Software on Internet of Things devices: monolithic vs containerized

Jagvir Singh Bal,
j.s.bal@student.utwente.nl,
University of Twente,
The Netherlands

***Abstract*—This study compares container-based versus monolithic application techniques while examining the performance metrics when updating IoT devices. IoT device security is in danger from flawed security implementations and ignored updates. For each architecture, the effects of resource limitations are looked at. The study aims to identify and compare these challenges and their effects on updating IoT devices. Applications built with containers are anticipated to perform better and be easier to maintain. Tests are run to investigate these aspects. The tests show that Django is able to manage resources better compared to Flask, but the overall deployment of a container application is easier to manage and scale up. This study sheds light on IoT web application initialization and updating difficulties. There seem to be advantages and disadvantages depending on which implementation is used. The paper emphasizes the advantages of container-based applications.**

***Index Terms*—Internet of Things (IoT), monolithic application, container-based application**

1. Introduction

The speed at which new devices are connected to the internet is increasing exponentially. The use of IoT devices is increasing all across the world [1].

The world is developing at a high rate at this also has a similar effect on IoT devices. Technological advancements lead to devices that have more software components. Devices need to provide even more functionality and also need to create support for new and upcoming services that make use of data available on the internet [2].

However, the implementation of IoT devices as it is right now does bring risks, such as exposing data of individuals or falsifying data [3]. Therefore, it is important that updating and patching of these software components is done on a regular basis to ensure that a certain level of security is maintained and new vulnerabilities or attacks are neutralized.

In recent years, there has been notable attention towards research on the process of updating software/firmware of IoT devices. There are many challenges, mainly because of limitations in IoT devices and networks [2]. Examples of these limitations include power consumption or memory availability, and on a network there could be bandwidth constraints. The ever growing complexity of current IoT systems and deployments necessitates the management of software versions and dependencies between the various software components within a device or system, if this is not done complex systems may easily stop functioning. It may be crucial to ensure that updating such systems do not cause any disruption in the service that they are providing, e.g. for IoT devices used in the medical field. Additionally the process of updating should not compromise the security of the system or any of its components.

So, there is a need for a better approach to update IoT devices while taking networks constraints into account in particular for remote IoT devices. The approach should be lightweight and flexible. Because it is important that a suggested approach can be used in different kinds of environments. IoT systems can have different underlying communication technologies and protocols [2]. This is also one of the limitations of IoT systems. This study looks at the two different ways an IoT device can run an application, while monitoring the resources used on the IoT device. First of all, the application can be run in monolithic structure, meaning that the architecture is designed as a single, self contained unit. The monolithic application and all its components, modules and functionalities are tightly integrated together. The application is deployed as a whole, there is no separation between the management of the individual functionalities. Secondly, we look at container based applications which is an application that is split into containers which are lightweight and isolated environments that encapsulate the applications and its dependencies. The container provides a standardized and portable way to run applications, and the underlying architecture can be anything as all the necessary information is available inside the container.

2. Problem statement

The problem at hand are concerns in the field of IoT devices with regards to software updates and vulnerability patches. Specifically, there are cases that implementations of update methods are flawed, such as the lack of proper software patch signing, or in some cases even worse the disregards of updates altogether. There is literature that describes challenges with software updates on IoT devices [2], but there is no comparison between monolithic and container applications when initializing or updating an application.

2.1 Research question

From the problem statement the following research question is established.
What are the effects on the IoT device's performance when initializing and updating a container based application

compared to a monolithic application?

Sub-questions are as follows.
What is the initialization time?
What is the update time?
How is the CPU usage affected during initialization and updating?
How is the memory affected when initialization and updating?
How is the storage effected when initialization and updating?

The research questions suggests a general comparison between the two different application structures. To access the differences of the application we will make use of the sub questions. The sub question focus of the different permorfence metrics that will be collected.

3. Related work

As mentioned above there is an increase in complexity of IoT systems. The development of monolithic systems has reached its limitations, because in today's large, complex, and fast advancing technology such systems are slow and inefficient. The monolithic system works as a whole so it will need to be updated or adjusted as one unit. It would not be possible to simply update partial parts of the system on the go to maintain efficiency. So, connecting a lot of devices to the internet has many benefits and possibilities, but there is still room for improvement. This is the main reason for the development of the container based applications, this is comparable to microservices architecture. Instead of maintaining a large amount of code, it is modularized. Modular code is easier to update, fix and work with even for new employees, they can more easily understand their task as it is a specific module of the code [4].

The researchers in [5] talk about the use of microservices approach to split up a monolithic application into a set of distributed services. Also the best practices that are used in microservices approach are investigated and considered to be applied in IoT systems. This can lead to a high maintainability and scalability. Similarly research [6] shows that implementing a micoscerces-based IoT device can lead to benefits such as better mitigation time, performance and impact on the IoT device's functionality.

4. Research setup and data collection

The IoT device used in this project is a Raspberry Pi 4 Model B Rev 1.2. The operating system is the Linux based, Raspberry Pi OS, specifically "Debian GNU/Linux 11 (bullseye)" [7]. All development and testing is done on the raspberry through visual studio code using a SSH connection from a windows machine. In this research we will compare two web application deployments, both of them are based on the python programming language. Raspberry Pi OS has python 3 [8] installed by default. Besides python the packages manager for python (pip) is also required. Pip is used to install python packages also known as modules [9]. The modules are pulled from The Python Package Index(PyPI) [10]. Python and pip are the minimum software requirements to get started. The two different application deployments that will be tested are using the Flask [11] and Django [12] modules respectively.

Flask is a micro web framework that does not have any outside dependencies on external libraries. It is known to be flexible and lightweight. The developer can create a more personalized development environment, as per requirement external libraries can be added.

On the other hand we will be looking at Django, a full stack framework. This framework has a standard method for web deployment. It also offers several extensions within the framework to add more functionality such as forms, database administration and authentication.

4.1 Docker setup

Docker version 24.0.5, build ced0996 will be used on the Raspberry Pi to run the Flask application and a database in a containerized manner. The installation guide can be found in the docker documentation, It provides instructions on how to setup docker on a system that is running Debian [13]. The Docker software allows for virtualization on operating-system level, using docker any kind of software can be packaged into a container. The container is hosted by docker engine and can be run on a system regardless of the OS or device that it is hosted on.

4.2 Testing setup

The tests will be done using python. The python testing programs can be found in the git repository [14]. The setup sections of Flask and Django describe the structure of the testing programs. The overall structure of each programs is similar. Series of commands are being executed for initialization and afterwards for updating while measurements are taken. It is Important to note that the measurements must be taken while the commands are executed and running.

Python has a module called subprocess [15] that will be used to take measurements while other commands are run. Subprocess enables the creation of new processes and connect to their inputs, outputs or errors, and obtain return codes. For this project subprocesses: run, Popen and check_output will be used. When initiating a command using run, the command is executed and the program will wait until the command is finished to move on in the program. Popen is used to run the measuring commands. These commands run like child programs within the program so that the measurements are taken simultaneously with the initialization and update commands sequences. The check_output is used to read the output of a command, which is used to determine if the database is operational.

4.3 Flask setup

Firstly we look at a containerized approach to deploying a web application. For this purpose, we use the Flask framework to setup a web application that has a database. Flask version 2.3.2 is used. We start with a local Flask web application. This needs to be turned into an image to be able to run in a container. First a Dockerfile needs to be defined within the local project, this file contains the instruction on how to build the image. It starts by installing and setting up python 3.9.2 [8] in the image. This python version comes with pip 21.0.1. Next we must define any python modules that are used in the project. In our case, the Flask module is needed to build a Flask application. Flask is added to the requirement by put it in the requirements.txt file in the local project directory. Additionally a module called mysql-connector-python 8.1.0 is required to connect the web app to the database. Using pip this requirements.txt file is installed in the image. Then the application data is copied from the local project in to the image, and lastly the Dockerfile defines how to execute the Flask application within the image. After the image has been created docker can be used to run the image in a container [16].

The database will to be setup as a separate container, given that we would like to test a containerized application. We will be using a MySQL database for this project. The standalone image created from the local Flask application can be run using the docker run command, similarly is it possible to setup a image for the database part of the application. But an easier way to setup the database is to make use of docker compose [17]. Like the aforementioned Dockerfile, docker compose makes use of configuration data from a human-readable data-serialization language (YAML) file to create the Flask and database images respectively using only one configuration file [18].

The web application and MySQL database are defined as services within the configuration. Any number of services can be added to the configuration this is how an application can be split up in to different micro-services. Upon executing docker compose the Flask web app image is created and the database image for MySQL is pulled from docker hub [19]. The version, also known as a tag, is specified in the YAML file. This is essential because the YAML file needs to be modified to apply updates to the app as part of the testing. The local Flask application folder also contains a database initialization file which is associated with the database service defined in the YAML file by using the same name. The database service is ”db”, the local application contains a sub folder with the same name.

4.4 *Django setup*

Secondly we look at a monolithic web application deployment using the Django framework. We start by setting up virtual environment(venv) for python. The venv [20] is created on top of the existing python installation and is isolated from the packages that are installed on the system. This step is comparable with the the python installation in the docker container from the Flask initialization. After the venv has been setup all the testing will happen in this environment.

First of all the Django 4.2.4 package will be installed in the venv. After the installation a blank Django project is generated. This procedure is simple because Django, as monolithic framework, has all the necessary information in the Django generated project to launch a basic web application. By default Django makes use of sqlite3 database back end. We would like to use a MySQL database in this project. But during the setup of the Django app it was found that MySQL has stopped support for the Debian operating system [21]. So instead of MySQL a PostgreSQL 13 database will be used for testing. The setup procedure is simple using the Django documentation. After setting up the web app, the database details can be setup in the settings.py file within the project directory [22]. For Django to work with a PostgreSQL database, we first need install PostgreSQL on the Raspberry Pi. The python module called psycopg2-binary 2.9.6 is required for PostgreSQL to function and needs to be installed in the venv.

As mentioned before configurations for the Django application can be done in the settings.py. During the initialization the following PostgreSQL database information will be added: database engine, name of the database, username, password, network address and port [23]. Besides this it needs to be specify that the web application should be viable on the localhost, by default the application cannot be accessed.

4.5 *Data collection setup*

During the test execution the initialization time, CPU utilization, memory usage, storage usage is collected. Time-related measurements are done in the Raspberry Pi using the python module time. For the performance measurements vmstat from procps-ng 3.3.17 [24] is used. Vmstat is a performance monitoring command for Linux. The parameter used for vmstat is 1 this is the minimum, meaning the values will be re-measured and reported every second. The data output by vmstat will be converted to graphs by using the online tool vmstatly, developed by jsargiot [25]. The power composition cannot be measured since the Raspberry Pi does not support the way powerstat estimates power usage [26]. All the collected data is stored in .txt files. The initialization time for the Flask app consists of the time it takes to build and docker compose up the application. That means the total time is take to run the Flask service and the database service according to the docker configuration file. The initialization time for the Django app consists of the time spend generating the app with the correct database setup. In the following test we look at the same measurements when applying a update to the database for both deployments.

In addition to automated tests the localhost is checked after completion of every test, to ensure the web application behaved as expected. The expected behaviour is that the two pages of the web application load. The homepage displaying text, and a second page accessing and printing the initial database table.

5. Results

In the following section, there is an overview of all the data that has been collected. Most of the results have been graphed and tables have been added where two different setups are compared. The in-depth analysis can be found in the next section.

5.1 Flask

5.1.1 Initialization testing

In Table 1 you will find the time measurements of test 1, building a Flask application with no cached data. Table 2 provides the measurements for test 2, essentially rebuilding the app with cached data. Table 3 show building with only python in cache and Table 4 with only MySQL in cache, which function as tests 3 and 4 respectively.

TABLE 1
BUILD WITH NO CACHED DATA

Initialization task	Time in seconds
Total:	379.36
Application:	315.71
Database:	63.61

TABLE 2
BUILD WITH CACHED DATA

Initialization task	Time in seconds
Total:	70.79
Application:	6.68
Database:	64.05

TABLE 3
BUILD WITH ONLY PYTHON CACHED

Initialization task	Time in seconds
Total:	160.98
Application:	95.75
Database:	65.19

TABLE 4
BUILD WITH ONLY MYSQL CACHED

Initialization task	Time in seconds
Total:	303.16
Application:	236.13
Database:	66.00

Figure 1 to 4 show CPU stats over time during the four initialization tests that have been run.

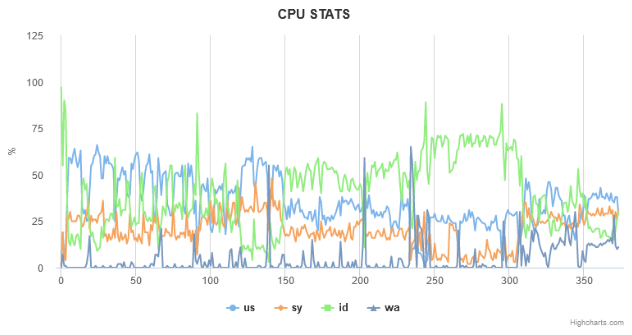

Fig. 1. CPU stats during Flask initialization

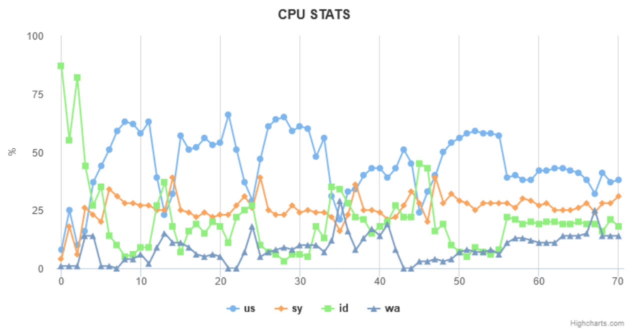

Fig. 2. CPU stats during Flask initialization test with cache

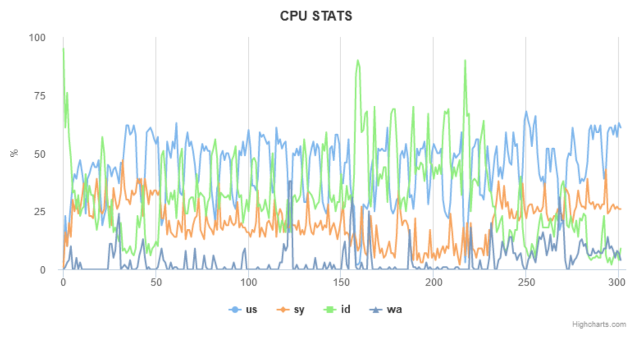

Fig. 3. CPU stats during Flask initialization with python cache

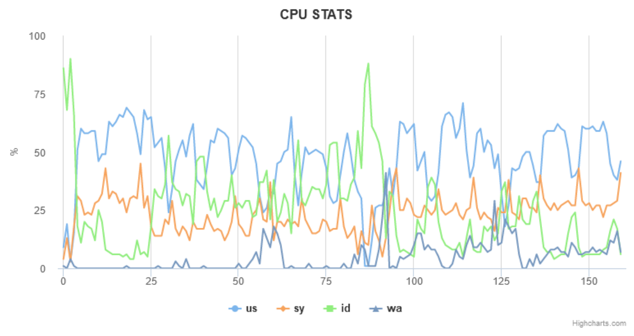

Fig. 4. CPU stats during Flask initialization with MySQL cache

`us`: percentage of cpu used for running non-kernel code.
`sy`: percentage of cpu used for running kernel code.
`id`: cpu idle time in percentage.
`wa`: percentage of time spent by cpu for waiting to IO.

Figures 5 to 8 show memory stats during the initialization testing.

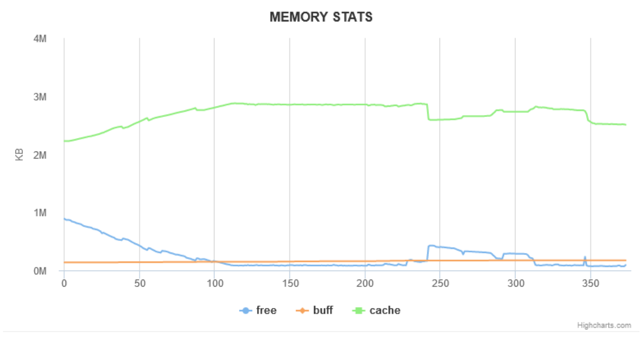

Fig. 5. Memory stats during Flask initialization

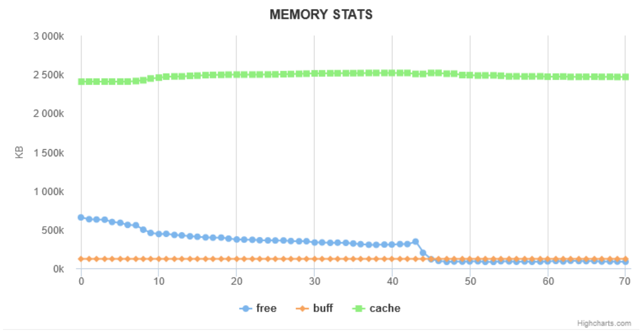

Fig. 6. Memory stats during Flask initialization with cache

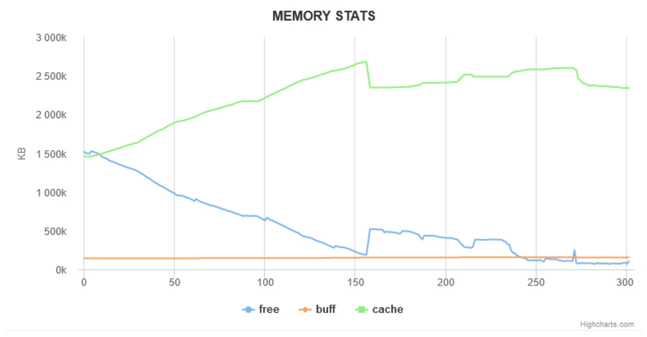

Fig. 7. Memory stats during Flask initialization with python cache

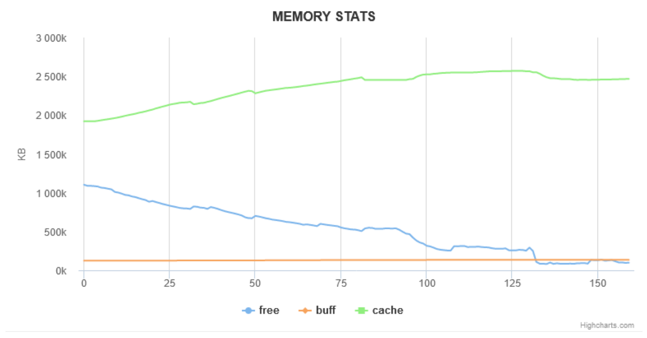

Fig. 8. Memory stats during Flask initialization with MySQL cache

`free`: Idle Memory
`buff`: Memory used as buffers, like before/after I/O operations
`cache`: Memory used as cache by the Operating System

Figures 9 to 12 show disk stats during the initialization testing.

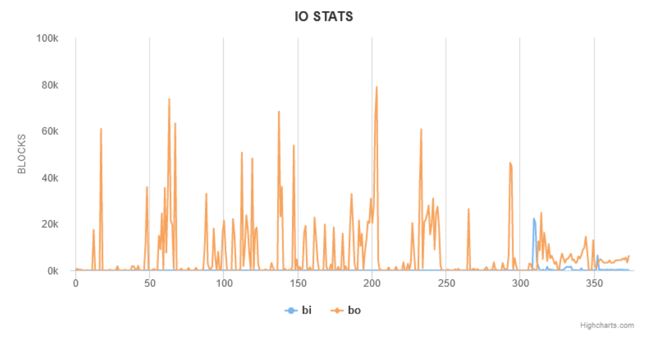

Fig. 9. Disk stats during Flask initialization

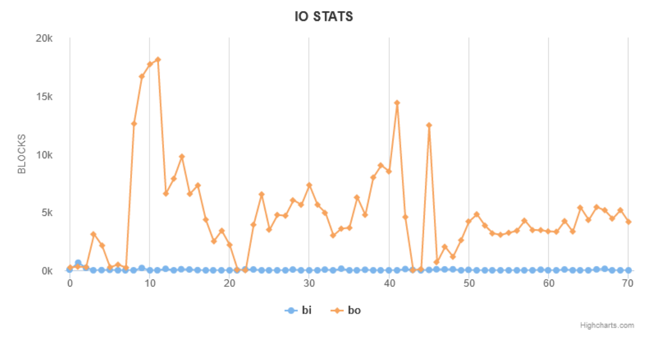

Fig. 10. Disk stats during Flask initialization with cache

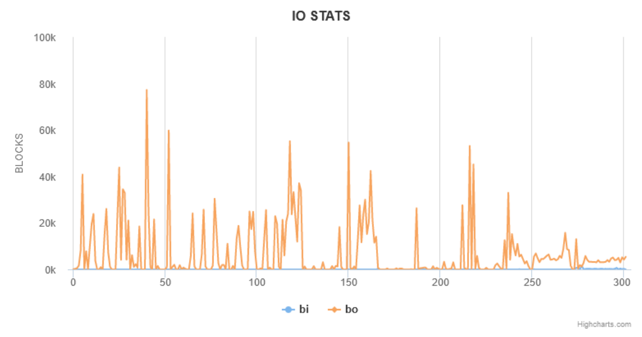

Fig. 11. Disk stats during Flask initialization with python cache

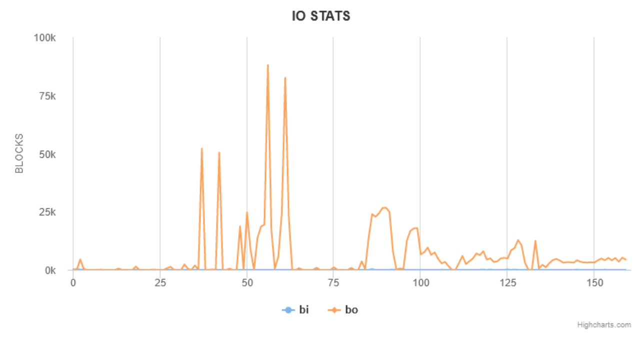

Fig. 12. Disk stats during Flask initialization with MySQL cache

`bi`: Blocks received from block device - Read (like a hard disk)
`bo`: Blocks sent to a block device - Write

Table 5 shows the total storage used for each initialization test.

TABLE 5
STORAGE USED FOR INITIALIZATION TESTING

Test	Used storage in GB
1	1.65
2	0.19
3	1.65
4	0.67

5.1.2 *Update testing*

For updating the MySQL database running with Flask the following two tests were conducted. The first pulls the new database version from docker hub, see the results in Table 6. The second test does the update with a cached newer version of the database, see the results in Table 7.

TABLE 6
BUILD WITH PULLING MYSQL:8.0.34

Update time:	119.39 seconds
Space used	0.59 GB

TABLE 7
BUILD WITH MYSQL:8.0.34 CACHED

Update time:	10.27 seconds
Space used	0.00 GB

Figures 13 and 14 display CPU stats during the update of the database.

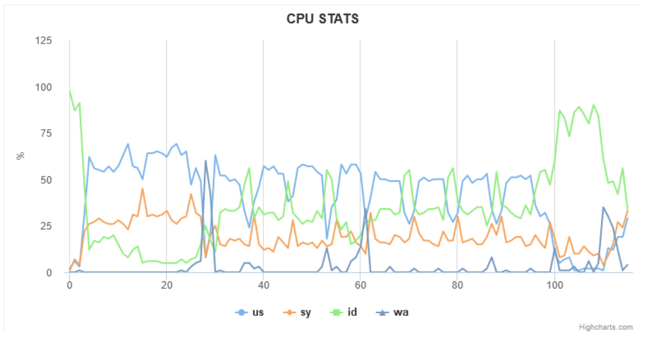

Fig. 13. CPU stats during updating

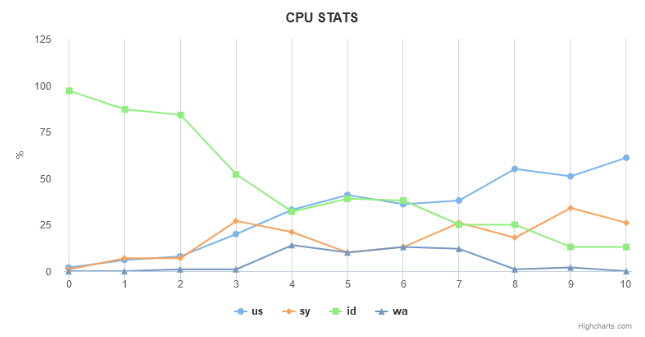

Fig. 14. CPU stats during updating with cache

Figures 15 and 16 display memory stats during the update of the database.

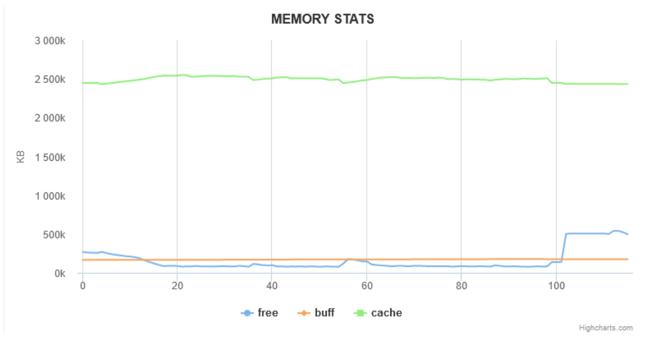

Fig. 15. Memory stats during updating

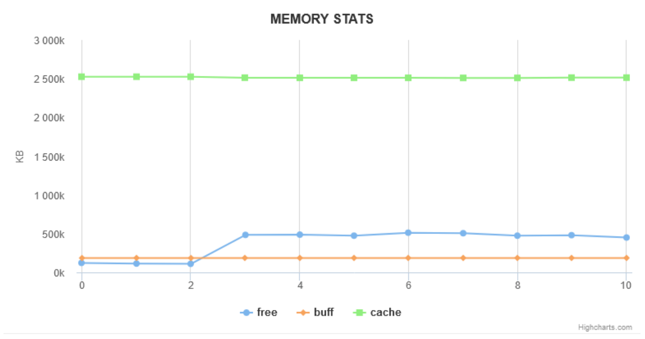

Fig. 16. Memory stats during updating with cache

Figures 17 and 18 display disk stats during the update of the database.

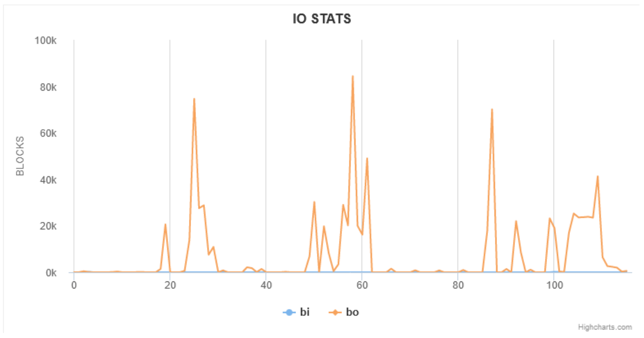

Fig. 17. Disk stats during updating

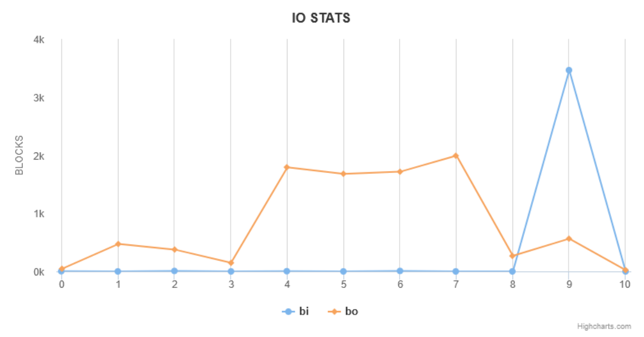

Fig. 18. Disk stats during updating with cache

5.2 *Django*

5.2.1 *Initialization testing*

Table 8 provide time measurements for the Django application initialization for the first test with no cache. Table 9, displays the time measurements with cached data.

TABLE 8
BUILD DJANGO

Initialization time:	134.54 seconds
Space used	0.21 GB

TABLE 9
BUILD DJANGO WITH CACHE

Update time:	38.41 seconds
Space used	0.00 GB

Figure 19 and 20 show the CPU stats while the Django application is initialized.

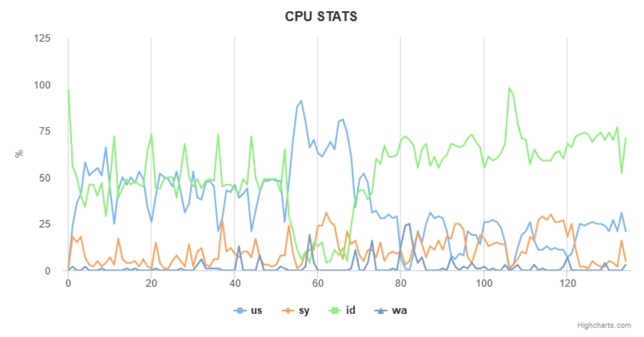

Fig. 19. CPU stats during Django initialization test 1

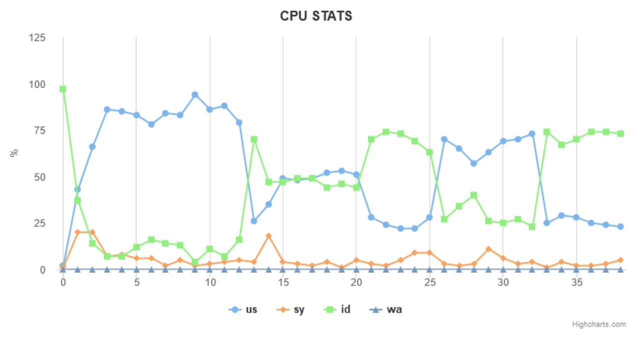

Fig. 20. CPU stats during Django initialization test 2

Figure 21 and 22 show the memory stats while the Django application is initialized.

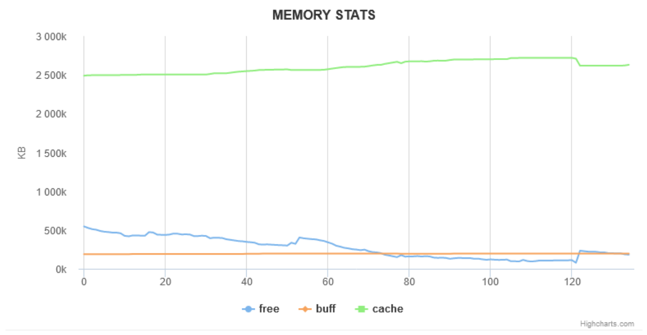

Fig. 21. Memory stats during Django initialization test 1

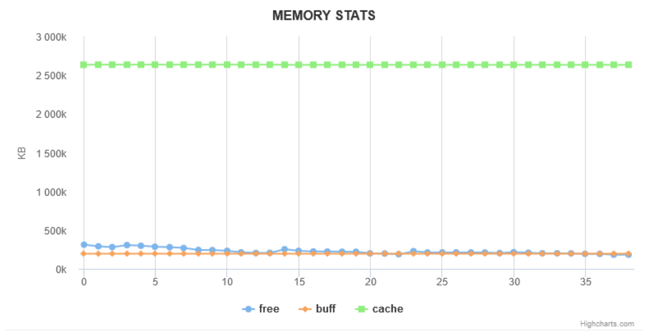

Fig. 22. Memory stats during Django initialization test 2

Figure 23 and 24 show the disk stats while the Django application is initialized.

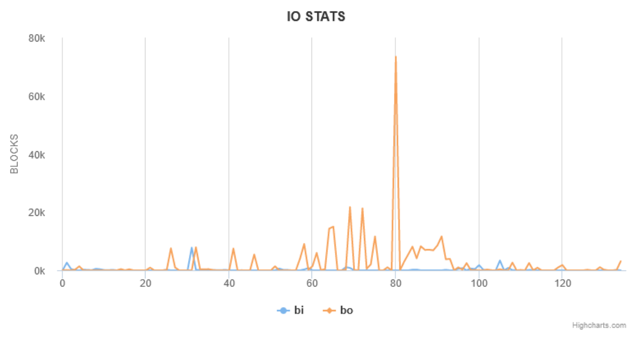

Fig. 23. Disk stats during Django initialization test 1

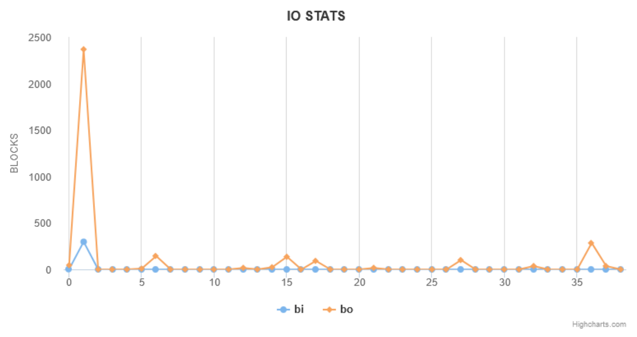

Fig. 24. Disk stats during Django initialization test 2

6. Discussion

6.1 *Flask and Django initialization*

By comparing Table 1 through 2 it is found that there is a difference in build time when cached data is used. The test in Table 1 needs to pull all the data, this is then cached and the test in Table 2 uses this data to run the initialization process. The database initialization time is effectively equivalent, considering all tests. The graphs indicate that a significant amount of time is spend on pulling data from the internet, that is either from docker hub or PyPI in this case. Comparing Table 3 and 4 it can be seen that having python over MySQL cached results in a faster build time. This means that the network plays a big role in the initialization and should be further explored in future work.

According to tables Table 1, 2, 8 and 9 the build time for the Flask application is longer when compared to the Django times. Flask is slower by 244.82 seconds with no cached data internet and 32.38 seconds slower with cache. No cache means that the python installer and MySQL image need to be pulled from the internet. Thus we find that the initialization time is faster for setting up the monolithic Django framework. The steps required to setup Django are also easier compared with a the Flask application. This is mainly because Django contains all the necessary information to run the application, where as Flask is run using docker and thus has to create the container environment. If Flask is run by itself without the use of docker the setup is as easy as the Django application, but that is not relevant for this project's scope.

6.2 *Testing limitations*

After reestablishing a SSH connection with the Raspberry Pi, or restarting the device, and running the initialization test, the Flask web application is not accessible or visible on the local network. During testing this is fixed by running "docker compose up" by itself in the terminal before running the test file. The test file executes the docker compose up in detached mode using parameter -d. This is the reason that the port of the Flask web application is not opened. This would not be relevant in a normal use case of an IoT device, but rather a result of the Raspberry Pi having to self monitor during the execution of the docker compose. Future work could look into monitoring the IoT device by using a second device.

When setting up Django it was found that MySQL has been deprecated for the Debian OS, but by using docker you are still able to run it on the Raspberry Pi. Docker is able to port the current version of MySQL to the ARM architecture of the Raspberry Pi. This is one benefit of using docker to containerize an application. This change in the setup will not affect the results, because differences between these two types of databases can mainly be seen when querying. In our case, we look at the initialization.

The python testing script is unable pass commands to PostgreSQL. Before testing PostgreSQL has been installed manually, to set define three parameter. The user, password and a database name is defined [27]. These operation are simple command that are send to PostgreSQL, they have no significant impact on the test. They are necessary to run the test. PostgreSQL is uninstalled after the parameters are set.

6.3 CPU metrics

Looking at the CPU usage in Figure 1 through 4, the idle time drops significantly at the start of each test. According to the tests this is when the docker compose is started. The CPU usage related to non-kernel code increases, this is all code not related to maintaining the continues operation of the operating system. This is relevant, because the test executes a series of commands which are not related to kernel operation. The sy, kernel related CPU usage seems to be steady throughout all the different tests. The us and id lines from the Figure 1 through 4 show a clear pattern of opposites in the CPU stats data.

The wa, time the CPU spends on waiting on IO. Throughout test 1 with cached data and test 2 without cached data, in Figure 1 and 2 respectively, there seem to be significant spiking in wa after the tests are at the halfway point. It is notable that after the halfway point both tests have a steady increase in wa. This is explained by the image being extracted after it has been pulled. The steady increase towards the end could be explained by the database initialization. Tests 3 with python cache and 4 with MySQL cache, in Figure 3 and 4 seem to show a similar pattern, but with a lower usage overall. Figure 19 of Django initialization has a high peak at the halfway point this is when the PostgreSQL installation started. Similarly, to Flask is idles towards the end of the test when the application is started. It is notable that overall the Django application puts less load on the CPU.

6.4 Memory metrics

The most interesting part of the memory usages is looking at the cache line in Figure 5 through 8. In the first test when there is no cached data available for the test it is notable that during the test more data is cached. The could be explained by the images being pulled and cached for use later in the test. During the second test it can clearly be seen that no new data is being cached as the graph is steady throughout the whole duration of the test. This can be explained by the fact that the application data is directly available in the storage.

In the last two tests some data is being cached as only a part is available is the storage, so the overall usage is less then the first test.The buffer is not relevant as there are no changes during testing. For the idle memory it can be said that the graph mostly runs as an opposite to the memory used as cache. As the memory in use is subtracted from the overall available memory. Comparing Figure 5 with 21 it is notable that Django used less memory overall during the while duration of the test.

6.5 Disk metrics

In the disk usage we first look at bo, write operations during the tests. Apart from the end of Flask test 1 with cache there seems to be near zero reads happening during the tests. The storage writes have a similar pattern to the memory usage. Test 1 has the most usage and, with all data cached test 2 the least. Tests 3 and 4 have different pattern, so writing happens differently depending on what needs to be written to storage either the python or MySQL image. The storage used to run each test can be found in Table 5.

The is no clear pattern in the disk usage that can be explained by using the CPU or memory stats from Figure 19 through Figure 24. Figure 23, show a significant spike at the 80 seconds mark, this is likely due to the database being installed. In Figure 24, there is a spike for disk writing at the start this because cached data is used to initialize the application. The spike is explained be the pyhton venv setup at the start of the test.

In the cached testing memory is even more stable. The CPU and disk activity has moved towards the beginning of the test.

6.6 Updating Flask

The results of the update tests can be found in table 6 and 7. When applying a update to the Flask application most of the test duration is spend on downloading the updated database image. The update itself is completed within 10.27 seconds, see table 7. As mentioned before network plays an important role when initializing and also when updating.

Figure 13 and 14 show the CPU usage when updating. The non-kernel code increases and the CPU idle time percentage goes down. During the update test a newer image in downloaded and the container is rebuild, these processes are all part of non-kernel code.

There is more free memory when cached data is used, this is depicted in figure 16. In figure 15, there are multiple writes to storage this is not the case in figure 16, because the cached data is used.

6.7 Updating Django

There is no data available for update testing using the Django environment, because running the measurements from the same device as the tests it self proved to be difficult. There is an opportunity to look at this aspect in future work.

7. Future work

It was notable that network play an important role when in comes to building an application both on the Django framework and also with Flask. In future testing the network of the IoT device should be monitored. This is relevant in a real world case, because while initialization could be done using a local device the same cannot be said about applying updates. In most cases updates are applied remotely so the stability and bandwidth of the network are important factors to monitor.

During testing with Flask and Django, it was found that measuring on the same device where the test are being run creates limitations. For future work the use of a secondary device or attachments to the IoT device should be considered for measurements. For power consumption a breakout board such as the INA219 could be used [28].

8. Conclusion

Based on the results we can conclude that the Django monolithic framework uses less resources overall. Making use of a containerized Flask deployment does have benefits even though it would cost more resources to initialize. The flexibility that the container-based system offers allows for greater scalability. Based on the exact implementation and scale of any IoT device a trade needs to be made between the resources required to maintain, update, and patch the device over its service lifetime and the resources available on the device, and the initial cost associated with it.

9. Acknowledgment

I would like to express my appreciation to my supervisor S. Simonetto and my coordinator dr. ir. A. Chiumento for providing me with the opportunity to conduct this research and for their guidance throughout.

References

[1] R. Kandaswamy and D. Furlonger. Blockchain-based transformation. [Online]. Available: https://www.gartner.com/en/doc/3869696-blockchain-based-transformation-a-gartner-trend-insight-report/

[2] J. L. Hernández-Ramos, G. Baldini, S. N. Matheu, and A. Skarmeta, "Updating iot devices: challenges and potential approaches," in *2020 Global Internet of Things Summit (GIoTS)*, 2020, pp. 1–5.

[3] V. Hassija, V. Chamola, V. Saxena, D. Jain, P. Goyal, and B. Sikdar, "A survey on iot security: Application areas, security threats, and solution architectures," *IEEE Access*, vol. 7, pp. 82 721–82 743, 2019.

[4] P. Krivic, P. Skocir, M. Kusek, and G. Jezic, "Microservices as agents in iot systems," in *Agent and Multi-Agent Systems: Technology and Applications*, G. Jezic, M. Kusek, Y.-H. J. Chen-Burger, R. J. Howlett, and L. C. Jain, Eds. Cham: Springer International Publishing, 2017, pp. 22–31.

[5] B. Butzin, F. Golatowski, and D. Timmermann, "Microservices approach for the internet of things," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2016, pp. 1–6.

[6] U. Maroof, A. Shaghaghi, R. Michelin, and S. Jha, "irecover: Patch your iot on-the-fly," *Future Generation Computer Systems*, vol. 132, pp. 178–193, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X22000589

[7] Debian gnu/linux 11 (bullseye). [Online]. Available: https://www.debian.org/releases/bullseye/

[8] Python 3.9.2. [Online]. Available: https://docs.python.org/release/3.9.2/whatsnew/changelog.html#changelog

[9] Pyhton pip. [Online]. Available: https://pypi.org/project/pip/

[10] Pypi · the python package index. [Online]. Available: https://pypi.org/

[11] Pyhton flask. [Online]. Available: https://pypi.org/project/Flask/

[12] Pyhton django. [Online]. Available: https://pypi.org/project/Django/

[13] Install docker engine on debian. [Online]. Available: https://docs.docker.com/engine/install/debian/

[14] J. S. Bal. [Online]. Available: https://gitlab.utwente.nl/jagvir/research-project.git

[15] subprocess — subprocess management. [Online]. Available: https://docs.python.org/3/library/subprocess.html

[16] Build and deploy a flask app using docker. [Online]. Available: https://blog.logrocket.com/build-deploy-flask-app-using-docker/

[17] Docker compose. [Online]. Available: https://docs.docker.com/compose/

[18] Deploy flask-mysql app with docker-compose. [Online]. Available: https://www.devopsroles.com/deploy-flask-mysql-app-with-docker-compose/

[19] Docker hub, mysql. [Online]. Available: https://hub.docker.com/_/mysql

[20] venv — creation of virtual environments. [Online]. Available: https://docs.python.org/3/library/venv.html

[21] Mysql product support eol announcements. [Online]. Available: https://www.mysql.com/support/eol-notice.html

[22] Writing your first django app. [Online]. Available: https://docs.djangoproject.com/en/4.2/intro/tutorial01/

[23] Django documentation databases. [Online]. Available: https://docs.djangoproject.com/en/4.2/ref/settings/#databases

[24] Vmstat. [Online]. Available: https://man7.org/linux/man-pages/man8/vmstat.8.html

[25] vmstatly by jsargiot. [Online]. Available: https://github.com/jsargiot/vmstatly

[26] 0 w value in debian. [Online]. Available: https://github.com/ColinIanKing/powerstat/issues/3

[27] Setting up a postgresql database on a raspberry pi. [Online]. Available: https://pimylifeup.com/raspberry-pi-postgresql/

[28] Raspberry pi ina219 tutorial. [Online]. Available: https://www.rototron.info/raspberry-pi-ina219-tutorial/