside another docker container. While this is possible, it is not recommended as it does not conform with the linux securiry model, and different issues regarding file systems can arise. [@UsingDockerinDockerYour] To solve this, Cortex can use any docker socket we pass to it. This essentially gives a container the ability to start another container alongside it on the same host machine, nullifying above mentioned problems. One limitation to this approach is that the host supposedly needs to be a linux machine, as Docker running on Windows via Hyper-V does not have a `docker.sock`.

Furthermore, Cortex is not set up to use directly from the docker image. As Cortex is controlled via web interface one would normally go through these steps on the first startup:

As this is a showcase meant to be started from scratch with a single action, this is not acceptable. We therefore created the `cortex_setup` service to do this. Below is the Dockerfile for that container.

As the cortexSetup.sh script is about 200 lines long, and most of the important code is very similar, I will only discuss parts of it here.

The script needs to know the locations of cortex and is used by cortex. These locations can be given to the script either via argument in the command line, or via environment variables, as seen above via the compose.env file that is included in the service definition.

The script needs to know the locations of cortex and elasticsearch, which is an open source NoSQL database that

The CORTEX_ADMIN_PASSWORD variable is used as the admin password, or if empty a random password is generated by the script.

The first step for the setup script, is detecting if Cortex is already set up. This is the case if we shut down the whole showcase and start each service back up. Trivially, we do not want to try to set up, an already set up system.

Above execerpt is used to check if Cortex needs to be set up. The queried endpoint can return a status code of 520, which indicates some internal error from which we deduce that the server is not set up. If the status code is 401, we know that Cortex is set up as it knows that the request we made is not authorized for that endpoint. Other endpoints can also be used, but some have more complex behaviour regarding the status code. We also check and exit if the supplied Cortex URL is not reachable.

Once we know that we need to set up Cortex we start following similiar steps as a user would do:

Most of these steps can be done by calling an API endpoint, which I found with the help of the developer tools in the Chrome browser.

Above is an excerpt from the entrypoint.sh, where we handle step 3. `curl` is one of the most common ways to make arbitrary http request from the console or a script on linux. Before the curl command we put out what we are going to do, in this case create the organization, together with some formatting to add color the output with VTS (Virtual Terminal Sequences).

Most of the other steps are handled similarly, with the endpoint being different and the data specified with `-d` changed accordingly.

The difference between the steps the script takes and what a user would do, are in how the API key is handled. Normally the user would copy the randomly generated API key from the web interface, and paste it in their other application. This is not doable in our case as we can not have any user interaction required in the setup process. Instead of copying the random API key, and somehow saving it to use in PeekabooAV, we find out in which Elasticsearch database Cortex saves its internal data. Once we know the database we can replace the API key with our own key, which the user supplied with the compose.env. By supplying it with in the compose.env file it can be accessed by any service using the compose.env file.

The function `check_last_command` is run after each step, it checks if the last command exited with status code 0. If so, it prints a green `o` and the script continues. If not, it resets the terminal with VTS and exits the script.

1.3.11 Architecture

There is essentially a chain of dependencies between the services.

In the figure above you can see the dependencies between the services. Those dependencies have the effect that the startup of the showcase can take a lot of time as postfix_rx, which is used to send a test email, has a long dependency chain. The upside to this is that you can not use the pipeline if it is not set up properly, possibly eliminating confusion.

1.4 Result

As explained in the beginning of this overarching chapter, the assignment was to containerize a showcase pipeline for PeekabooAV with the goal of easing adoption of PeekabooAV.

The assignment was fully functionally completed at the end of my phase. Although some shortcuts were taken, for example, only enabling one analyzer with cortex or disabling all other rspamd modules. But this is acceptable as it is made clear that it is a showcase and not suited for production. These shortcuts also do not affect the goal of easing adoption, as this is achieved by simply giving a user the ability to try out the pipeline with sample emails or even their own files as attachments. This pipeline is also a good start if one wants to start using PeekabooAV in a production environment.

There was some more work done after my phase, which mostly includes streamlining health-checks and some configurations, and cleaning up what is logged and what is suppressed, to improve the overall quality of the showcase.