mira: an Application Containerisation Pipeline for Small Software Development Teams in Low Resource Settings

Alex Mwotil, Engineer Bainomugisha
{alex.mwotil,baino}@mak.ac.ug
Makerere University
Kampala, Uganda

Stephen G.M. Araka
garaka@andrew.cmu.edu
Carnegie Mellon University Africa
Kigali, Rwanda

ABSTRACT

Cloud native applications leverage Development and Operation (DevOps), microservice architectures and containerisation for primarily availability, resilience and scalability reasons. Small developer teams in low resource settings have unique DevOps needs and harnessing its principles and practices is technically challenging and distinctly difficult in these contexts. We conducted a survey with professional developers, students and researchers situated and working in a low resource setting and the results indicate that these principles and practices are relatively new. In application containerisation, an operating system virtualisation method that can significantly optimize the use of computing resources, the respondents indicated challenges in the process steps. Particularly, small developer teams in low resource settings require custom tools and abstractions for software development and delivery automation. Informed by the developer needs, we designed and developed a custom automated containerisation pipeline, *mira*, for a managed cloud native platform situated in a low-resource setting. We validate *mira* against 6 major application frameworks, tools and/or languages and successful deployment of the resultant applications onto a cloud native platform.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

containers, automation, cloud, orchestration, cloud native, docker

ACM Reference Format:
Alex Mwotil, Engineer Bainomugisha and Stephen G.M. Araka. 2022. *mira*: an Application Containerisation Pipeline for Small Software Development Teams in Low Resource Settings. In *Federated Africa and Middle East Conference on Software Engineering (FAMECSE '22), June 7–8, 2022, Cairo-Kampala, Egypt*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3531056.3542769

1 INTRODUCTION

Application properties such as scalability, availability, portability, security and management are increasingly becoming essential in cloud-supported deployments. This has driven the need to invest in new tools to support modern software development and delivery. Previously, legacy monolithic applications would be run on Virtual Machines (VMs) in the Infrastructure as a Service (IaaS) cloud category. This has limitations such as difficulties in enforcing effective application scalability and monitoring, resource wastage as the VM resources are usually overly reserved, cloud vendor lock-in and application reliability constraints due to tight coupling with the infrastructure. In order to achieve resiliency and elasticity (scalability) [15], cloud native is now a recurrent term in application development lexicon and mainly encompasses areas of microservices, containerisation, orchestration and DevOps (development (Dev) and IT operations (Ops)). Containerisation coupled with related DevOps practices and principles is revolutionising the development and delivery of modern software.

The transition to DevOps is not clear-cut and requires a technical and cultural shift characterized by different hurdles as per the context and environment. Furthermore, there is need for significant investment in time, effort and resources. Small developer teams in low resource settings have unique challenges and needs that can be demanding to ensure seamless software development and delivery. A software development project, for example, may not have precise roles and team numbers to decompose an application into microservices and subsequently develop and package them. In as much as DevOps agitates for similar team skill sets, small developer teams do not have the principle prerogative to function desirably. The DevOps principles and practices are also relatively new in these environments. We provide more context to the problem in Section 4. To bridge the gap, there is need for appropriate tools and abstractions at different stages of the CI/CD pipeline. Most research on DevOps is focused on its definition [1] [2], practices [18], culture [17], value proposition [5] and factors affecting its adoption [6]. There are no clear statistics on understanding, use and adoption within different environments and especially small developer teams in low resource settings.

The aim of this paper is twofold. Firstly, to present the state of DevOps and cloud native application development in small developer teams situated in low resource settings. Secondly, we advance **mira**, a containerisation pipeline for a cloud native application deployment platform, Crane Cloud [1]. The rest of the paper is organized as follows: Section 2 describes concepts, Section 3 reviews the related work, Section 4 presents the problem, the level of awareness of DevOps practices and the DevOps needs for small software development teams in low resource settings, Section 5 presents a containerisation platform for small software development teams

[1] https://cranecloud.io/

FAMECSE '22, June 7–8, 2022, Cairo-Kampala, Egypt

ACM ISBN 978-1-4503-9663-9/22/06…$15.00
https://doi.org/10.1145/3531056.3542769

in low resource settings, Section 6 details the validation of the platform, Section 7 provides limitations of the study and Section 8 concludes and provides future direction of evolving works in this research.

2 CONCEPTS

In this paper we adopt Cloud Native Computing Foundation (CNCF)'s [2] definition of cloud native as a software stack that is containerised, dynamically orchestrated and microservice oriented. The applications are decomposed into microservices which are containerised (run as containers) and are orchestrated by the service fabric. In orchestration, the service fabric monitors, maintains and scales the applications. At the application development and deployment level, developers require appropriate abstractions to efficiently package and deploy microservices onto a cloud infrastructure. Recently, containers have emerged as a novel abstraction to deploy microservices as opposed to traditional virtual machine approaches. Containers form the basic deployment units to release and ship microservices [10] given their modular design which closely maps with the functional decomposition of a complex business application. The lightweight nature of containerised applications coupled with functional and configuration encapsulation facilitates replication and portability, are cost-efficient and have a reduced overhead on the operation and maintenance line. It is for these reasons that containers have emerged as the most suitable packaging toolset for microservices.

A container is a lighweight "virtual machine" based on the Linux Kernel Extension (LXC) [3] *technology that allows running of many (up to hundreds) isolated and autonomous Linux environments on a single server or virtual machine.* It is a collection of communicating components such as application code, runtime, system tools, system libraries and settings required to run an application. In comparison with virtual machines, containers consume less computing resources and take the least provisioning time because virtual machines require different instances of the operating system (guest OS) while containers use the same host operating system as shown in Figure 1. CoreOS rkt [4], Mesos Containerizer [5], LXC Linux Containers, OpenVZ [6] and containerd [7] are examples of containerisation technologies but Docker [8] is by far the most popular.

In DevOps, the software development and operation teams collaboratively work to deliver a high quality software product with agility and velocity while overcoming communication and integration challenges. There is no clear distinction between the teams hence both have more or less the same skill-set required to move a product from requirements to production and the selection of tools to support the different processes is entirely dependent on skills of the team members. It is a culture of business practices, ideas, processes, tools, products, associated technologies, organizational structures, and opportunities that streamline the software product development [18] culminating in a product with richer features and less bugs [8]. Innovative products and features are brought faster

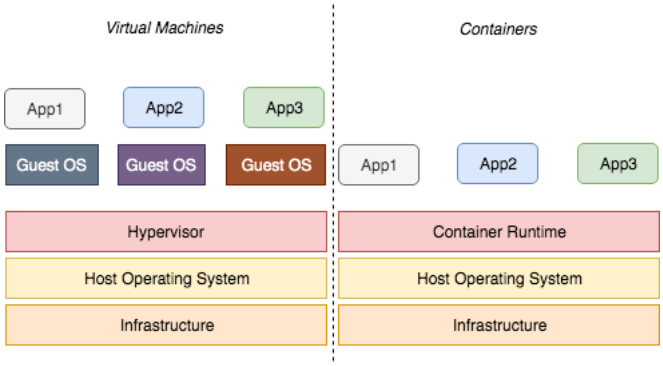

Figure 1: Difference between Virtual Machines and Containers

to the market with a 10% to 30% improvement in the requirements to product evolution lifecycle time and up to 20% reduction in production costs [2].
There are two key DevOps practices that form the CI/CD pipeline:

- **Continuous Integration (CI):** The team members frequently check code into a controlled source code repository, integrate and test their works before producing the master and stable release [3]. The integrations are verified by automated build and testing processes (unit, integration, system and user acceptance).
- **Continuous Deployment (CD)**: CD extends on CI by introducing an automated deployment process to the product release ensuring that there is always a production-ready copy of the application. CD ensures that changes are committed and deployed to production through a deployment pipeline [11].

3 RELATED WORK

Sokolowski *et al.* proposed *'muse'*, an Infrastructure as Code (IaC) system to automate DevOps deployments in serverless computing environments [12]. *'muse'* provides for automated coordination of service deployments for system-dependant components. *mira* is specific to single application instances and subsequent deployment to a container orchestration platform. Mysari *et al.* presented an automated CI/CD pipeline using Jenkins Ansible for single application deployments [9]. Other supporting CI/CD tools that include Github actions [9] operate comparably. These require considerable customization effort for different programming stacks. In addition, the pipelines deploy to manually configured node(s) that require terminal access. *mira* works in a multi-user setup and deployments are automated to a cluster of nodes. Other more specific use-case CI/CD pipelines include Kubernetes [7], 5G [16] and Robotics [13]. *mira* is not a replacement but rather a 2-stage CI/CD pipeline which is an integral component for container-based deployments in low resource settings. It can work with different use-cases but can also be adapted/extended if required.

[2] https://www.cncf.io/
[3] https://linuxcontainers.org
[4] https://coreos.com/rkt/
[5] http://mesos.apache.org/documentation/latest/containerizers/
[6] https://openvz.orghttps://openvz.org
[7] https://containerd.io
[8] https://www.docker.com
[9] https://github.com/features/actions

4 DEVOPS NEEDS FOR SMALL DEVELOPER TEAMS IN LOW-RESOURCE SETTINGS

We argue that small developer teams situated in low-resource settings have unique DevOps needs that require custom tool sets and abstractions for software deployment and delivery. In low resource settings, the teams are very limited and there is no precise separation in the roles. Also containerisation and the related principles and practices are relatively new to these environments. To concretise our claim, we undertake a study with selection of professional developers, students and researchers practicing software development in these settings. We set out to answer the research questions: *RQ1: What is the level of awareness about DevOps practices and tools among software development teams in low-resource settings?* and *RQ2: What are the unique DevOps needs for software development teams in low-resource settings?* To answer these questions we undertook a self-administered online questionnaire survey with professional developers, students and researchers situated and working in the low resource setting. A total of 35 responses were received with the participant demographics shown in the Figure 2 and their development tools/languages/frameworks in Figure 3.

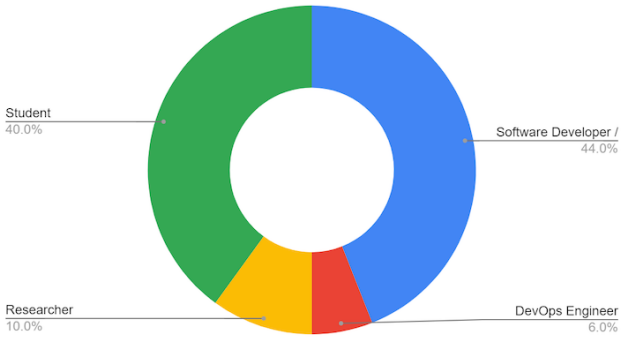

Figure 2: Participant Demographics

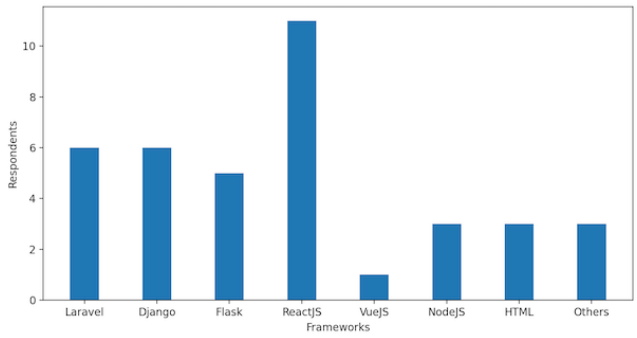

Figure 3: Participant Programming Languages / Frameworks

In particular, the survey sought to understand:

- the respondents knowledge of cloud native development, concepts and technologies
- containerisation approaches used by the respondents
- relevance of automation in the containerisation process

4.1 RQ1: What is the level of awareness about DevOps practices and tools among software development teams in low-resource settings?

Cloud native applications complement the DevOps processes and automation. DevOps principles and practices can drive organizations to adopt cloud native application development. A working knowledge of these concepts is therefore integral to understanding adoption and use in low resource settings. From the survey, less than 77% of the respondents had basic knowledge of the three concepts (microservices, containerisation and DevOps) while 23% of the respondents had no prior knowledge of these concepts as shown in Figure 4. This indicates that cloud native is a relatively new concept in low resource settings. This also reveals possible popularity of monolithic application development - an architectural style with a large and continuously increasing codebase that can suffer from potential performance issues. This eventually creates operational overhead and resource wastage and makes the entire codebase update process hard, time consuming and prone to error.

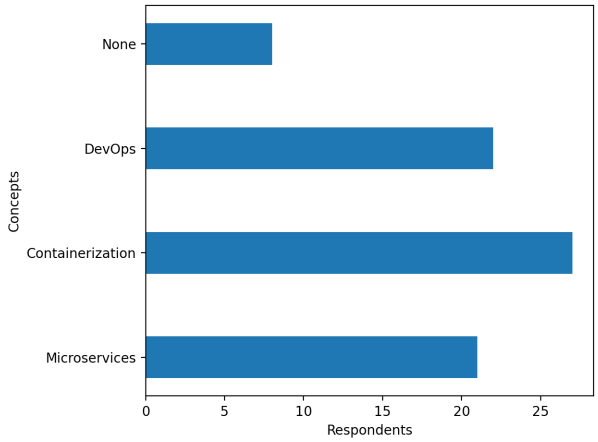

Figure 4: Cloud native Concepts

To further reinforce this conclusion, the respondents were exposed to Crane Cloud with the necessary documentation [10] to deploy an application. The Crane Cloud platform is hinged particularly on application containerisation (using Docker) and orchestration (using Kubernetes). 74% of the respondents have used Docker before and only 43% of the respondents were able to successfully deploy an application with relative ease. The main hindrance was reported as the inability of the respondents to containerise an application and push it to a registry which is a Crane Cloud deployment requirement.

[10] https://docs.cranecloud.io/

4.2 RQ2: What are the unique DevOps needs for software development teams in low-resource settings?

Software development teams in low resource settings are small and usually have cross-cutting responsibilities due to time, budget and personnel constraints. In some cases, organizations can opt to fly in consultants to design, develop, install and maintain software due to political and perception dimensions. This can negatively impact the DevOps culture and practices and ultimate delivery of software in these settings. International software development events such as KubeCon [11], a flagship cloud native event, currently attracts over 20,000 participants with a meagre representation from developing sects despite efforts for improvement as part of diversity and inclusion programmes. The event locations (Europe, Asia and America) have not helped the cause as travel costs are usually high. In as much as awareness is a key element in adoption and use of technology, developer teams in low resource settings require appropriate tools in the the software development and delivery life-cycle. It should be noted that most of the automation tools provide a subset of free features and the full packages are quite expensive for the teams.

As highlighted in Section 4, containerisation is one of the barriers to application deployment in cloud native setups. The containerisation process is normally dictated by the application requirements and can be complex with introduction of configuration methods, logging and persistence. In low resource settings, and according to Figure 5, the survey results indicate that 38.2% of the respondents had never manually created a Docker container and 47.6% of those who had done so expressed difficulty in the process. Positively, more than 83% of the respondents who created images had been able to publish them onto a container registry. The variation between the need for containerisation and competence to publish an image to a registry implies that 16% of the respondents are using dockerisation mainly for practice. The respondents further reported the dockerisation steps as being characterised by the need for different new concepts and technologies and complex for use with different application stacks.

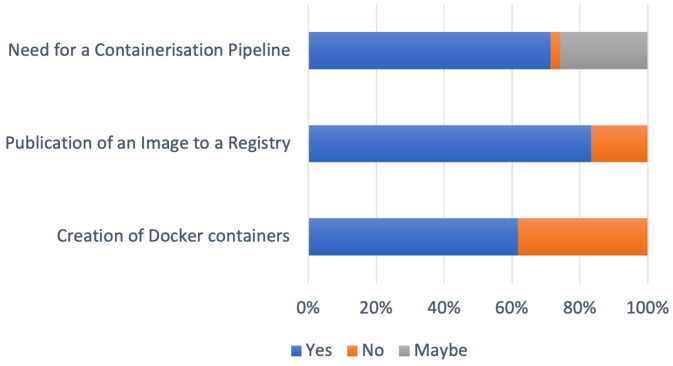

Figure 5: Survey Results

From these results, it is clear that tooling and automation of processes can lead to improved adoption of cloud native technologies. To validate this, the respondents were asked to present views on whether a containerisation pipeline for building Docker images and deploying them to a cloud native orchestration platform would be ideal and the results are shown in Figure 6. More than 70% of the respondents believe that an automated containerisation pipeline can ease application deployment in cloud-oriented setups.

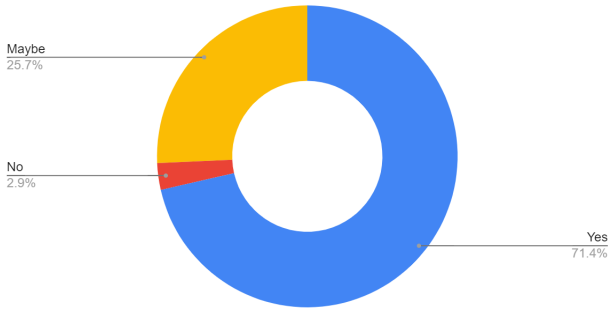

Figure 6: A Containerisation Pipeline

5 A CONTAINERISATION PIPELINE FOR SMALL DEVELOPER TEAMS IN LOW RESOURCE SETTINGS

5.1 Introduction

Automation is increasingly becoming an integral component of continuous software delivery. From source code management in CI to shipping of software to its operational environment in CD, organizations can leverage automation for speed, agility and efficiency in the software development life-cycle. *mira* is an extension platform to the CI/CD pipeline for containerised applications for small developer teams in low resource settings. It loosely automates the build, test, release and deploy operations of CI/CD for different programming frameworks. These operations enable containerisation of an application and its subsequent deployment to a container orchestration platform for production use. *mira* is composed of mainly two components as shown in Figure 7: mira front-end and back-end.

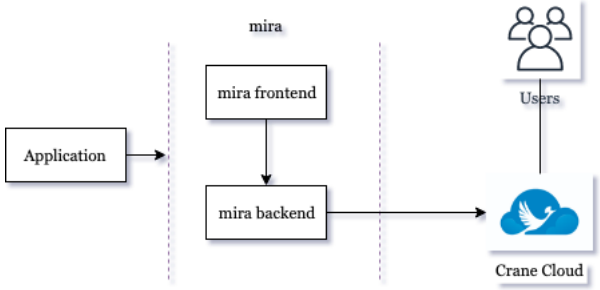

Figure 7: *mira* high-level architecture

[11] https://www.cncf.io/kubecon-cloudnativecon-events/

- *mira* front-end (*http://mira.cranecloud.io:3000/*): A portal that users can provide details about their applications. The details include the location of the source code, Crane Cloud access, application release/version information, image registry and the programming framework used. Currently, mira supports HTML-CSS-JS, ReactJS, NodeJS, PythonFlask, Django and Laravel.
- *mira* back-end (*http://mira.cranecloud.io:5000/*): The back-end is the *mira* API service that uses the front-end information and a preset *Dockerfile* at the root of the application to build the application image (containerisation), push it to an image registry and ultimately deploy to Crane Cloud for access. To better understand the automated process, we present the manual containerisation process in Section 5.2.

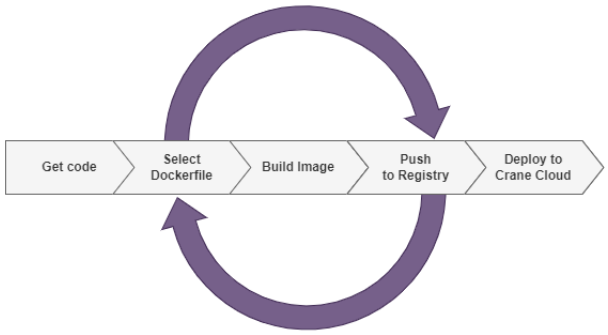

Figure 8: *mira* containerisation pipeline

5.2 Containerisation

Containers are lightweight, self-contained and infrastructure independent, this ensures portability across operating systems, hardware, computing clusters and cloud providers. We will use a basic *ReactJS application* [12] to demonstrate the containerisation process. The prerequisite is to have the Docker application installed on the workspace/host node. A *Dockerfile* (script to build the Docker image) is then created at the root of the application. There is an optional *.dockerignore* file to exclude containerisation of non-required build files and dynamic or sensitive environment variables. The *Dockerfile* has typically five important instructions of the containerisation process:

- **Parent/Base image**: Docker images are composed of dependent layers that form the intermediates and can ideally be rebuilt independently based on the hierarchy. Parent images form the foundation layer upon which other layers are built and are available for use in image registries. Base images have initial empty layers and provide users with flexibility for customization. The choice of the image is dependent on the use case. An application may have a ready-made image available from the official repositories and can be used as is. In some instances, users have to consider ease of use, language compatibility level, overhead creation requirements and security for a customizable solution. We use an official **Node.js** alpine parent image from Alpine Linux project.
  ```
  FROM node:14.15.0-alpine
  ```
- **Working Directory**: This specifies the working directory for subsequent instructions in the *Dockerfile*
  ```
  WORKDIR /app
  ```
- **Instructions**: These are commands executed in the working directory for specific roles. **RUN** is used to install applications or packages required by the container. **COPY** copies files or directories with the more general **ADD** that works with remote and compressed resources.
  ```
  COPY package.json .
  RUN yarn
  COPY . /app
  ```
- **Container Runtime**: These are run-time specific instructions executed when the container starts (**ENTRYPOINT**) or in running mode (**CMD**).
  ```
  CMD ["yarn","start"]
  ```
- **Exposure**: The container application port is then defined with an optional protocol to expose it for access. It is recommended to use a non privileged port that can be mapped to a privileged one during the container execution.
  ```
  EXPOSE 3000
  ```

The *Dockerfile* is a template for the next operational steps to ensure that the application is deployed and accessible to the users. This requires access to a container/image registry and proceeds as follows:

- Build the image
  ```
  docker build -t <username>/repository:tag
  ```
- Run the image to check its operational status
  ```
  docker run <username>/repository:tag
  ```
- Push the image to the registry
  ```
  docker push <username>/repository:tag
  ```
- Use the image: This depends on the hosting platform used and may require further information. In a pure Docker environment, this can be effected with:
  ```
  docker pull <username>/repository:tag
  docker run <username>/repository:tag
  ```
  Crane Cloud deployments require the name of the application, the registry and image URL, the container exposed port and additional environment variables if used.

mira provides a 2-stage abstraction layer for containerisation and deployment processes using preset *Dockerfiles* for the frameworks under consideration and as per requirements of the container orchestration platform. In containerisation (stage 1), the presets represent different programming tools, languages and frameworks that a user chooses. The underlying host platform uses the preset, source code files and docker Command Line Interface (CLI) tools to build and publish the image to a registry. For this study, we use two registries for access and latency reasons: Docker Hub and Harbor. In stage 2, the container orchestration platform's API features are used to pull the image and deploy it under a user defined project or name space.

[12] https://github.com/kabirbaidhya/react-todo-app

5.3 Implementation

The *mira* back-end service is implemented using the NodeJS framework due to its inherent responsive user experience and use in development of real-time, streaming and multiplayer games. It is also well suited for creating scalable APIs. The following NodeJS libraries are used:

- Multer: Middleware for handling multipart/form-data.
- Unzipper: Used for unzipping compressed source code contents.
- Unique-name-generator: Generates random names to uniquely name the file path to the unzipped contents.
- Docker-cli-js: Helps run docker commands on the virtual infrastructure on which *mira* runs, responsible for docker login, build and pushing.

This service has customizable preset *Dockerfiles* for the different frameworks and the *.dockerignore* file. Applications that require bash scripts for execution should include the feature in the source code.

The *mira* front-end is written in the ReactJS framework. Given the back-end service experience in its predominant use of Javascript, ReactJS was the preferred option. ReactJS is a powerful front-end agile development tool based on Javascript. Its inbuilt support for styling tools such as CSS, ease of use and convenience further strengthened our choice for the front-end implementation.

In order to deploy the application, the Crane Cloud API endpoint is required and authentication is done through a user/project token. This represents a subset location of the application in the hosting platform.

5.4 Example Use Case

We will now use the example *ReactJS* application in section 5.2 to demonstrate how *mira* works in a collapsed/simplified 3-step process:

(1) **Application**: We provide details of the application and more specifically React as the framework and the project source contents. At present, it is required that these contents are compressed in ZIP format and are available on the workspace/host node for upload to *mira* through the front-end service.
(2) **Containerisation**: To containerise the application, we specify the name of the image and the tag/release/version information for identification and future updates. In addition, we provide the image registry information. The registry is a store or repository and acts as a distribution point for the Docker image. *mira* supports the official Docker Hub [13] registry and a local service [14] based on the open source Harbor [15] registry. The local instance significantly reduces the times for image pull/push operations especially for large images as it is hosted on the same network as the orchestration platform.
(3) **Deployment**: In Crane Cloud, applications are deployed under projects created by the users. A project is accessed via an authenticated API endpoint using a token. For deployment to Crane Cloud, we provide the project name identifier and the user authorization token.

The 3-step process above on the front-end is shown in Figure 9 and on the *DEPLOY* action, the application is deployed and accessible to users on [16].

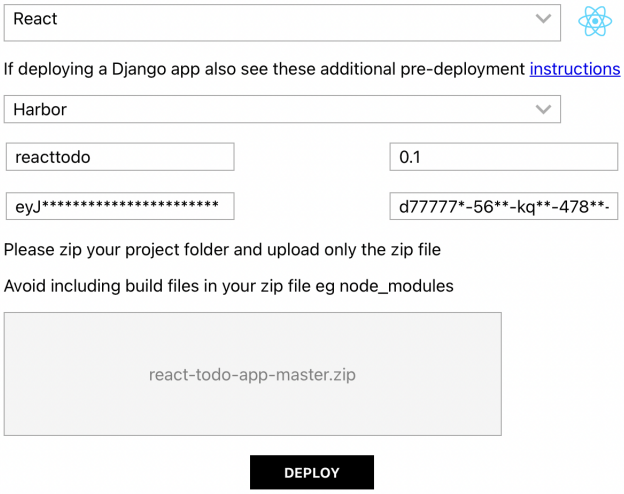

Figure 9: *mira* front-end for the *ReactToDo* application

6 VALIDATION

To validate *mira*, we deploy a total of 42 applications from the major frameworks as per the Figure 3 and shown in Table 1. A project is created in Crane Cloud and the authentication and authorization information required by *mira* is retrieved for use. The rest of the user deployment process is compressed into the front-end described in Section 5.4. It should be noted that number of deployed applications for a framework loosely corresponds to its containerisation complexity. The deployed applications consist of a blend of custom applications that we developed and example real world applications that we obtained from publicly shared github repositories [17]. Additionally, we test with some 'todo' applications [18] multiple times for the different frameworks. While these are somewhat basic examples, multiple successful tests serve as a proof of concept for the particular frameworks. We now provide a discussion of results from each framework:

6.1 HTML-CSS-JS

This contains applications built from a basic combination of CSS, HTML or Javascript and provided a Proof of Concept (PoC) for further frameworks. The only requirement is for the application to be compressed in the ZIP format but the platform is extensible to allow a file listing. Docker Hub, as a default image repository, was used and 5 different applications deployed. This framework does not require external additional libraries in the containerisation

[13] https://hub.docker.com/
[14] https://registry.cranecloud.io/
[15] https://goharbor.io/
[16] https://reacttodo-02-12cac7e6-ed9c-40d8-b8b3.cranecloud.io
[17] https://github.com/gothinkster/react-redux-realworld-example-app
[18] https://github.com/kabirbaidhya/react-todo-app;https://github.com/OmkarPathak/Django-to-do

Table 1: Frameworks and Test Cases

No	Framework	Deployed Applications
1	HTML-CSS-JS	5
2	React	7
3	NodeJS	7
4	Flask	8
5	Django	10
6	Laravel	5

process and hence use a generic *Dockerfile* template with the base *nginx* image. There are no extra reworks for successful deployment of applications in this category. In testing, no failures were recorded in all the 5 application cases.

6.2 React

React is an extensible and scalable open source Javascript 'framework' [4] widely used in developing front-end applications. 7 applications were deployed for validation: 4 used Redux and 3 required persistence using a relational database management system. The applications were developed using React and required a base NodeJS installation. The persistence applications use environment variables to store database connection information. Creation of the preset *Dockerfile* required multiple iterations especially to cater for the different application scenarios and possible external libraries. In the end, the *node:14.9.0-alpine* base image was widely compatible and successfully deployed the 7 applications. There are further considerations that include how environment variables are provided to the application and customized ports. It is assumed that the container orchestration tool provides a means to specify the variable values.

6.3 NodeJS

NodeJS is an asynchronous I/O event-based [14] run-time environment for execution of Javascript outside the browser. It provides a base for more specific Javascript frameworks. 7 applications built using NodeJS were successfully deployed using the pipeline including the *mira* back-end. The setups for NodeJS and React are similar for example in the use of *node:14.9.0-alpine* as the base image and refinements required therein for a universal *Dockerfile* preset. NodeJS requires a customisable entry file provided by the user. We restrict this to the default *index.js* and ensure that it exists for uniform deployment. As with React, the orchestration platform should support specification of dynamic properties including environment attribute values and port numbers where necessary.

6.4 Flask

Python Flask [19] is a simple, lightweight and flexible web application frameworks. It is arguably considered the most popular web application frameworks by the programming community especially given its ease of use and integration while ensuring effective maintenance. 8 applications were deployed using the pipeline 4 of which required relational database support. As expected, non-database support application were fluidly deployed as no extra configurations are required. The database support applications need specification of database credentials with an additional entry to run the database migrations. The *Python 3.5* base docker image worked successfully for all the application test cases and the resultant application exposed to run on port *5000*.

6.5 Django

Django [20] is a python web development framework with a mature, practical and clean design. We deployed 10 applications to run this set of tests: 2 non-database support and 8 with relational database support (Sqlite (2) and MySQL (6)). The applications supported are Django 3+ and used the *Python 3.5* base docker image. The non-database support for Django applications required a single iteration in testing. The database support applications took 3-4 deployment iterations with extra scripts required to run database migrations and creation of a superuser account. Additionally, there was need to customise the default application name used in the start command of the Django application and hence left to be defined by the user. On successful deployment, the user needs to add an environment variable of *Allowed Hosts* for the generated application for external access. Django 2 and lower versions required older deprecated versions of Python and hence our support for Django 3+.

6.6 Laravel

Laravel is an open source PHP web application framework that follows a model-view-controller design pattern. All the applications tested required relational database support, Laravel version 6+ and PHP 7.3+. As with Django, the Laravel version is closely tied to specific versions of PHP. The database credentials can be populated into the application before deployment but can also be later updated using environment variables. The migrations are enabled in the *Dockerfile* and are effected at containerisation stage and hence require that a database is already created for the application.

7 LIMITATIONS

In the survey, we had a total of 35 respondents which is not a full representation of the developer community in Uganda. This is attributed to the time and participation constraints of the respondents despite a 3-phased running of the survey. This also affected complete evaluation of the platform but this is planned in the next major release of *mira*.

We designed *mira* to initially support 6 development frameworks/languages/tools based on the immediate needs and results of the survey. The list can be infinitely long and scalability can easily become an issue. Updates of these tools may also require continuous review and development of the platform. *mira* is deployed using a client-server architecture and hence introduces a potential failure point. The same architecture could still be used with distributed nodes but this requires more compute, storage and network resources. In addition, *mira* uses the application as specified by the user and does not further perform security checks at creation or operation. It is assumed that the image registry is responsible for the security and integrity of the resultant image. Confidential information such as Docker secrets is not currently supported.

[19] https://github.com/pallets/flask/

[20] https://github.com/django/django

8 CONCLUSION & FUTURE WORKS

The use of DevOps is revolutionising the collaborative approach to faster development, integration and delivery of software. However, our survey results indicate that DevOps concepts and its related principles and practices are relatively new in developer environments of low resource settings. Application containerisation is an important practice that can simplify the use of DevOps practices. The lack of tool support in containerisation can further hinder adoption and use. The assumption is that developers are in position to manually containerise their applications, push them to a repository and pull to the hosting environment. Our results show that this assumption is false, and over 73% of the respondents further agreed that automation of the containerisation process can simplify and accelerate software delivery.

Based on the results and recommendations from the survey, we designed and developed *mira* - a containerisation pipeline for a cloud native application deployment platform, Crane Cloud. *mira* provides a front-end interface through which users can provide the development framework/tool and the source code and the back-end containerises and deploys the application to a container orchestration platform for use. To validate *mira*, we successfully deployed 42 applications from 6 different widely used frameworks/tools in low resource settings. The deployments are after a series of iterations to understand the application and development framework/tool requirements.

In the next phase of the project, we will port *mira* as a CLI application for scalability and to provide more usage options to the users especially for experienced developers. For completeness as a CI/CD pipeline, the platform will support application updates in a staging-production context in collaboration with integration and source code management tools. New development frameworks/tools shall be incorporated as and when the need arises.

ACKNOWLEDGMENTS

The authors would like to acknowledge all persons who have contributed to the development of the containerisation pipeline, Crane Cloud and resources in funding or in-kind to bring the project to life. We thank Paul Maritz for the support and advice in the conceptualisation of the Crane Cloud project. We thank the Crane Cloud team for feedback and input. The authors would also like to acknowledge support from the Government of Uganda through the Makerere University Research Innovation Fund (RIF).

REFERENCES

[1] Len Bass, Ingo Weber, and Liming Zhu. 2015. *DevOps: A software architect's perspective.* Addison-Wesley Professional.
[2] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. 2016. DevOps. *Ieee Software* 33, 3 (2016), 94–100.
[3] Martin Fowler and Matthew Foemmel. 2006. Continuous integration.
[4] Cory Gackenheimer and Akshat Paul. 2015. *Introduction to React.* Vol. 52. Springer.
[5] Jez Humble and Joanne Molesky. 2011. Why enterprises must adopt devops to enable continuous delivery. *Cutter IT Journal* 24, 8 (2011), 6.
[6] Morgan B Kamuto and Josef J Langerman. 2017. Factors inhibiting the adoption of DevOps in large organisations: South African context. In *2017 2nd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT).* IEEE, 48–51.
[7] Jamal Mahboob and Joel Coffman. 2021. A Kubernetes CI/CD Pipeline with Asylo as a Trusted Execution Environment Abstraction Framework. In *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC).* IEEE, 0529–0535.
[8] Krikor Maroukian and Stephen R Gulliver. 2020. Leading DevOps Practice and Principle Adoption. *arXiv preprint arXiv:2008.10515* (2020).
[9] Sriniketan Mysari and Vaibhav Bejgam. 2020. Continuous Integration and Continuous Deployment Pipeline Automation Using Jenkins Ansible. In *2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE).* IEEE, 1–4.
[10] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. 2016. *Microservice architecture: aligning principles, practices, and culture.* " O'Reilly Media, Inc.".
[11] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. 2017. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access* 5 (2017), 3909–3943.
[12] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. 2021. Automating serverless deployments for DevOps organizations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 57–69.
[13] Sérgio Teixeira, Rafael Arrais, and Germano Veiga. 2021. Cloud Simulation for Continuous Integration and Deployment in Robotics. In *2021 IEEE 19th International Conference on Industrial Informatics (INDIN).* IEEE, 1–8.
[14] Stefan Tilkov and Steve Vinoski. 2010. Node. js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing* 14, 6 (2010), 80–83.
[15] Giovanni Toffetti, Sandro Brunner, Martin Blöchlinger, Josef Spillner, and Thomas Michael Bohnert. 2017. Self-managing cloud-native applications: Design, implementation, and experience. *Future Generation Computer Systems* 72 (2017), 165–179.
[16] Kostis Trantzas, Christos Tranoris, Spyros Denazis, Rafael Direito, Diogo Gomes, Jorge Gallego-Madrid, Ana Hermosilla, and Antonio Skarmeta. 2021. An automated CI/CD process for testing and deployment of Network Applications over 5G infrastructure. In *2021 IEEE International Mediterranean Conference on Communications and Networking (MeditCom).* IEEE, 156–161.
[17] Mandi Walls. 2013. *Building a DevOps culture.* " O'Reilly Media, Inc.".
[18] Liming Zhu, Len Bass, and George Champlin-Scharff. 2016. DevOps and its practices. *IEEE Software* 33, 3 (2016), 32–34.