The Applicability of Container Technologies for Building a Secure Production Ready PaaS

Ruben Vasconcelos, August 10th, 2018. *M.Sc. in Computing.*
Student Number: 16212630, email: ruben.vasconcelos3@mail.dcu.ie

***Abstract*—The way production software is managed and delivered has drastically changed over time to cope with shorter release periods in a highly competitive market. Most companies are now focusing their efforts on building internal platforms using Linux containers and other open source tools for the rapid delivery of software products.**

The basis of this project was to research the challenges faced by users trying to employ container technologies; more specifically, the areas that must be taken into account and the reasons why smaller companies are finding it hard to adopt the latest methods and processes.

We introduce PaasPure as a possible solution to help mitigate against standardisation problems. The tool aims to work as a layer of abstraction on top of the various elements that form a container platform, allowing its users to share full platforms specifications and source code. We claim this approach would considerably reduce the standardisation problems and subsequently the entry level barrier for anyone lacking the necessary amount of resources and knowledge required for building a production-ready PaaS.

Numerous experiments were performed to investigate the current capabilities of PaaSPure, the platforms built using it, Docker containers and its built-in orchestrator. From our results, we found that users believe PaaSPure or a tool of similar abilities could help reduce the standardisation problems faced when trying to adopt container technologies.

During our performance tests, we discovered there was a considerable performance increase when using clusters with a higher number of nodes; but the performance improvements were diminished under greater system loads. Even though clusters were not being used to their full potential, we noticed a significant decrease in the orchestrator's performance. We also found a problem with the Docker's AMI (Amazon Machine Image); the issue caused containers to take longer to start in instances with more than one CPU.

***Keywords*—Cloud Computing, Platform as a Service (PaaS), Linux Containers, Orchestration, Logging, Monitoring, Continuous Integration (CI) and Continuous Delivery (CD).**

I. Introduction

The success of software development companies is often directly linked to their ability to deliver software fast and reliably. According to Paul Clarke, software release periods have been reduced from one or two times per year to as short as hourly intervals, placing IT businesses under more pressure than ever before [1].

One of the solutions found to accomplish rapid delivery of software is the use of open source software for implementing complex delivery pipelines (sort of internal PaaS platforms). At the centre of the open source tools used are Linux containers, more specifically Docker. Docker is one of the world leaders in the delivery of container technologies [2].

These platforms built on top of Docker make use of central source code control tools and a CI/CD (Continuous Integration and Continuous Delivery) server for ensuring quality and delivering single services without perturbing the system as a whole [1]. Securely employing Docker containers in production is a complex task. Later on, we will further discuss the key areas and associated challenges that need to be considered; we will also look at the tools that surfaced in the PaaS market to help monitor, manage and deliver containerised applications.

One of the problems that has not received as much attention as the others described in this paper is the way container based platforms are set up. The processes used vary on a company basis, and setup methods often involve large amounts of scripting; the low process uniformity plus smaller budgets make it harder for smaller companies trying to adopt the same principles.

The tool developed as part of this practicum aims to offer a standard software development driven approach for creating container platforms by breaking platforms into smaller units called PureObjects. PureObjects are self-sufficient and portable across platforms of similar characteristics. The method enables users to share the source code and tests required for the creation and validation of container platforms via a central location (the PaasPure hub).

We split PureObjects into modules and components. Modules are a high-level overview of some general requirement such as infrastructure or logging while components are implementation specific. The application knows how to run modules and modules offer a layer of abstraction on top of components. This architecture gives users the freedom to do things like writing components in different programming languages.

II. Background

A. *Evolution of Virtualisation*

Traditionally, applications ran on dedicated servers (bare-metal); to deal with periods of higher demand companies had to acquire extra hardware and configure it manually. This hardware was underused during the periods of lower demand which caused other secondary effects, for example, wasted energy and poor software level security. This problem is known as server sprawl [3].

Virtualisation techniques like Virtual Machines (VMs) and Virtual Storage started emerging as a way of ensuring server consolidation. These technologies became the basis for what we today know as cloud computing [4]. Their inherent elasticity allowed for rapid scaling of resources in a way that non-virtualized environments found hard to achieve.

Linux Containers (a type of OS-level virtualisation) are the newest advancement in the evolution of virtualisation techniques [5]. According to Claus Pahl, containers had a direct impact on the progression of PaaS (Platform as a Service). Several companies are now investing vast resources into developing their internal software delivery pipelines, usually composed of various tools creating a sort of internal PaaS that can run on in-house data centres or public IaaS clouds (Infrastructure as a Service) [6].

B. Container PaaS

Containers have a bright future in the PaaS use case due to their inherent advantages over VMs. These include the ability to quickly deploy services across multiple servers and even different IaaS, minimal start-up/shut-down costs and very low resource usage resulting in reduced overhead and dependency issues in comparison to VMs [7].

Pahl breaks the evolution of PaaS into three generations, the most recent generation or third generation is composed of platforms built around Docker such as Deis and Flynn. We acknowledge that the latest generation of platforms is in fact composed of tools built using Docker. However, the ones described by Pahl lack many of the fundamental services for managing production applications, so it can be argued that the third generation platforms are not full production-ready solutions, but instead one of the many tools that are used to create a PaaS solution.

Flynn is the most promising solution from the ones outlined by Pahl; it is a reasonable solution for small production deployments such as proof of concepts with a reduced time to market. However, it is still relatively young and lacks some key features required for larger deployments [8]. For example, Flynn's logging feature allows a user to track logs and retrieve the last 10000 lines [9]; this is suitable for troubleshooting issues but missing some of the more advanced features for log aggregation and analyses. In general, this is an area that moves far too quick making it hard for solutions that aren't easily extendable to succeed. For instance, Deis barely had one stable release before it was replaced by Deis Workflow, which has since reached its end of life as of 03/01/2018 [10].

C. Using containers in production

From our research, we identified six key areas that must be taken into account when developing a container platform:

*1) **Infrastructure**:* One must first decide on the type of cloud for the underlying infrastructure. The available options are as follows [11]:

- Public cloud: Implemented using a public IaaS such as AWS (Amazon Web Services).
- Private cloud: Implemented using internal data centres.
- Hybrid cloud: Mix of public and private.

Keep in mind that the configuration and services provided vary across solutions making it harder to change approach in the future. Companies may use any of these approaches, given that an application does not require the services of a particular IaaS provider and there aren't any other business requirements that limit the possible choices.

To implement one of these approaches, one can use infrastructure as code (IAC) tools like Terraform or CloudFormation to provision infrastructure resources in addition to configuration management tools such as Chef and Ansible for configuring, managing and automating other setup processes.

Tools like these give users the ability to manage all aspects of infrastructure setup through code creating reproducible environments with clear advantages [12]. This significant shift in mindset follows through from the automation principles of DevOps [13]. However, it requires serious consideration and extended infrastructure design knowledge, for making choices on matters such as High Availability, Scalability and Networking.

*2) **Orchestration**:* With the increasing popularity of Docker containers, ways to automatically manage containers inside a cluster also started emerging. A container cluster is composed of several nodes or hosts (Virtual instance and Bare-metal Server) connected by the orchestrator (see Figure 1) [14]. Each node can run several services (containerised applications) simultaneously. The most popular ones are Swarm Mode (by Docker) and Kubernetes (originally designed by Google), with many others built on top of these appearing in recent years. The main responsibilities of an orchestrator are application deployment, scheduling and scaling across nodes, but they also offer many other features like volumes for data persistence and the ability to create logical network partitions of containers.

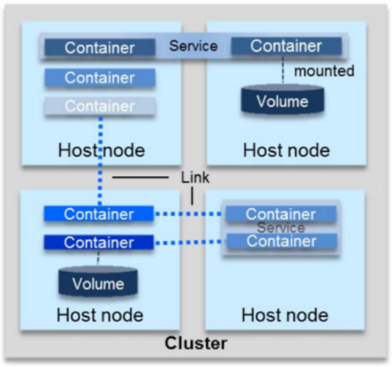


Figure 1. General cluster architecture [14].

Kubernetes is said to have the steeper learning curve and requires the expertise of experienced platform admins for setting up and managing the cluster. Swarm Mode is built into Docker and integrates with the native Docker tools that developers also use in their local development environments, making it more desirable for smaller companies. Kubernetes tends to be preferred by larger corporations because of built-in deployment tools for facing many of the challenges associated with managing a production cluster [15].

*3) **Networking**:* Traditionally, orchestrators exposed containers via the shared host machine's address [14]. Therefore all the network layering was implemented at the infrastructure level; if the host machine was publicly accessible, then all the containers in that host were also reachable. Though, in production environments, external traffic must not have access to applications that should only be accessible to services within the same private sub-network. Nowadays, orchestrators have built-in tools for creating smaller private networks, giving users the ability to partition the cluster into different container stacks. Because of business requirements or preference reasons, some platform teams prefer alternative tools such as Consul.

In addition to a way for containers to communicate with one another, it is recommended to use a Layer 7 load balancer also known as reverse proxy servers like HaProxy and Nginx. These tools operate in the top networking layer of the OSI model [16]; in addition to making load balancing decisions, they can be used to optimise and change content. Reverse proxy servers operate in the same layer as applications making it very easy to deploy as containers and a great candidate for performing other tasks such as SSL termination, compression and encryption of messages [17].

*4) **Data Storage and Back-up**:* Data storage and volume management is still a problem with container clusters. Without an adequate solution, nodes could malfunction and die resulting in loss of data forever. So a reliable backup/restore strategy is vital for running containerised applications in production environments; the approach selected will positively or negatively impact the PaaS customers.

Traditionally data containers were used (container with a mounted volume); these followed the app container to the physical machine regardless of its location in the cluster [14]. Local disk volumes are a good solution for local development where the loss of data should not have any significant consequences. Nowadays, there is a large variety of solutions for managing persistent storage in production environments. Usually, the options available depend on the orchestrator used but often involve the utilisation of some third-party solution like AWS' EBS Volume [18].

*5) **Continuous Integration and Continuous Delivery**:* Paul M. Duvall, Steve Matyas and Andrew Glover's definition of a simplified CI pipeline for software consists of four steps [19]:

1) Developer commits code to the version control repository. Meanwhile, the CI server keeps on continuously polling this repository for changes.
2) When the CI server detects new source code commits, it retrieves the latest copy from the code repository and then executes a build script, which integrates the software.
3) The CI server generates some feedback, logs presented by some GUI or via email; the data usually consists of some form of test results.
4) The CI server goes back to repeatedly polling for changes in the code repository.

Typically there would also be a post-stage (CD) for deploying applications onto a server. A delivery pipeline for containers is somewhat similar; it still makes use of a central code repository and a CI/CD server such as Jenkins or TravisCI. With the addition of some extra components, such as a Registry for storing images or an image scanning tool for identifying security vulnerabilities.

The DockerHub is excellent for distributing open source tooling, but unless we plan on releasing our Docker image to the world, we will also need to set up an internal Docker image repository for securing private Docker images, while still making them accessible to the build and deployment processes. There are plenty of tools for this out there, and once again it comes down to business requirements or preference, specialised purely for images and other general purpose ones such as Nexus that also works as a central repository for different types of binaries.

*6) **Central Logging and Monitoring**:* A good log management solution is not just useful for troubleshooting purposes; it also helps companies make decisions that offer enormous business value based on information such as the geo-identity of their user traffic data. There are several log management solutions, but many are costly or don't scale very well so careful consideration must be put into picking the one that best suits one's needs. A popular highly scalable open source solution is ELK [20]. The main components of ELK are Elasticsearch, Logstash and Kibana; together they form a handy tool for interactively analysing log files [21].

Monitoring is an essential part of cloud computing, even more so in container-based platforms, where services frequently move between nodes and the amount of tooling involved tends to be so large that it becomes impossible to monitor all of them manually (See Figure 2). There is a vast abundance of monitoring tools; some are container specific as is the case of Google's cAdvisor, other are general solutions such Nagios or a combination of Grafana (web-based front-end) and Prometheus (analytics and alerting).

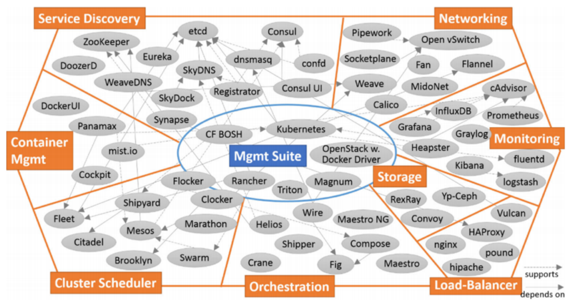


Figure 2. Partial view of the Docker ecosystem [22].

Implementing an adequate monitoring strategy will help minimise the impact of outages and maximise resources, resulting in happier customers. Monitoring should provide information about resource usage for the various system components and the ability to send alerts based on predefined thresholds. In some cases, a viable monitoring solution is also the basis for a pay-per-use scheme [22].

D. The Standardisation Problem

Most IT companies successfully using these technologies in production developed their custom solutions and processes to address their particular business requirements; however, smaller companies are finding it hard to implement similar solutions.

Some standards have started emerging in recent years; technologies such as Docker or Kubernetes slowly became de facto standards. There have also been some efforts from the IT industry, as is the case of the Open Container Initiative (OCI) the governance structure that creates open industry standards for container formats and runtime, which was established in 2015 by a group of companies seen as leaders in the container industry [23].

Despite the recent advancements in container standards, there has not been much work put into standardising the processes used and the way services integrate with each other. Since containers started gaining traction, the tool ecosystem has increased exponentially making the problem more prominent. See Figure 2 to get an idea of the number of tools that form a container PaaS. Running and connecting these services varies on a company basis, often involving large amounts of scripting. This lack of standardisation is making it difficult for new small IT companies to adopt similar approaches [24].

III. METHOD

A. General Introduction

PaaSPure's overall approach is to break down PaaS architectures into smaller reusable pieces known as PureObjects. The tool aims to help reduce the impact of the lack of standardisation in setting up platforms. This method will provide companies with a framework for sharing full platform specifications, including source code and the corresponding tests required for setting up different elements which could then be re-used and personalised by other users, significantly reducing the entry level barrier for smaller companies trying to adopt container technologies.

The tool makes use of Object-Oriented Principles such as Abstraction and Inheritance for ensuring PureObjects follow an expected pattern and common code reuse. We also identified code that was repeated across some but not all PureObjects and built a framework that further improves code reuse; we developed a set of general-purpose modules containing many useful features. For example one of these features is a method for deploying a service stack as this is missing from the Docker SDK (software development kit) for Python.

The tool breaks platform elements into two types of objects:

- **Modules**: General overview and requirements for things such as infrastructure, logging and monitoring. The application knows how to run modules and modules offer a layer of abstraction on top of components.
- **Components**: Are implementation specific, for example, a component for provisioning a platform's infrastructure on a public IaaS would be Terraform or CloudFormation.

Two companies may decide to use the same tools for a given task, for example, the provisioning of some resources on AWS using Terraform. The Terraform scripts may be company specific, but the work required for running Terraform in the first place is still the same; at present both would end-up writing somewhat similar code for running the desired approach. PaaSPure allows users to pull that code and reuse it to run their own Terraform scripts. The overall method is not specific to container platforms and could conceivably be extended to work with any type of platform.

B. Architecture

The framework consists of two main components:

- **PaaSPure CLI**: A Command Line Interface developed in Python for building platforms based on PureObjects.
- **PaaSPure Hub**: Composed of a REST web server using NodeJS plus ExpressJS and a user-friendly web application developed in Vue.js for managing PureObjects.

The Hub is the central location for storing and retrieving meta-data about PureObjects; The Hub lets users do things like creating new PureObjects, release new versions and search existing objects. To add a new PureObject to the Hub, the author must first create a new GitHub repository with the code (see Figure 3). In addition to being used to store PureObjects, GitHub is also used for third-party authentication to ensure that only the creator of an object can perform actions such as removing that object.

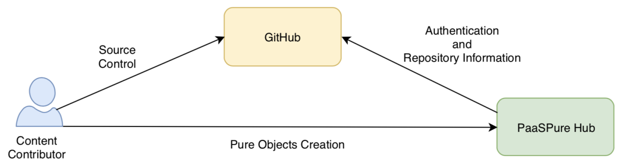


Figure 3. Architecture - Content creation.

At present all PureObjects are written using Python because it has a very mature community with an abundance of libraries that integrate with many other tools. We hope that in an open source world using a mature base language would reduce the contribution entry level. We designed the application so it would easily integrate with objects written in different languages. That way if we identify a more suitable language in the future it would be relatively easy to adopt it gradually.

The CLI is the primary form of interaction with the tool; potentially it could be used for the same purposes as the web app, but it can be quite intimidating at first for users not accustomed to using a terminal. To define a new platform the users must create a configuration file with the necessary information to pull and run the desired PureObjects; this will be discussed in more detail later on. Given a valid configuration file, the CLI connects to the Hub and fetches the required metadata for finding and cloning the code from GitHub. After pulling the source code onto the local machine, the CLI executes each module and child components in order as defined by the config file (see Figure 4).

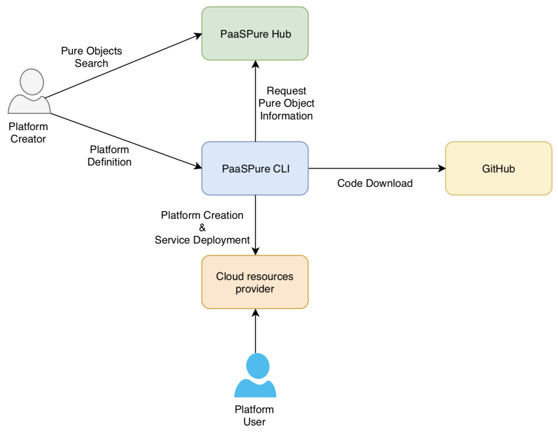
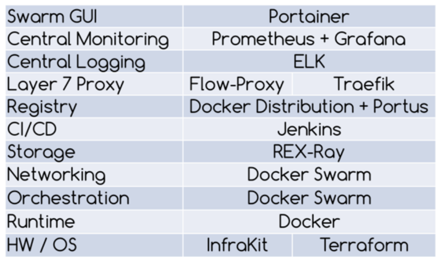


Figure 4. Architecture - Platform creation.

C. *Configuration*

The tool was used to build several distinct platforms such as the open source stack proposed by Bret Fisher at the official European Docker conference 2017, which also works as a FaaS (Functions as a Service) platform (see Figure 5) [25]. This particular stack is a good starting point for companies trying to adopt containers; it is composed entirely of free tools except for the IaaS infrastructure, and it satisfies the key areas previously discussed. To deploy any platform the first step is to create a configuration file called the Purefile; Figure 6 contains the corresponding PureFile for running the stack proposed by Bret on AWS.



Swarm GUI	Portainer	
Central Monitoring	Prometheus + Grafana	
Central Logging	ELK	
Layer 7 Proxy	Flow-Proxy	Traefik
Registry	Docker Distribution + Portus	
CI/CD	Jenkins	
Storage	REX-Ray	
Networking	Docker Swarm	
Orchestration	Docker Swarm	
Runtime	Docker	
HW / OS	InfraKit	Terraform

Figure 5. Open Source Tech Stack, Bret Fisher [25].

The ability to build all different kinds of platforms by mix and matching different solutions is one of PaaSPure's primary objectives. In the shown example we are using Terraform to invoke and pass in parameters to the Docker for AWS template which is in fact, a CloudFormation template [26]. Most of the elements that make up the platform are independent. Hence we can deploy the exact stack on any IaaS or internal datacentre; this can be achieved using almost all of the same code by merely replacing IaaS specific elements. For example, to use Microsoft's Azure instead of AWS the two components that need to be swapped out are `terraform_aws` and `swarm_aws`.

Nowadays, many companies can run their platform on different clouds using some internal solution, however PaaSPure allows different users to re-use a solution given that it has been added via the central hub. The proposed method would let users define their platform's requirements and in a matter of minutes deploy a working version enabling them to start delivering customer value straight away. The tool's high modularity makes it very easy to upgrade platforms allowing these to grow with the organisation.

For larger companies this would mean a faster development of new features and bug fixes, improving the platforms reliability. It should also reduce the time to market in cases where the company wants to test a different method applied by another company. In general, the collective power of an open source community rather than just one team within a company would allow us to build better platforms for everyone.

```
version: 1
hub: 'Overwrite default HUB'
credentials:
  private_key: /path/to/key.pem
  aws_access_key: access
  aws_secret_key: secret

modules:
  infra:
    tag: 0.1
    components:
      terraform_aws: (See available args in the documentation.)

  orchestrator:
    repo: repo_url
    commit: commit_hash
    components:
      swarm_aws: (See available args in the documentation.)

  networking:
    orchestrator: orchestrator
    components:
      traefik:

  logging:
    ...
      elk_stack:

  monitoring:
    ...
      pom_stack:

  deployer:
    ...
      registry:
      portainer:
      jenkins:
```

Figure 6. Sample config file.

D. *Versioning and execution*

We assume PureObjects will continuously change just like any other piece of software; hence backwards compatibility across all objects cannot be assured. To help avoid breaking changes we developed a versioning mechanism that provides users with the means for locking PureObjects to specific code versions.

The authors of PureObjects decide when to release new versions through their Hub accounts. The Hub maps each release to the hash value of git commits. It is good practice to lock every object to a specific version (see the infra module in the sample config); if no tag is selected it defaults to latest (GitHub's master branch). Users can bypass the tagging system by providing the hash for a specific commit (see the orchestrator module in the sample config), this allows the user to lock code versions in cases where there are no releases. Assuming there is a config file called "**pure.yml**" and that the CLI tool has been installed (using pip install paaspure or python install setup.py to install from source code); the steps for executing the tool are as follows:

1) **paaspure pull**: This command finds and downloads the code for each PureObject defined in the config file.
2) **paaspure build**: Execute PureObjects in the order defined in the config file.
3) **paaspure destroy**: Remove resources created during the build phase in the reverse order.

The CLI's argument parser was designed to be easily extended; every PaaSPure module should implement their argument parser that imports and extends the default. This feature allows the CLI tool to adapt to each user based on their config file because it does not need to know all the options in advance. For example, the output of running paaspure with the sample config shown above would be the following:

```
PaaSPure - Build the PaaS of the future.
Options:
  -f FILE, --file       Config file (Default 'pure.yml').
  -h, --help            Show help message and exit.
  ...
Commands:
    pull                Pull a module or component.
    build               Build all PureObjects.
    destroy             Destroy all PureObjects.
    infra               Setup cloud infrastructure.
    orchestrator        Connect to orchestrator.
    ...
```

Figure 7. Sample CLI help message.

Most Pure modules are self-sufficient so they can be executed individually (E.g. **paaspure `MODULE_NAME` build**), this allows for the extension of existing platforms without the need for being taken down. However, extending live platforms might not always be possible, in some cases platforms will have to be taken down to add new components.

IV. Experiments

A. User Evaluation

For our first experiment, we asked potential end-user to evaluate our tool; the aim was to gather users opinions early in the development process to validate the work already done and collect useful information that may benefit future work. The evaluation process consisted of:

1) A five-minute introduction to the research topic and the tool developed as part of the practicum.
2) A ten-minute hands-on session where we showed the software in action.
3) Afterwards, candidates had the opportunity to provide some feedback on the given feedback sheet.

Altogether, 43 users took part in the experiment, the participants consisted mostly of Full Stack Developers and DevOps Engineers. The questions and answers followed a Likert scale approach, and the results were grouped into six categories (see Figure 8).

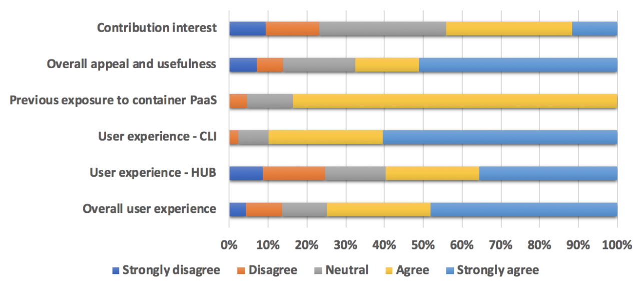


Figure 8. User evaluation results.

We previously cited that at present many companies are investing in container platforms; although some users might not directly interact with these platforms, we assumed most of the participants would have a basic understanding of what they are. We believe our assumption was correct as over 80% of the participants confirmed they had some form of exposure.

Overall the results show that 67% of participants reported the tool to be useful and they would use it if a stable version were released; in contrast, only 13% of participants believed the opposite.

The success of this kind of software often depends on its open source community, so we asked about the interest in contributing to the tool's development directly or via the creation of PureObjects. Over 40% of participants claimed they would like to contribute with 11% of these showing greater interest; in contrast, 23% of the respondents stated the opposite leaving around 33% unsure. Although the number of participants that strongly agreed was relatively small, we believe that initially a relatively small number of contributors would be required for substantially advancing the software.

The reality is that container platforms can take years and numerous iterations to perfect depending on a company's use case. Building a general solution without any business requirements such as the ones proposed by Bret Fisher should be much faster. Companies could then reuse and adapt the general solutions to their particular needs. We believe that given enough time more solutions for edge cases would appear as more people get involved.

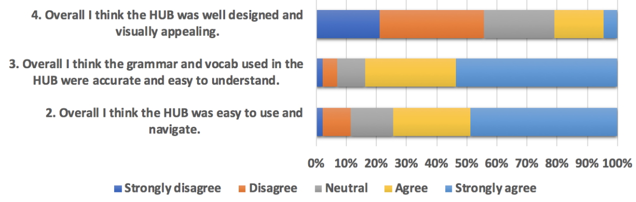


Figure 9. HUB - User experience results.

One of the primary goals of the user evaluation was to find out the user's opinion about the current state of the tool. Overall the participants found the tool easy to use and were satisfied with the user experience; the clear outlier was the

HUB's user experience. When further examining the results related to the HUB's user experience (see Figure 9), we found that although the users found the HUB easy to use and understand, more than 50% did not particularly like its design nor found it visually appealing.

From the results, we were able to gather insights on what the participants found useful and areas for improvement. Thanks to the questions and feature request section we were able to gather particular features that the users would like to see on the tool and areas that may need further explanation.

B. Scaling

The second set of experiments focuses on testing the scalability and response time of individual services and the cluster as a whole. We decided to use Andrea Luzzardi experiments as a base [27]. Andrea's work dates back to November 2015, which is a significant period in this particular area; in fact, it is more than half of Docker's lifetime, so we expect some improvements and changes to have occurred meantime. For example, Swarm Mode (docker's built-in orchestrator) became the default standard with Docker Engine version 1.12.0 (2016-07-28) [28]; this means Andrea must have been using the standalone version of Swarm.

Although we also used AWS and could have built a platform using the standalone version of Docker, we believe that was not a valid approach since the recommendation nowadays is to use the built-in version. Also, we are not trying to replicate Andrea's work; instead, we want to use some of his methods and expand on it such as is the case of using the command "docker info" to measure the API response time.

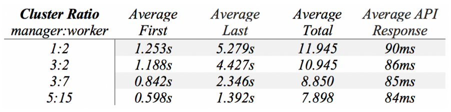
Another distinction is the fact that in the original experiments 1000 worker nodes were used to stress test one single manager, we decided to simulate smaller more realistic cluster sizes; the cluster sizes used were 3, 5, 10 and 20 nodes with the ratio of managers and workers of 1:2, 3:2, 3:7 and 5:15 respectively.

Because real life systems may have other services already running, we decided to add a new system load variable. We used a tool called stress-ng [29]; a Linux command line tool for the simulation of different system loads. Because it is a Linux tool, it is straightforward to containerise and can be deployed as a service and just like any other containerised application; Figure 10 contains the command to deploy the stress application as a service. The number of stress replicas can be adjusted using the "–replicas" flag, for example for a cluster of 30 (1 manager : 29 workers), setting the number of replicas equal to 30 will place one replica per node.

```
docker service create --name stress stress-ng --cpu 1 \
  --io 1 --vm 1 --hdd 1 --vm-bytes 64m --hdd-bytes 128m
```

Figure 10. Deploy stress service.

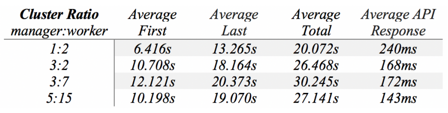
*1) **Service Creation and Scaling**:* We started off with a simple experiment to measure service creation and scaling time; we timed how long it took to create a service consisting of 30 replicas; this was done using the "docker service" command which is a Swarm mode only feature. We used docker info to measure API response time, and we performed a number of tests with the clusters under different loads.



Cluster Ratio manager:worker	Average First	Average Last	Average Total	Average API Response
1:2	1.253s	5.279s	11.945	90ms
3:2	1.188s	4.427s	10.945	86ms
3:7	0.842s	2.346s	8.850	85ms
5:15	0.598s	1.392s	7.898	84ms

Figure 11. Single service scaling.

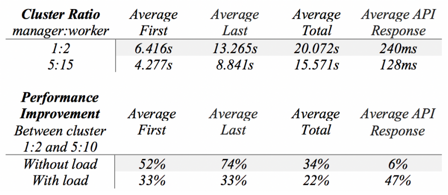
The service create command was executed against each cluster 1000 times; we stored the averages for the API response time, the time it took for the first and last replicas to start and the total time the command took to complete (see Figure 11). We then repeated the process, but this time we also deployed a stress replica per node (see Figure 12). We expected the results to follow a similar pattern to the ones without stress because we increased the stress at the same rate as the clusters size (1 stress replica per node), but this was not the case. We believe our assumptions were incorrect because we had not taken into account that extra nodes and extra stress replicas also meant the orchestrator was doing more work in order to schedule and deploy the same number of test replicas. We believe that when the orchestrator is under stress, the speedup improvements of larger clusters are decreased.



Cluster Ratio manager:worker	Average First	Average Last	Average Total	Average API Response
1:2	6.416s	13.265s	20.072s	240ms
3:2	10.708s	18.164s	26.468s	168ms
3:7	12.121s	20.373s	30.245s	172ms
5:15	10.198s	19.070s	27.141s	143ms

Figure 12. Single service scaling with stress.

To validate our theory, we decided to repeat the tests but this time with the same load exerted on each cluster. Figure 13 contains the identical stress results and a direct comparison between the improvements observed between a cluster of ratio 1:2 and another of 5:15. Even though both clusters had the same number of stress and test replicas, the variation in performance improvements with and without stress show that having more resources helped reduce average times; however, that improvement was diminished under stress.



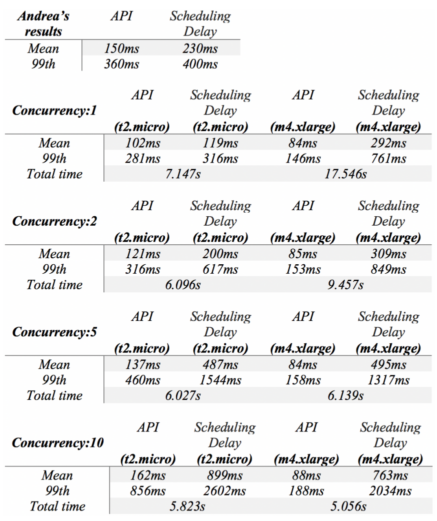
Cluster Ratio manager:worker	Average First	Average Last	Average Total	Average API Response
1:2	6.416s	13.265s	20.072s	240ms
5:15	4.277s	8.841s	15.571s	128ms

Performance Improvement Between cluster 1:2 and 5:10	Average First	Average Last	Average Total	Average API Response
Without load	52%	74%	34%	6%
With load	33%	33%	22%	47%

Figure 13. Identical stress results and performance improvements.

In contrast, the API response improved significantly, which indicates that the smaller clusters were finding it hard to cope with the load exerted on them (each of the smaller cluster's nodes had to run ten test service replicas plus an extra stress replica, 33 containers in total). While in comparison the larger cluster could easily handle the same (33 containers in total), but the orchestrator cannot necessarily keep up and starts becoming a bottleneck.

*2) **API Responsiveness and Scheduling:*** Using Andrea's swarm-bench method, we tested the API response times and Scheduling delays. Andrea's results do not mention the type of concurrency used so we tried different values to see which would be closer to his results. Initially, we planned on using the powers of two as the currency values, but we found an issue with Andrea's swarm-bench that required the concurrency value to be a divisor of the total number of replicas; otherwise it would not run the expected number of containers.



Andrea's results	API	Scheduling Delay
Mean	150ms	230ms
99th	360ms	400ms

Concurrency:1	API **(t2.micro)**	Scheduling Delay **(t2.micro)**	API **(m4.xlarge)**	Scheduling Delay **(m4.xlarge)**
Mean	102ms	119ms	84ms	292ms
99th	281ms	316ms	146ms	761ms
Total time	7.147s		17.546s	

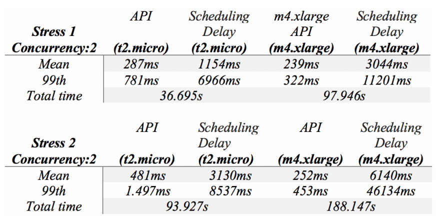
Concurrency:2	API **(t2.micro)**	Scheduling Delay **(t2.micro)**	API **(m4.xlarge)**	Scheduling Delay **(m4.xlarge)**
Mean	121ms	200ms	85ms	309ms
99th	316ms	617ms	153ms	849ms
Total time	6.096s		9.457s	

Concurrency:5	API **(t2.micro)**	Scheduling Delay **(t2.micro)**	API **(m4.xlarge)**	Scheduling Delay **(m4.xlarge)**
Mean	137ms	487ms	84ms	495ms
99th	460ms	1544ms	158ms	1317ms
Total time	6.027s		6.139s	

Concurrency:10	API **(t2.micro)**	Scheduling Delay **(t2.micro)**	API **(m4.xlarge)**	Scheduling Delay **(m4.xlarge)**
Mean	162ms	899ms	88ms	763ms
99th	856ms	2602ms	188ms	2034ms
Total time	5.823s		5.056s	

Figure 14. API Response and Scheduling Delay results.

Andrea claimed in his findings that 99% of the containers took less than half a second to start. From our results, we believe he must have ran one container at a time, as these were the only ones that came close to his results (see Figure 14). We decided to test different types of instances because we noticed something odd. From the results, it appears that it takes less time to schedule a container on a t2.micro instance although the API response times were higher, which is a side effect of having fewer resources.



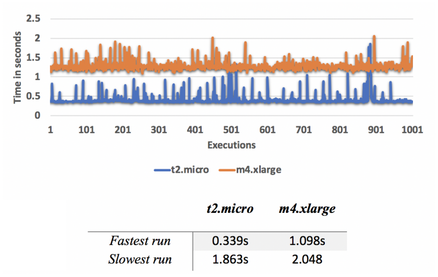
Stress 1 Concurrency:2	API **(t2.micro)**	Scheduling Delay **(t2.micro)**	m4.xlarge API **(m4.xlarge)**	Scheduling Delay **(m4.xlarge)**
Mean	287ms	1154ms	239ms	3044ms
99th	781ms	6966ms	322ms	11201ms
Total time	36.695s		97.946s	

Stress 2 Concurrency:2	API **(t2.micro)**	Scheduling Delay **(t2.micro)**	API **(m4.xlarge)**	Scheduling Delay **(m4.xlarge)**
Mean	481ms	3130ms	252ms	6140ms
99th	1.497ms	8537ms	453ms	46134ms
Total time	93.927s		188.147s	

Figure 15. API Response and Scheduling Delay stress results.

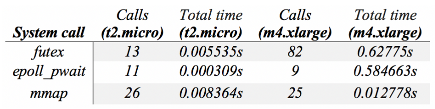
The scheduling results were not what we expected but they are too close to be sure, so we decided to rerun the tests; this time around we put the nodes under different levels of stress. Figure 15 contains the stress results with two concurrent requests (other levels of concurrency followed a similar pattern); under stress, it became evident that it takes longer to schedule containers in the more powerful m4.xlarge machines. We tried to scale the number of stress containers further but the API became very unresponsive, and we started noticing spurious node failures.

The results were counterintuitive, so to ensure our experiments were not faulty we ran a different experiment that involved recording the time to start a single container, and we repeated the process 1000 times (See Figure 16); this produced similar results which proves our experiments are working accordingly. So we dug a bit deeper using strace [30]; we found that almost all system calls took less time in the better machine except for the ones in Figure 17.



	t2.micro	*m4.xlarge*
Fastest run	0.339s	1.098s
Slowest run	1.863s	2.048

Figure 16. Container run comparison.



System call	Calls **(t2.micro)**	Total time **(t2.micro)**	Calls **(m4.xlarge)**	Total time **(m4.xlarge)**
futex	13	0.005535s	82	0.62775s
epoll_pwait	11	0.000309s	9	0.584663s
mmap	26	0.008364s	25	0.012778s

Figure 17. Container run comparison.

We further examined the two main culprits, the `epoll_wait` and `futex` syscalls. The `futex` system call is generally used by threading implementations to implement high-level locking primitives such as mutexes and semaphores [31]. The `epoll_pwait` system call is similar to `epoll_wait` but with the addition of a sigmask argument (a pointer to a signal mask); it is used by applications to wait until a file descriptor is ready or for a particular signal to be caught [32].

During our experiments, we noticed a recurrent pattern where two processes seemed to be blocking each other while invoking the `epoll_wait` and `futex` syscalls; we believe this would have caused the `futex` syscall count to increase as seen in Figure 17. This pattern led us to believe that the odd behaviour was related to the underlying Operating system. Although containers isolate processes, these processes share the same kernel and compete for the same resource; so the underlying kernel has a significant impact on how containers behave.

In our previous experiments, we used the default Docker for AWS AMI based on Alpine Linux; to prove our theory, we reran the tests on new AMI we built using a PureObject we created with Packer. This time around the API and scheduling results were much lower on the more powerful machines, this proved that the default Docker for AWS AMI was at fault. We believe the issue was not apparent in the t2.micro machines because they only had one CPU.

*3) **Cluster Scaling**:* In addition to the scaling experiments already discussed we also examined cluster scaling; for this exercise, we created a two-node cluster, one manager and one worker; next, we scaled the workers one by one up to 100 while tracking the execution time (workers used t2.micro instance). On average it took 1 minute and 36.93 seconds to scale the number of workers up by one. The difference between the fastest run and slowest was less than 60 seconds.

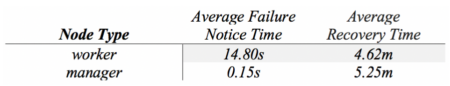
These results show that virtual machines can also be quite small and have extremely fast boot up times. There may be some performance benefits of using containers instead of VMs as an application running environment, but it is possible to build platforms without containers that can be just as resource and time efficient. We believe that the deciding factors for implementing a container platform instead of an alternative solution is not solely their size and speed but instead how easy they are to create, configure, deploy and manage thanks to cluster orchestrators. It is also not a matter of choosing between containers and VMs because virtual machines are the basis of most container clusters.

C. Failure Recovery

We decided to produce different kinds of failures in a controlled manner in our cluster to examine how long it took to recover. We started off by timing how long it took the system to recover from a manager failure, node failure and service failure single container and multiple. We ran the tests on different size clusters with managers and workers running on instances of type m4.xlarge and t2.micro respectively. It is import to note the difference between High Availability and Automatic Fault Recovery:

- **High Availability**: Aims to achieve zero downtime; this requires node and service redundancy making it harder to implement.
- **Automatic Failure Recovery**: As the name suggests its the act of recovering from an unexpected failure. Node recovery can be implemented using IaaS tools such as AWS Auto Scaling, while service recovery is handled out of the box by mature orchestrators like Swarm Mode and Kubernetes.

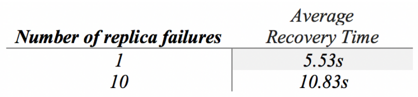
As seen from the results in Figure 18 it took less time for the orchestrator to notice that a manager had gone offline. The reason for this is because manager nodes are responsible for managing, and reporting on the life state of the Swarm and are in constant communication. The Raft Consensus Algorithm is used amongst the managers for making decisions on how to manage the swarm state [33]. Raft tolerates up to (N-1/2) failures where N is the number of managers and requires a majority of (N/2)+1 to agree on decisions; when the number goes below this point, the following error is shown ”The swarm does not have a leader. It’s possible that too few managers are online ...”.



Node Type	*Average Failure Notice Time*	*Average Recovery Time*
worker	*14.80s*	*4.62m*
manager	*0.15s*	*5.25m*

Figure 18. Node Failure recovery.

The actual bootup times between the manager and node works was relatively small with the managers taking a bit extra on average because machines with more resources tend to take longer to be provisioned. It is worth noting that this heavily depends on the solution used for node recovery (in this case it is AWS Auto Scaling).



Number of replica failures	*Average Recovery Time*
1	*5.53s*
10	*10.83s*

Figure 19. Service Failure recovery.

During our service failure experiments, we induce failures on different numbers of replicas across the whole cluster. As seen in the results from Figure 19, it took swarm very little time to notice the failure and redeploy a replacement, taking only around 10 seconds to recover from the simultaneous failure of 10 replicas. An interesting behaviour we noticed during our experiments was that after a short successive number of service failures the swarm tried to schedule these on a different node. Putting the cluster under stress did not reveal any unexpected behaviours we merely noticed the times growing exponentially.

V. Conclusions and Recommendations

Nowadays, containers are primarily adopted for PaaS clouds and although it is a relatively new area it has a considerable potential to further advance the PaaS cloud [13]. Despite that, some improvements are required, mainly in the areas of process and tool standardisation. There is a considerable number of applications for dealing with different requirements; setting these up as part of a PaaS solution is often company specific and usually involves copious amounts of scripting. The lack of standard processes and tools used is making it difficult for smaller companies to adopt container technologies, besides it makes these platforms challenging to test.

Vendor lock-in is still an issue, although it has been disguised by this misconception that once it uses containers, it can run anywhere; although containerized applications using Docker are portable across different Docker-based platforms, the platform themselves are often very complex systems directly tied to the underlying infrastructure or tooling used making it very vendor dependent. Due to fear of vendor lock-in or because of specific business requirements, companies will continue to work on developing their container platforms, by using other technologies or implementing their solutions to address many of the challenges they face on a day to day basis. From our user evaluation results we believe that a tool like the one developed in this practicum could significantly reduce the entry level barrier for companies and individuals trying to adopt these tools; but this sort of tool will require the collaboration of larger companies and engineers with the knowledge and experience of running these sort of systems in secure production environments.

We also believe there is an excellent opportunity for the academic world to get involved and help further advance these platforms by providing the general public with the right set of tools for making sensible decisions. From our experiments we showed that container platforms are highly scalable and efficient, producing results that should give users thinking of adopting Docker + Swarm a good idea of different cluster capabilities such as scaling and failure recovery; although there is other great academic work done in this area, much of it covers theoretical approaches or related topics at a very high level.

We believe there is a need for more in-depth research work that covers the sort of topics faced by people trying to implement container platforms, for example, we think it would be very beneficial to have individual researches done on the critical areas described earlier on in addition to direct comparisons between existing tooling. This area has ample content on the web, but can sometimes be hard to find. Besides in many cases, it is not well structured or explained; making it hard to reproduce, unlike published research papers. More research work in this area will also allow us to gain more significant insights into the current capabilities of containers, and subsequently, help identify problems like the one we found where containers took longer to start in some instances that used the default Docker for AWS AMI.

One of the issues researchers face is the pace at which this area has been evolving in recent years; the number of changes that happen in periods as short as six months to a year is enormous meaning that research could become obsolete shortly after their publication. To avoid this, it is required that the academic world and IT industry work closely together, potentially even getting people like the Docker Captains and Campus Ambassadors to take an active role in research projects.

REFERENCES

[1] P. Clarke, R. O'Connor and P. Elger. (January 2017). *Technology Enabled Continuous Software Development*, 2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery, IEEE.

[2] Docker. [Online]. Available: https://www.docker.com/ (accessed January 29, 2018)

[3] W. Vogels. (January/February 2008). *Beyond Server Consolidation*, Queue Virtualization Volume 6 Issue 1. ACM New York, NY, USA.

[4] Intel IT Center. (August 2013). *Virtualization and cloud computing*, Intel.

[5] R. Smith. (2017) *Docker Orchestration a concise, fast-paced guide to orchestrating and deploying scalable services with Docker*, Packt Publishing.

[6] C. Pahl. (July 2015). *Containerization and the PaaS Cloud*, IEEE Cloud Computing Volume: 2, Issue: 3. IEEE.

[7] Zhiyong Shan, Xin Wang, Tzi-cker Chiueh and Xiaofeng Meng. (2017). *Safe Side Effects Commitment for OS-Level Virtualization*, ICAC '11 Proceedings of the 8th ACM international conference on Autonomic computing. ACM New York, NY, USA.

[8] Thibaut Gery. *Discovering Flynn*, [Online]. Available: https://blog.octo.com/en/discovering-flynn/ (accessed March 6, 2018)

[9] Flynn. [Online]. Available: https://flynn.io/docs/production (accessed March 6, 2018)

[10] Deis. [Online]. Available: https://github.com/deis/workflow (accessed March 7, 2018)

[11] Y. Jadeja and K. Modi. (August 2015). *Container-based orchestration in cloud: state of the art and challenges*, 2012 International Conference on Computing, Electronics and Electrical Technologies, IEEE.

[12] Stelligent. *Devops Benefits of Infrastructure as Code*. [Online]. Available: https://stelligent.com/2017/06/29/devops-benefits-of-infrastructure-as-code/ (accessed March 10, 2018)

[13] B. Yevgeniy. *Terraform: Up and Running by Yevgeniy Brikman*. [Online]. Available: https://www.safaribooksonline.com/library/view/terraform-up-and/9781491977071/ch01.html (accessed March 10, 2018)

[14] C. Pahl and B. Lee (August 2015). *Containers and Clusters for Edge Cloud Architectures – A Technology Review*, 2015 3rd International Conference on Future Internet of Things and Cloud, IEEE.

[15] Platform9. *Kubernetes vs Docker Swarm* [Online]. Available: https://platform9.com/blog/kubernetes-docker-swarm-compared/ (accessed February 2, 2018)

[16] N. Briscoe (2000). *Understanding the OSI 7-Layer Model*, PC Network Advisor, 120 (July 2000), pp.13–14.

[17] NGINX. *What is layer 7 load balancing.* [Online]. Available: https://www.nginx.com/resources/glossary/layer-7-load-balancing/ (accessed July 6, 2018)

[18] AWS. *Amazon Elastic Block Store* [Online]. Available: https://aws.amazon.com/ebs/ (accessed July 6, 2018)

[19] P. M. Duvall, S. Matyas, and A. Glover (2007). *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*, ISBN: 0321336380. Addison-Wesley Professional.

[20] Elastic. *ELK Stack* [Online]. Available: https://www.elastic.co/elk-stack (accessed July 6, 2018)

[21] T. Prakash, M. Kakkar and K. Patel. (2016). *Geo-Identification of Web Users through Logs using ELK Stack*, 2016 6th International Conference - Cloud System and Big Data Engineering. IEEE.

[22] R. Peinl, F. Holzschuher and F. Pfitzer. (2016). *Docker Cluster Management for the Cloud - Survey Results and Own Solution*, Journal Grid Computing Volume 14. Springer-Verlag New York, Inc.

[23] Open Containers Initiative (OCI). [Online]. Available: https://www.opencontainers.org/ (accessed February 6, 2018)

[24] A. Tosatto, P. Ruiu and A. Attanasio. (August 2015). *Container-based orchestration in cloud: state of the art and challenges*, 2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems, IEEE.

[25] B. Fisher. *Going production with docker and swarm.* [Online]. Available: https://speakerdeck.com/bretfisher/going-production-with-docker-and-swarm (accessed February 6, 2018)

[26] Docker. *Docker for AWS.* [Online]. Available: https://docs.docker.com/docker-for-aws/ (accessed February 6, 2018)

[27] A. Luzzardi. (November 2015). *Scale testing Docker Swarm to 30,000 containers.* [Online]. Available: https://blog.docker.com/2015/11/scale-testing-docker-swarm-30000-containers/ (accessed July 9, 2018)

[28] Docker. *Docker Engine release notes* [Online]. Available: https://docs.docker.com/release-notes/docker-engine/ (accessed July 9, 2018)

[29] Ubuntu. *Stress-ng* [Online]. Available: http://manpages.ubuntu.com/manpages/xenial/man1/stress-ng.1.html (accessed July 25, 2018)

[30] *strace(1) - Linux manual page.* [Online]. Available: https://linux.die.net/man/1/strace (accessed July 25, 2018)

[31] *futex(2) - Linux manual page.* [Online]. Available: http://man7.org/linux/man-pages/man2/futex.2.html (accessed July 25, 2018)

[32] `epoll_wait`*(2) - Linux manual page.* [Online]. Available: http://man7.org/linux/man-pages/man2/epoll_wait.2.html (accessed July 25, 2018)

[33] D. Ongaro and J. Ousterhout. (August 2015). *In Search of an Understandable Consensus Algorithm*, Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference, USENIX.