UNIVERSITY
OF OULU

FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Väinö Juntura

Embedded device farm proof-of-concept - Enabler for test execution on target hardware as a part of continuous delivery pipeline

Master's Thesis
Degree Programme in Computer Science and Engineering
06/2022

Juntura V. (2022) Embedded device farm proof-of-concept - Enabler for test execution on target hardware as a part of continuous delivery pipeline University of Oulu, Degree Programme in Computer Science and Engineering. Master's Thesis, 49 p.

ABSTRACT

Agile software development has produced a completely new way of working into the field of software development. The new focus is to continuously integrate, deliver, and deploy each software change. The term continuous practices is used to refer to these practices. Comprehensive testing plays a major role in so called continuous delivery pipeline.

The goal of this thesis is to implement an embedded device farm, a system which is used to effortlessly connect embedded hardware targets as part of continuous delivery pipeline. Hardware is playing a big role in embedded software development and testing. On the other hand, it is seen as a major challenge in implementing continuous practices for embedded software project. Embedded device farm is used to interact with target hardware targets by both automation systems and individual developers in unified manner. Six platforms are evaluated for the purpose and a system called Linaro Automation and Validation Architecture (LAVA) is integrated as part of existing CI/CD service.

In addition, this thesis describes the continuous practices in general introducing the benefits as well as the challenges related to implementing them. A closer look is taken into adopting the practices into embedded systems domain. Embedded systems software development differs from traditional or web software development. Embedded systems' domain specific characteristics and challenges related to continuous practices are presented.

Keywords: continuous practices, embedded systems

Juntura V. (2022) Sulautetun laitefarmin konseptitoteutus – Mahdollistaja testien suorittamiselle kohdelaitteistossa osana jatkuvan toimituksen ketjua Oulun yliopisto, tietotekniikan tutkinto-ohjelma. Diplomityö, 49 s.

TIIVISTELMÄ

Ketterä kehitys on tuonut ohjelmistokehityksen alalle täysin uudet toimintatavat, joiden keskipisteessä on ohjelmiston muutosten jatkuva integrointi, jatkuva toimitus ja jatkuva muutosten käyttöönotto. Näistä uusista menetelmistä käytetään kollektiivisesti nimitystä jatkuvat menetelmät. Kattavalla testaamisella on tärkeä rooli niin sanotussa jatkuvan toimituksen ketjussa.

Tämän työn tavoitteena on toteuttaa sulautettu laitefarmi, jolla sulautettua tietokonelaitteistoa voidaan vaivattomasti yhdistää osaksi jatkuvan toimituksen ketjua. Tietokonelaitteistolla on tärkeä rooli sulautettujen järjestelmien ohjelmistokehityksessä ja -testauksessa, mutta toisaalta laitteisto nähdään suurena haasteena toteutettaessa jatkuvia menetelmiä sulautetussa ohjelmistoprojektissa. Sulautetun laitefarmin kautta sekä automaatiojärjestelmät että yksittäiset ohjelmistokehittäjät voivat käyttää sulautettuja laitteistoja yhtenäistetyllä tavalla. Työssä arvioidaan kuuden eri järjestelmän soveltuvuutta käyttötarkoitukseen, ja järjestelmä nimeltään Linaro Automation and Validation Architecture (LAVA) integroidaan osaksi olemassa olevaa CI/CD palvelua.

Lisäksi tässä työssä esitellään jatkuvat menetelmät yleisesti, niiden toteuttamiseen liittyvät haasteet ja niillä saavutettavat hyödyt. Työssä paneudutaan tarkemmin menetelmien toteuttamiseen sulautettujen järjestelmien alalla. Sulautettujen järjestelmien ohjelmistot eroavat perinteisistä ja web-ohjelmistoista, joten jatkuvia menetelmiä ja niihin liittyviä haasteita tarkastellaan myös sulautettujen järjestelmien näkökulmasta.

Avainsanat: jatkuvat menetelmät, sulautetut järjestelmät

TABLE OF CONTENTS

ABSTRACT
TIIVISTELMÄ
TABLE OF CONTENTS
FOREWORD
ABBREVATIONS
1. INTRODUCTION 8
2. CONTINUOUS PRACTICES AND DEVOPS 9
2.1 Defining the terms 9
2.2 Benefits 10
2.3 Challenges 12
3. CONTINUOUS PRACTICES AND DEVOPS IN EMBEDDED SYSTEMS DOMAIN 14
3.1 Defining an embedded system 14
3.2 Characteristics for embedded systems domain 15
3.3 Challenges on implementing continuous practices in embedded systems domain 16
4. DESING AND IMPLEMENTATION 19
4.1 Scope 19
4.2 Requirements 20
4.2.1 Observations from future users 20
4.2.2 Requirements definition 20
4.3 Embedded device farm platforms 21
4.3.1 Search for the platforms 22
4.3.2 Commercial systems 26
4.3.3 Choice of the platform 26
4.4 System architecture 27
4.5 Implementation 30
4.5.1 Standalone phase 31
4.5.2 Connecting to the existing system 34
4.5.3 Manual connection method 36
4.5.4 Testing 37
4.5.5 Documentation 38

5. DISCUSSION AND FUTURE WORK 39
6. CONCLUSION 41
7. REFERENCES 42
8. APPENDICES 46

FOREWORD

I would like to thank my employer for gratefully offering me the opportunity, time allocation, and subject for this thesis. I also thank my colleagues for their support and expertise during the process as well as understanding towards my absence from my daily tasks.

I would like to thank my fellow students for their support during this thesis and during the studies overall. I am also thankful for my lovely wife for supporting me on the process of writing this thesis and understanding towards my long nights spent with this document.

Oulu, 5.6.2022

Väinö Juntura

ABBREVATIONS

API	Application Programming Interface
CaC	Configuration as Code
CD	Continuous Delivery
CDE	Continuous Deployment
CI	Continuous Integration
CI/CD	Continuous Integration and Continuous Delivery
CLI	Command Line Interface
COTS	Commercial off-the-shelf
CP	Continuous Practices
DS	Distributed System
DUT	Device under Test
FOSS	Free and open-source software
ID	Identification
IO	Input/Output
IoT	Internet of Things
LDAP	Lightweight Directory Access Protocol
MVP	Minimum Viable Product
OS	Operating System
PC	Personal Computer
PoC	Proof of Concept
PSU	Power Supply Unit
QA	Quality Assurance
ROI	Return of Investment
SaaS	Software as a Service
SD	Secure Digital
SSH	Secure Shell
TDD	Test Driven Development
UI	User Interface
URL	Uniform Resource Locator
USB	Universal Serial Bus
WoW	Way of Working

1. INTRODUCTION

It has been already 20 years since the manifesto for agile software development was published [1]. During those two decades, agile software development has become one of the most important software development methods. The manifesto was published to streamline the software developing processes. On the very first sentence of the principles behind agile manifesto, the term continuous delivery is mentioned. The manifesto claims the ability to continuously deliver the software to be the highest priority to be able to satisfy the customer.

Nowadays, continuous delivery is a development practice in which each software change is treated as a potential release candidate [2]. An abbreviation CD is usually used for continuous delivery. The term CD is many times complemented with a term CI, together constructing something called CI/CD. CI stands for continuous integration, which is a developer practice impressing developer actions to integrate their software with mainline on regular basis.

Implementing so called CI/CD in practice usually includes an automated build and test environment which includes automated integration and delivery methods. The main goal of CI/CD is to reduce integration problems as well as risks on a software project. It usually also makes detection of bugs easier and faster as well as makes the transition to continuous deployment (CDE) achievable. [3] Continuous deployment on the other hand is an operations practice focusing on deploying the software to the production environment on continuous manner. CI, CD, and CDE combined is often referred as continuous practices. The latest vogue word related to continuous practices is DevOps, which is more abstract, but usually refers to mindset and culture rather than anything concrete. The terms are further defined in chapter 2. [2]

While the discussion around continuous practices has been a hot topic in the software development field already for relatively long period of time, in the embedded systems domain it has not got so much attention yet [4]. Embedded software differs quite a lot from traditional computer or web software due to nature of its domain. There are domain specific factors which have been hampering the transition to continuous practices in embedded systems domain.

The goal of this thesis is to implement a proof of concept (PoC) system to facilitate an often-faced challenge of building a comprehensive automated testing system for continuous practices in an embedded software project. The PoC implementation offers a solution for connecting different types of embedded hardware to be part of an already established continuous delivery pipeline in a unified manner.

In addition, methods, benefits, and challenges of continuous practices are described generally, as well as more specifically in the embedded systems domain. Common challenges and barriers on the road of implementing the practices are presented. Deeper look is taken into the challenges which apply particularly for embedded domain.

2. CONTINUOUS PRACTICES AND DEVOPS

Agile software development has been changing the software development field since early 2000s. However, the term continuous integration (CI) was introduced way earlier, in 1991. It became popular as part of the Extreme Programming practices around 1997. The term indicates a practice, where developers integrate their software frequently with mainline. Afterwards, the term continuous delivery (CD) has often been combined with CI. CD is a practice to frequently release the software. [5]

Approximately 20 years after the first introduction of CI, the term continuous deployment (CDE) has been introduced keeping the revolution going on the software industry. CDE aims to introduce the software releases frequently to be able to adapt the development according to the usage [6]. Above mentioned three terms are often combined and called simply continuous practices (CP). Important focus area on CP is to shorten the feedback time and deliver software to the customer quicker. According to the agile manifesto, that is a core principle of agile software development [1].

Nowadays, most of the companies working in the software industry are executing CP in some form or are at least planning to do so. Well known companies implementing CP include for example Facebook, IBM, Netflix, Google, and Red Hat Software. [7]

DevOps is newer topic, and it is an evolution of agile movement. The term was first introduced in 2009. DevOps thinking has two main principles. First one can be interpreted from the term itself: collaboration between development and operations. Another one highlights the usage of agile methods in collaboration with automation to manage environments and configurations. While agile embraces constant change and close co-operation with customer, DevOps focuses on constant testing and delivery and the collaboration with the development and operations teams. [8]

This chapter presents the terms related to the topic, their numerous benefits as well as known challenges.

2.1 Defining the terms

Continuous integration, continuous delivery, continuous deployment, continuous practices, and DevOps. These terms are trending in the software development industry as well as amongst researchers currently [2]. However, the definition for the terms is somewhat ambiguous, relation between the methodologies is unclear and different terms are often used interchangeably, even though they have different meanings. This paragraph tries to reveal the difference between the terms and define them explicitly.

Continuous practices is the most evident of the terms. It is a superset of other terms having the word continuous in the beginning. The three most popular continuous practices are the terms already mentioned above, continuous integration, continuous delivery, continuous deployment. There are also other continuous practices, but in the sake of clarity, they are not covered in this definition.

Continuous integration is a developer practice. Developers executing CI integrate their work on regular basis with the mainline software. The pace can vary, but usually there is multiple integrations per day. CI is considered as developer action instead of development action because, despite having all the capability to integrate continuously, individual developers do not necessarily integrate their work frequently enough to call the process continuous.

Instead, continuous delivery refers to a development practice. CD considers each code change as a potential release candidate. Usually a CD pipeline exists, which runs regularly for release candidates to assess the changes. In CD, there should always be ability to release an up to date, working version of the software. In practice, however, it is usually not feasible to execute the whole pipeline for each change, so the changes might get combined within the pipeline. Even though generally the term CI/CD is used, CD does not necessarily incorporate CI, since CI is a developer action. An organization can have fully working CD, even though all the developers had not adopted CI practices.

Continuous deployment is seen to be an operations practice. CDE is a set of actions where the output of CD is deployed to the production environment. Deploying to production environment usually means making the software available for end users, but this may vary depending on context.

Definition for DevOps is the vaguest amongst the terms described in this paragraph. Software development field is nowadays wide, and many opinions exist on the definition for DevOps. It is often presented to be a culture or mindset, but it actually is more like a superset of continuous practices, relevant tools, processes, methods and principles. It offers practitioners ideas on how to carry out their work, while also offering concrete examples and tools. This thesis concentrates more on CP over DevOps, still not completely leaving it out.

2.2 Benefits

The main motivation for using CP is the benefits it is offering. The competition in the software development field is fierce, forcing companies to be agile, efficient, and reliable. This chapter presents the benefits which can be gained by implementing CP. Benefits related to CP are presented in multiple papers [9, 10, 11, 12].

One of the most visible advantages of CD is the accelerated time to market for products. Compared to traditional way of releasing the software, there is a dramatical increase. Studies show the release cycle to be over twice more frequent in average when CD is used compared to traditional releasing. Also, there is high possibility to decrease the cycle time for a feature request to reach the production. In practice, faster time to market means that the customers get the value more quickly. This is seen as a business advantage as well.

Improved product quality is also significant benefit of CP. The number of open bugs decrease mostly due to comprehensive and automated testing. Testing is often a monotonous but precision-demanding process. Humans are not particularly talented to

execute such tasks on regular basis. Automated system always runs the tests with the same accuracy while us humans tend to make mistakes by our nature. Automated testing leads the bugs to be discovered quickly after code changes have been applied. The bugs are more often found before the developer goes into next task, enabling the developers to directly fix them without a delay. Also, in the rare case that the bug is found not until in the production, the next release will follow soon so the bug can be corrected shortly.

The productivity and efficiency of the software teams are also considered to be better when CP are used. This is the outcome of multiple factors. Less time is used in many different repetitive and monotonous tasks. Less time is wasted on fixing developers' build and test environments, because CI/CD environment offers a consistent platform for building and testing. CI is helping developers to avoid applying code that is going to break the software allowing them to proactively fix the problems. Code reviews are also faster since reviewers are more comfortable to accept changes when CI system has verified them. In addition to that, people from outside the projects often feel more comfortable to contribute the code when there is a CI environment in place verifying the changes which adding agility and efficiency. Time is also saved on release activities, which without CDE would include manual efforts. Manual efforts often also lead to even more work since humans in general make more mistakes than machines. Time used for fixing those mistakes is saved because they do not happen at first place.

CP can also help on building the right product improving the efficiency. Frequent releases lead to faster user feedback, which helps the team to prioritize and adapt the efforts more rapidly. This is saving time from developing unnecessary features. Organizational independence can also increase after implementing CP. The release and deployment phases are traditionally executed by specific people in the organization having the needed skills and knowledge. Properly implemented CDE system allows any developer to deploy the software into production.

CDE is usually enhancing release reliability and reducing the risks of release process. Again, this is an outcome of multiple factors. When implementing CDE, release process is automated. Tests are implemented for the process and even a production-like staging environment can be used to verify the release before deploying it into the production. Manual steps are avoided, so there is less room for human errors. Due to frequent releases, the delta between the releases is smaller. That lessens the probability of integration problems while making it easier to debug and find out the problem if such happens. In rare case of failure in the release, there are often automated rollback features implemented to effectively recover from such occurrence.

The above-mentioned benefits have also positive side effects. Reliable processes reduce the amount of stress the developers are facing in their everyday work. Also, the amount of repetitive and menial tasks for the developers is lower. The job satisfaction and work morale of the developers have been improving after implementing CP. Another positive change is that the customer trust towards product or supplier is often increasing. Fewer issues in the release process and the software itself often leads to happier customer.

2.3 Challenges

Apart from vast benefits CP is offering, there are considerable challenges involved. Challenges are varying from enterprise level challenges into monetary, process, and technical challenges. Some of the challenges are relevant for CP in general, while others are characteristic for either CI, CD or CDE. Numerous papers are presenting the challenges faced during implementing CP [9, 12, 13, 14, 15].

Studies have proven that transition to CP is not a straightforward change in organizational level. Challenges related to organization structure and way of working (WoW) is often faced during that activity. On bigger organizations, multiple divisions having their own established practices are usually involved together in release activities. Successfully implementing CP requires strong co-operation and visibility across the different teams. Traditionally however, there might be tensions and competition between the teams. Change is needed from competitive mindset into a collaborative culture. This has proven to be even more challenging in distributed development organizations.

In already established organizations, there are already formulated processes for development practices. Big change might be needed on the processes, especially if agile methods are not already present in the organization. Traditional software development processes are not very well compliant with CP and are often hindering the change towards CP. Changes are needed for example from long-living branches into small software iterations. Another often mentioned process challenge is related to quality assurance (QA) processes. Traditional QA process might be too time-demanding or requirement specification process not flexible enough for CP.

There is often change resistance involved on both organizational and process changes. Humans tend to resist change by nature. Successfully implementing CP may require a significant change on the WoW also on individual level. Completely new mindset might be needed along with new working habits, taking time to adopt and asking trust to the end goal to be successful. Often faced individual level barriers on the way to CP are skepticism towards added value, distrust on the practices, and lack of motivation.

Technical challenges are often faced while implementing CP. The reality is that a production ready, customizable, reliable, and commercially available all-in-one CI/CD system does not yet exist. That is forcing the companies to invest money, time, and effort to build such system themselves. Calculation of return of invest (ROI) for the investment on CP is known to be challenging, which might add resistance for the investment in management level. However, it has been studied that implementing CP generates 19 percent increase in revenue in average [16].

CI/CD systems are often a combination of wide range of different tools and technologies. There are both risks and challenges related to the system itself. Common risk is the possible vendor lock-in. There are competing tools in the market and not all of them have open application programming interfaces (APIs) for example. Challenges are also related to the investments on the systems. Monetary investment is needed for example

to upgrade infrastructure and to train personnel. Investment is needed on personnel skills, since many new technologies needs to be adopted during implementation of CP.

More practical technical challenges include for example testing, integration, and build system problems. Common integration problems often mentioned on the literature are merge conflicts and too large commits. Testing problems include too complex testing systems and particularly long running or constantly failing tests which are preventing rapid and accurate feedback for the developers. Build system problems can originate from lack of automation or lack of needed resources to execute builds.

Customer originated challenges are also relatively common aggravating adoption of CP. Customers might be unwilling to adopt continuous updates due to earlier experience on poor quality releases or due to not realizing the added value of continuous updates. There might be also dependencies to customer environment especially on deployment phase. Customer environment might be unreachable or there could be unique environments for different customers making it hard to generalize and automatize deployment process. Customers can also be unwilling to share the usage information used on monitoring and improving the software and deployment process.

Domain specific challenges also exist. Domain of the software is playing a significant role on adopting CP. For embedded systems for example, additional hardware might be needed during delivery and deployment process. Challenges specifically related to implementing CP on embedded systems domain will be addressed comprehensively in the following chapter.

3. CONTINUOUS PRACTICES AND DEVOPS IN EMBEDDED SYSTEMS DOMAIN

Discussion around CP and DevOps is concentrated on traditional software development and especially on web software development. The development process in such a project differs from an embedded systems project. The web domain projects are highly concentrated on the software development only, while in embedded domain there are constraints like mechanics and electronics that needs to be taken care as well as the software. [4, 12]

The eventual use case of an embedded system usually differs a lot from pure software product. In embedded systems domain, the end products are usually owned by customers, who have purchased the product. For web development, the situation might be the opposite. Often seen operating model in web domain is Software-as-a-Service (SaaS), where software is owned by software provider and customer is only buying the service that the software implements. [6]

While most of the practices from for example web domain can be adopted to the embedded systems domain quite easily, some practices need contextualization and further adaptation. This adds additional constraints to the design of the continuous workflow for embedded systems. Overall, it is seen to be more challenging to successfully implement CP to an embedded systems software project than for a web or cloud-based software project. [4, 6]

3.1 Defining an embedded system

Embedded system is a combination of software, computer hardware, and possibly mechanics, electronics, and other peripherals serving a dedicated purpose. An embedded system is subject to physical constraints. There are two kinds of interaction between an embedded system's computational process and physical world: it needs to be able react to changing environment and it is executed on physical platform. The difference to for example personal computer (PC), is that PC is meant to be a general-purpose device letting the user decide what to do with it. Embedded system however executes a specific function. [17, 18]

There are also other terms often used to refer to similar kind of system, for example cyber physical system, internet of things system or ubiquitous computing system. Scope in those terms is mostly the similar as for an embedded system, but the details differ slightly. In this thesis, only the term embedded system is used for clarity.

3.2 Characteristics for embedded systems domain

There are some characteristics for embedded systems domain projects which differentiate them from web or traditional software projects. In this chapter, characteristics of embedded systems are presented from software testing point of view. The main difference in embedded software project compared to traditional software project is that the software cannot be fully tested on the same machine that it is being built on. From CP point of view this means that the software needs to be tested also in so called target hardware. The term target hardware refers to the hardware that the software will eventually end up in the production.

Virtualization technologies are widely used in for example web software domain to abstract the hardware layer. For embedded domain, similar approach is to use emulators which in some cases lessen the need for testing on physical target hardware. However, not everything can be emulated, and target hardware still plays a major role in embedded software development and testing. [4, 20]

Great amount of embedded software projects is built on top of custom hardware. That will usually lead into a situation in which the hardware is not yet ready when the software project starts [19]. The hardware can also be relatively expensive compared to commercial off-the-shelf (COTS) products due to for example small production patches. This might limit the amount of hardware available for software testing. On the other hand, development boards and emulators can often be used especially on initial phases of the project for software testing to replace the actual hardware target. [20]

Compared to testing in a traditional software project, there are both similar and different kind of limits hardware-wise for embedded systems software testing. Similarly, the computing resources are always limited. When it comes to embedded devices the limitations are sometimes tighter due to nature of the hardware. Especially for IoT devices, computational capacity available is often very limited. Reasons for that are multiple. The restrictive constraints usually originate from the intended use case of the device and can be for example cost, physical size, or power consumption limits. [21]

Earlier, very characteristic feature for an embedded system has been the fact that once the device leaves production line, it is never updated or updating is done by dedicated technicians [4]. Companies working in embedded systems domain have traditionally had hardware and product oriented operating model. Due to all the time increasing digitalization, however, the focus is shifting towards selling services and solutions rather than products forcing companies to rethink their whole operating models. This includes also adopting continuous practices, especially CDE. [22]

From organizational point of view, the embedded software domain somewhat differs from web software domain. In the web domain, the term full stack developer is widely used [23]. On the contrary in the embedded domain, the development often needs to be separated into two departments being software and hardware development. Many special skills are needed on both, which makes it hard to combine them. In that scenario, visibility within the project often decreases and more communication is needed. [4] Also, engineers

in embedded systems domain have traditionally favoured waterfall model over agile development which is hampering the transition to CP. However, the movement towards agile methods among embedded software development has been strong lately. [24]

3.3 Challenges on implementing continuous practices in embedded systems domain

While most of the general challenges on adopting CP presented in the chapter 3 also apply for the embedded systems, there are also multiple domain-specific challenges in addition to those [4]. Starting with the tools. As mentioned, the literature on CP is concentrated on web software because the practices are most adopted in web software domain. The same applies for the tools needed on implementation of CP. There are numerous open-source tools available to implement CP and automation overall on web software project. Some of them can be used also in embedded domain. However, the domain specific requirements especially for the deployment phase create a need for different kind of tools. There is still a lack of those tools which is raising the threshold for adopting especially a CDE system. One reason behind the shortage is that the type of embedded system projects varies a lot, and the same tools cannot possibly be used across different projects without major modifications.

Another limiting factor for successfully implementing CP in embedded systems domain is that embedded development organizations are often divided into silos. One fundamental idea behind DevOps is to bridge the silos between development and operations. Also, agile principles encourage to rather work on feature teams in which everyone has the visibility on the whole development process. Organization models in the embedded domain are often module team organizations, which are quite an opposite to cross-functional feature teams. However, certain constraints in the embedded domain force the organizations to module team setup. There are often very specific skills needed in different areas of embedded software project, especially when moving closer to the hardware. Overall, the hardware development is usually strictly differentiated from the software development. In addition to lack of knowledge and visibility, the systems are often quite complex which makes it harder to quickly propagate the code changes across the module teams. Visibility and cross-functionality between the module teams needs be increased to add agility and achieve the readiness to move to CP.

The nature of the customer environment sometimes shows up as a barrier to implement continuous practices into an embedded domain project. There are several different possibilities which could cause challenges in this matter. Firstly, there might be limited visibility to the customer environment, which makes it hard to implement a working CD system. Apart from the lack of visibility, there can be different kind of customer environments for the same product due to the fact that the same product is sometimes sold to different customers in different variants. The product which is sold can also act as a platform on top of which customer builds their own system. The general problem in the cases mentioned above is that the final production environment for the product can vary a

lot even for the same product. That makes it challenging to implement a general CD system, since the system would need to be tailored to many different environments containing multiple different configurations. It has also been discovered that some customers in embedded domain have “you don’t fix what ain’t broken” mindset which obviously makes it harder to reason a CD system.

Another challenge for implementing CP is the existence of legacy code and legacy systems in general. This has been identified to be a problem particularly when it comes to the embedded systems domain. The reasons for that are for example that sometimes the systems are built on top of legacy systems. The initiative for such behaviour can originate both from customer and from the supplier. Customer could wish for example the new system to be a part of old system while the supplier might want to reuse some already existing solution to keep the effort minimal. There are sometimes also very long maintenance contracts for the systems in embedded domain which can lead to a considerable amount of work on a legacy system. In such scenario it might be beneficial to have the CP in place, but it is not feasible to apply them on the old system which has not been originally implemented having that aspect in mind.

When it comes to CDE, there are couple of challenges that are characteristic for embedded systems. In embedded systems, the hardware on top of which the software is eventually run on is not always a COTS product. Sometimes it might take very long time to get the actual hardware due to hardware designing being a tedious process, often taking multiple iterations to get right, and the lead times for the custom hardware being long. In the past year we have also seen that the global market for the semiconductors is a fragile business which has also caused some delays in the hardware delivery process [25]. Fortunately, both virtualization and evaluation boards can often be used before the actual hardware is ready to make the process quicker.

Also, the deployment systems in embedded domain are often very complex and might sometimes be difficult to automate. The dependency between the hardware and software places a challenge also for the deployment. While virtualization technologies are widely used for example in the web software domain during the deployment phase to abstract the hardware layer, it is not similarly possible for embedded systems due to the nature of the systems. In addition, embedded systems are sometimes used critical solutions where almost 100% uptime is required. In such scenario CDE is not an option. As an example, paper machine cannot be stopped due to software update every night. On the other hand, it is a matter of missing mechanisms, which usually is highly related to the legacy that many products are carrying. [12]

The challenges mentioned in this chapter are in some cases leading to partially or even completely missing CI/CD systems. DevOps mindset stresses the importance of the production-like testing. However, it has turned out that building such systems in embedded systems domain could be challenging. This is also leading to manual, non-comprehensive testing, which in turn may lead for example to the delivery of poor-quality software. For example, it has been discovered that in the embedded domain, the bugs and faults are often found not until in the customer’s tests or even in the production.

Considering the nature of the software delivery process in most embedded systems, it would be very beneficial if there were mechanisms to detect the problems earlier in the supply chain.

4. DESING AND IMPLEMENTATION

Building a comprehensive automated testing system in embedded systems domain is a common challenge during implementing CP. To lower the threshold for moving to CP in an embedded software project, a PoC implementation of an embedded device farm is presented in this thesis. The device farm is used to connect different kind of embedded hardware to existing CI/CD service in a unified manner.

The organization for whom this thesis is carried out for has recognized the value of CP. Due to that, there is already existing implementation of an enterprise level centralized CI/CD service. However, a need for a system for easily utilizing embedded hardware as part of the CI/CD service had been identified in the organization. During continuous improvement actions, the projects using the service had been proposing such feature to be added to the system. The existing approaches to solve the problem across the projects were varying quite a lot from not having any system to manually connecting each hardware or even using commercially available solutions.

Being an additional part of already established CI/CD service places additional requirements for the design and implementation. Also, due to the size of the organization using the system, the solution should be able to both scale up and adapt to different kind of requirements.

The main goal of the PoC is to unify the methods used to utilize embedded hardware as part of CI/CD pipelines within the organization, while keeping most of the functionality of various existing systems. Having a unified solution should also lower the threshold to initially utilize the needed hardware in the automation system overcoming one barrier in the way to CD. Also, in terms of agility, it is emphasized that developers need to be able to move across different projects within the organization effortlessly. To make that transition easier, the processes and tools used across different projects should be as similar as possible. This implementation is intended to be used as a common tool within the organization to unify the usage process of embedded hardware in CI/CD pipeline.

4.1 Scope

Already during refining the topic for the thesis, it turned out that it will be necessary to scope the PoC solution quite precisely to avoid the project to swell too much. Similar approach to the problem had not been tried on the organization before. The goal was decided to be a minimum viable product (MVP) to prove if implementing such system on enterprise level would be possible in first place. During requirement definition, it was initially determined on what requirements will be in scope for the PoC, keeping in mind the MVP scope. A closer look on the requirement definition will be taken later in this chapter.

4.2 Requirements

This paragraph describes the process of defining the requirements for the embedded device farm. The requirements are presented in table format also containing the information about PoC phase scope.

4.2.1 Observations from future users

A need for a unified method for utilizing embedded targets through existing CI/CD service had come up in numerous discussions with existing users of the service. Even though the concept and implementation idea were rather clear, it was seen beneficial to discuss with the possible future users of the system before defining the final requirements. Bidirectional knowledge transfer sessions were arranged with three projects that were using the service and had embedded hardware as a part of their development workflow.

The sessions were not planned to be too formal since the goal was just to gather some additional information on top of the existing idea of the system. Anyway, the following topics were covered in all the discussions. First, the concept of the new platform was introduced to the project. After that, the project introduced their existing solution to the problem, if there was none. The embedded hardware used by the project was listed along with possibly needed peripherals. The projects were also asked for wishes and suggestions for the embedded device farm.

The initial reception for the concept was positive within all the projects. A lot of useful information, some additional requirements and useful suggestions were gathered. Information about the existing solutions was particularly interesting to avoid lack of crucial features on new system to be designed. Along with existing knowledge and vision, a specification of requirements was defined also considering the newly gathered information.

4.2.2 Requirements definition

System requirements for embedded device farm are presented in Appendix 1. For each requirement, the table contains requirement ID, both short description and additional information of the requirement as well as use case, priority, and the information whether the requirement is going to be in scope of the PoC solution. Requirement ID is a unique identifier for each requirement. Main requirements are marked with ID in format RQx, while sub-requirements in format RQx.y. For main requirements, blue background colour is used to achieve better readability of the table. Description field defines the requirement itself and the additional information adds some details on top of the description if it was seen necessary. Some of the requirements are rather technical while others rather abstract.

Use case field was decided to be added to the table to concretize the need for individual requirements.

The requirements were prioritized into three categories: low, medium, and high. Even though priority affected to the choice whether certain requirement is on scope for the PoC phase, priority alone was not directly translated into that information. As the goal in the first phase was to create an MVP, strict scoping was needed which led to adding a separate field to the table for the PoC scoping.

In addition to the priority, there were also other factors affecting to the scoping. For example, there were dependencies between requirements. Also, the choice of the platform to be used as a base for the implementation affected on the scoping. That is why being in scope for the PoC was also categorized into three categories: yes, possibly, and no. Possibly-alternative was added, because feasibility and effort estimation of some requirements was highly dependent on the platform which would be selected to be used in the implementation. For some requirements, possibly-alternative was used, because the requirement would be partially fulfilled during PoC.

Requirements listed in Appendix 1 are mostly self-descriptive, but some highlights are still worth mentioning. For example, different connection methods were introduced in the requirements to achieve wide adaptability in enterprise level. In addition, support for many peripherals was introduced and scalable distributed architecture overall was set as target. Even though generally a goal in CP is to automate everything, a possibility to manually connect to the target hardware for so called debugging sessions was also presented in requirements. Importance of proper documentation was also underlined.

Configuration as code (CaC) was emphasized across the requirements. CaC is a convention in which configuration of software components is done via version-controlled configuration files enabling automated deployment of the systems [26, 27]. It is very commonly executed practice especially in CDE. CaC was introduced in the requirements to achieve scalability and traceability on enterprise level more easily, and to configure the device farm in standardized manner also enabling automating the deployment steps.

4.3 Embedded device farm platforms

Writing a comprehensive device farm system from the scratch would be a big and complicated task and it would be out of scope of this thesis. Fortunately, various platforms implementing the required features already exist. There are multiple advantages on using a ready-built system. For example, many of the systems have been under development for years already, which make them more complete compared to a self-made system. However, there is also disadvantages. There might for example be limitations or missing features for specific use cases on these systems. However, the most of them are free and open-source software (FOSS) enabling everyone to modify them for their needs as well as contribute on development.

4.3.1 Search for the platforms

After defining the requirements for the embedded device farm system, a throughout study on suitable platforms enabling the implementation of device farm was carried out. The best single source of information about available systems turned out to be eLinux wiki [28]. eLinux stands for embedded Linux. The wiki has a specific page for test systems suitable for embedded Linux development. The page has plentiful of generic tools, but also references to systems, which would serve the purpose for the embedded device farm. Apart from the eLinux wiki, the search for suitable systems was conducted all over the net.

Multiple decent candidates were found, Table 1 lists all the possibly suitable systems discovered during the research. These platforms were examined further and on the following paragraphs, all the systems considered as a potential candidate during initial study are described more in detail. Also, suitability for the device farm PoC implementation is considered. The examination was conducted quite precisely to have comprehensive understanding on the available systems.

Table 1. Potential platforms to implement an embedded device farm

System
LAVA
labgrid
r4d
hottest
Opentest
Fuego

4.3.1.1 LAVA

LAVA is an abbreviation for Linaro Automation and Validation Architecture [29]. The name itself is quite descriptive. LAVA is a system used to deploy operating systems (OS) onto physical and virtual hardware to conduct tests on them. It is not a complete CI/CD system but is meant to be used as a part of a CI/CD system. The communication with LAVA takes places through CLI client, making it easy to be implemented as part of other systems. It can also be controlled over graphical web interface. Also, it is not a test suite, rather an infrastructure to be used to conduct tests on the embedded (Linux) hardware. Virtually, that means that LAVA is not bothered what is going to be run on the devices connected to it. It is just offering the connection methods. That makes it good candidate for the job, since in many scenarios the tests already exist but a proper platform to connect target hardware to test automation is missing.

LAVA implements a scheduler, which makes it aware of the state of each target connected to itself. Scheduler is also capable of parallelisation of the jobs to multiple targets. According to the documentation, LAVA can also be used to manage and share targets within the developers. It has a capability for so called Hacking Sessions, which allows developers to communicate the boards over the same API that the CI/CD system does. The scheduler is aware of those sessions, making it convenient for the developers to use the same hardware for their debugging purposes alongside automation systems.

LAVA has a master – worker architecture, which is designed to be used also on distributed manner. In practice that means that there could be multiple device laboratories communicating to the rest of the CI/CD service through one master node.

LAVA project is originally started by Linaro, but today it is maintained and developed further by LAVA Software Community Project allowing anyone to propose changes to the software. The project is well established as well as documented. There is also a new version of documentation under development [30].

The challenge with LAVA from this PoC's point of view is that it is originally built for kernel testing. The priority in device support architecture is on embedded Linux devices, but there have been also non-Linux devices connected to it successfully in past. The device farm shall preferably support also other than Linux based boards. Another con is that LAVA does not implement support for remote controlled PSUs or other peripherals, so they need to be used via build scripts for example.

4.3.1.2 labgrid

labgrid is an embedded board control system, which focuses on testing, development, and general automation [31]. labgrid has a remote-control layer designed to control hardware boards connected to separate host computers. Fundamental idea of labgrid is to abstract the hardware control layer for testing and for other automation where target hardware is accessed. Development and maintenance of labgrid is active. It is FOSS, on which everyone can contribute. labgrid has been developed already for five years and it is quite comprehensive system with well-organized documentation.

The core of labgrid is a remote client-exporter-coordinator architecture, enabling communication with target boards connected to it all over the network. For the target connection, it supports interaction with bootloader as well as Linux shell on top of either serial console or SSH connection. labgrid also implements support for different kind of peripherals often needed during automated testing. It supports target power and reset management using power switches. It has control over digital outputs, SD cards and USB multiplexers. It has also integration for adding audio and video measurement devices. At least during the PoC phase of this embedded device farm, there is no need for all of them, but later in the production such demand can arise.

Downside of labgrid is that even though it is not a test framework itself, it is somewhat built to be mainly used with pytest test framework. On the other hand, it has the CLI which

can be used instead. Another negative aspect on labgrid is the scheduler. It implements methods to reserve boards which would come handy in embedded device farm PoC, but a proper scheduler is not present in the system. The scheduling could however be implemented in other parts of the CI/CD service, so lack of it is not a showstopper.

4.3.1.3 r4d

r4d is a shorthand for Remote for Device-under-test [32]. The project is rather minimalistic and the functionality of it can be summarized into singe sentence: r4d is an infrastructure used for controlling power and accessing console for multiple embedded Linux boards through a CI/CD system. That is basically what is the main goal of this embedded device farm also.

r4d is working as a controlling server for an embedded device rack. It is supposed to use a dedicated serial console server for the connection to the devices and a power switch for power control, which means additional appliances needs to be acquired. Out of the box, it supports only one specific model of serial console server and two different power switches. It is a clear downside of r4d that it is bind into dedicated lab hardware.

r4d has a basic command line interface for operating the system. As mentioned, the system is minimalistic and, even though the scope would be perfect for the use case, it is too basic to be selected as base for this PoC. The project has neither had any activity after October 2020. This PoC is intended to be used far into the future, so preferably the underlying platform should be maintained actively.

4.3.1.4 Hottest

Hottest is a system used to test Linux-based firmware especially in target hardware [33]. The main idea of Hottest is to use Jenkins as a base for the project. Jenkins is used as a scheduler as well as interface for the whole project. There is no proper API for the system, which makes it difficult to integrate it to be part of CI/CD loop. It is rather designed to be used as a standalone automated testing solution.

Hottest offers serial communication to target hardware, but one key design principle is to leave the device setup operations for the user. Hottest documentation focuses on the tests itself, which are written in shell scripts.

Hottest is made by HMS networks, but it is an open-source project. The project is inspired by LAVA and Fuego. Hottest is small project and it is no longer developed nor maintained. The last changes to the source code are from year 2019. Documentation for Hottest is relying on git README, which is not very comprehensive.

4.3.1.5 Opentest

Opentest ties together two existing projects called STAF (Software Testing Automation Framework) [34] and TestLink [35] while adding some extra features on top of them to accomplish a complete test automation framework designed specifically for embedded systems' use case [36]. As the name already tells, STAF is a software testing automation framework. TestLink on the other hand is a test management and execution system.

The architecture of Opentest allows a test system to be installed on a single machine, while also making it possible to scale the system up to for hundreds of machines. It implements a scheduler which is aware of the resources attached to the system. It has both web interface and CLI for test execution. These are features which would make it a possible candidate for implementing this device farm PoC.

Being a complete test framework means that the tests itself are defined within the system. That is not an ideal situation for this PoC, since the goal is not to bind the system into any test framework. The objective is to let users decide on the testing part and rather only offer an infrastructure to easily run utilize embedded targets during testing. Opentest supports many kinds of test and performance report generation out of the box.

Opentest project has been originally started by Texas Instruments around 2009. It is an open-source project allowing everyone to contribute. In recent years, however, the project has not got so much further development neither maintenance. Last commits to the source code are from 2020. Documentation for the project is comprehensive, but it also seems obsolescent, and the project overall seems deprecated. Opentest architecture as well as setup process are also a bit complicated compared to other systems.

4.3.1.6 Fuego

Fuego is a test automation system which is designed for embedded Linux testing [37]. Fuego enables automated software testing on embedded target hardware. It is a complete CI system using Jenkins as a core. Fuego also comes with some ready-packaged tests for embedded Linux targets. The main idea behind Fuego is that it can be used as a CI solution for embedded Linux project out-of-the-box. The whole system is containerized to a Docker container. That is quite far away from the scope of this embedded device farm PoC.

Fuego project has originally been started by Sony Corporation. Nowadays, it is an open-source project. The system is quite extensive, but in recent years, the development around it has decreased. Having such wide scope, the system might be too comprehensive for the use case of this project. Also, being designed to be used as complete CI system, it is not the perfect choice to be used as a part of CI loop. However, it has APIs, and there are known use cases where it is used in such scenario [38].

4.3.2 Commercial systems

Even though the main goal for this PoC was to use a freely available system as a base, also commercial systems were briefly explored. Turned out there are not too many of suitable systems on the market. Analysing the reasons thoroughly for that is beyond the scope of this thesis. However, the possible factors can be for example too small size of the market leading to non-profitable business. Another factor could be the broad scope, yet high customisability and scalability requirements for the system making it hard to develop a comprehensive commercial system. Outcome of the above factors can also be seen from some of the FOSS systems presented in Table 1. Some of the systems are not comprehensive enough, but rather implemented for a single use case while others are perhaps little bit too "hacky" for being a commercial product.

An example of a commercially available system is a product called EBF (Embedded Board Farm) from a company called Timesys [39]. It is relatively new product, launched in June 2021. EBF offers a customer-owned continuous testing infrastructure to remotely access target hardware during development, debugging, and continuous integration. It offers an open-source API to access target boards and other lab equipment in standardized manner. It also integrates directly with some test frameworks.

EBF has a lot in common with for example LAVA and labgrid. Hence, it would also be suitable to be used as a basis for this PoC, but as mentioned, the goal was to use a freely available system.

4.3.3 Choice of the platform

All together six open-source systems were reviewed to evaluate the suitability of them to be used as a base for this embedded device farm. Once the research through suitable systems had been carried out, the next task was to compare them and decide the platform that would be used in the PoC implementation. Systems were compared with each other as well as assessed against earlier defined system requirements. Naturally, the most important factor for the choice was the compliance with the requirements. The effort estimation for making individual systems to fulfil the requirements was also considered during decision making process. Other things affecting to the choice were licensing, customizability, and scalability. The latter two of which are highly dependent on architectural design of the system. Therefore, suitability of the system design and architecture of each system was also closely estimated. Table 2 summarizes the evaluation of each individual system by factors affecting the suitability.

Table 2. Overview of FOSS device farm platforms evaluated for the PoC

	LAVA	labgrid	r4d	Hottest	Opentest	Fuego
Can be used as part of CI/CD loop	Yes	Yes	Yes	Partially	Yes	No
Does support variety of test frameworks	Yes	Partially	Yes	Partially	No	No
Has API/CLI	Both	CLI	Partially	No	CLI	API
Configuration done as code	Yes	Yes	Partially	Partially	No	Partially
Implements scheduler	Yes	Partially	No	Yes	Yes	Yes
Implements manual connection method	Yes	No	No	No	No	No
Enables distributed scaling	Yes	Yes	Partially	No	Yes	Partially
Comprehensive documentation	Yes	Yes	No	No	Partially	Partially
Supports needed lab peripherals	Partially	Yes	No	No	No	No
System is under active development	Yes	Yes	No	No	No	Partially
Licensing is suitable	Yes	Yes	Yes	Yes	Yes	Yes

Examining the table reveals that LAVA and labgrid clearly stand out from the rest of the systems. They are the most complete amongst the systems having active development around them. In terms of system design and the intended use case, they are closest to desired. Therefore, LAVA and labgrid were taken into further analysis and comparison between each other. In the end, they are quite similar in many ways both fulfilling most of the requirements, which made the decision harder.

After considering all the aspects, LAVA was selected to be used in this PoC. As mentioned, the difference to labgrid was minor, but some things still spoke for LAVA. For example, labgrid's pytest centric mindset and lack of proper scheduler were seen as disadvantages in labgrid compared to LAVA. Also, general impression on LAVA was slightly better.

4.4 System architecture

Being part of an already established CI/CD service places additional constraints for the system architecture of the embedded device farm. The main goal for the device farm was to unify the methods how embedded targets are utilized by existing CI/CD service. During initial brainstorming on the embedded device farm before further requirement definition, Figure 1 was drawn to realize the top-level architecture of the future system.

The main idea was to separate the connection of the embedded hardware targets to a centralized system which could be used by the CI/CD service over an API. There was also a need within development teams for ability to manually connect to the target hardware for "debugging sessions", to utilize the same precious hardware by both developers and automation system. The device farm was planned to hold all the necessary information about the embedded hardware connected into it. The device farm would also need to have a connection into the artifact storage to for example fetch firmware images.

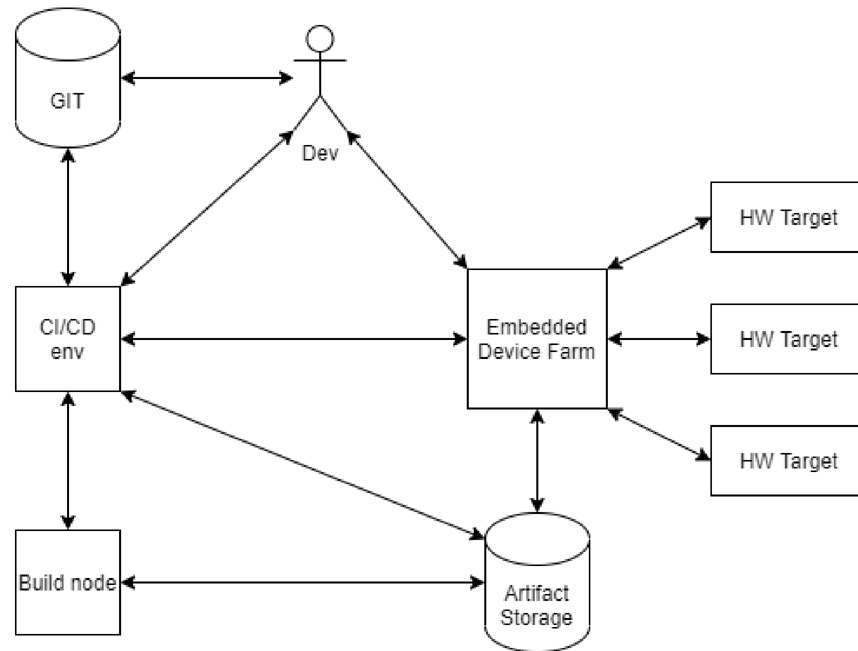

Figure 1. Top-level architecture of embedded device farm.

During more detailed requirement definition for the system, it was decided that the device farm should preferably be implemented using manager – worker architecture. Manager was planned to communicate with the existing CI/CD service as well as with worker instances. Worker instances was planned to be able to be connected to the manager instance from everywhere within the organization's network, but commonly from device laboratories. Hardware targets along with possibly needed peripherals was planned to be connected directly into worker nodes. Each worker could have a connection to multiple targets parallelly. Once LAVA was selected to be used as a platform for the device farm, it was clear that above mentioned architecture would be perfect choice for the system. A more detailed description of embedded device farm system architecture is presented in Figure 2.

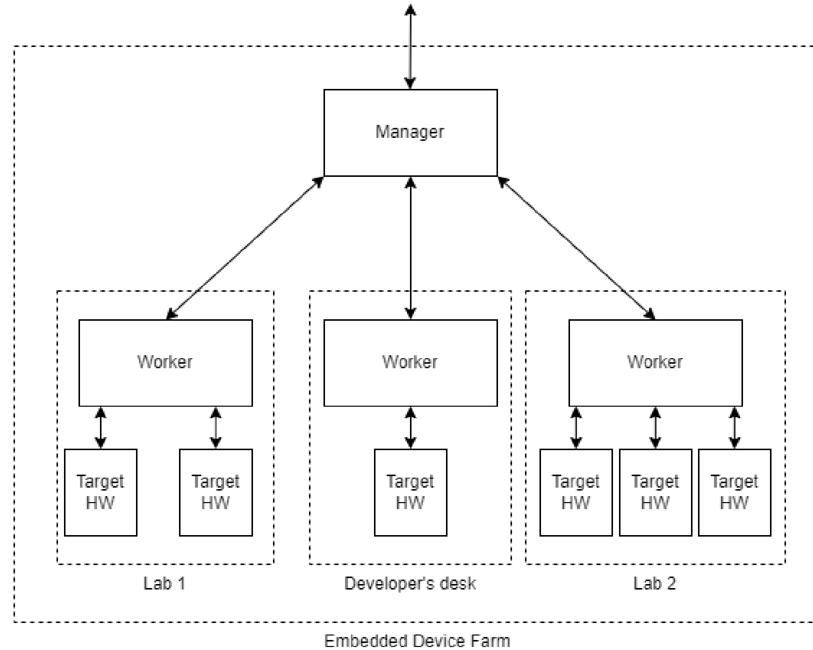

Figure 2. Embedded device farm system architecture.

LAVA implements an API which was planned to be used for communication between manager instance and the existing CI/CD service. In the PoC phase, a single manager instance would be implemented, but in the later phases duplication of manager instance could be considered to achieve better fail-safety. Containerization of manager instance would also be possible to achieve better scalability and reproducibility. Through the manager instance, connection to all the targets connected to the farm would be possible. This would comply with the requirement of scalability into an enterprise-wide target hardware pool in which same hardware could be used across different departments in the organization.

Target hardware connected to the system would be labelled uniquely and the verifications would flow from the CI/CD service to the target hardware through manager and worker according to the device labels. Multiple targets could also have the same label in case they are identical. On the test definition in CI/CD service, it would be defined on which label a verification would be run on. A typical scenario of verification sequence using embedded device farm is described in Figure 3. Firmware image is built and unit tests for it is run in the CI/CD service's containerized executors. If those steps are successful, image is archived as an artifact. Next, verification continues running on embedded device farm where firmware image is fetched from artifact storage and flashed into appropriate target board. On the target side, integration tests are run, and results are handed back to the CI/CD service.

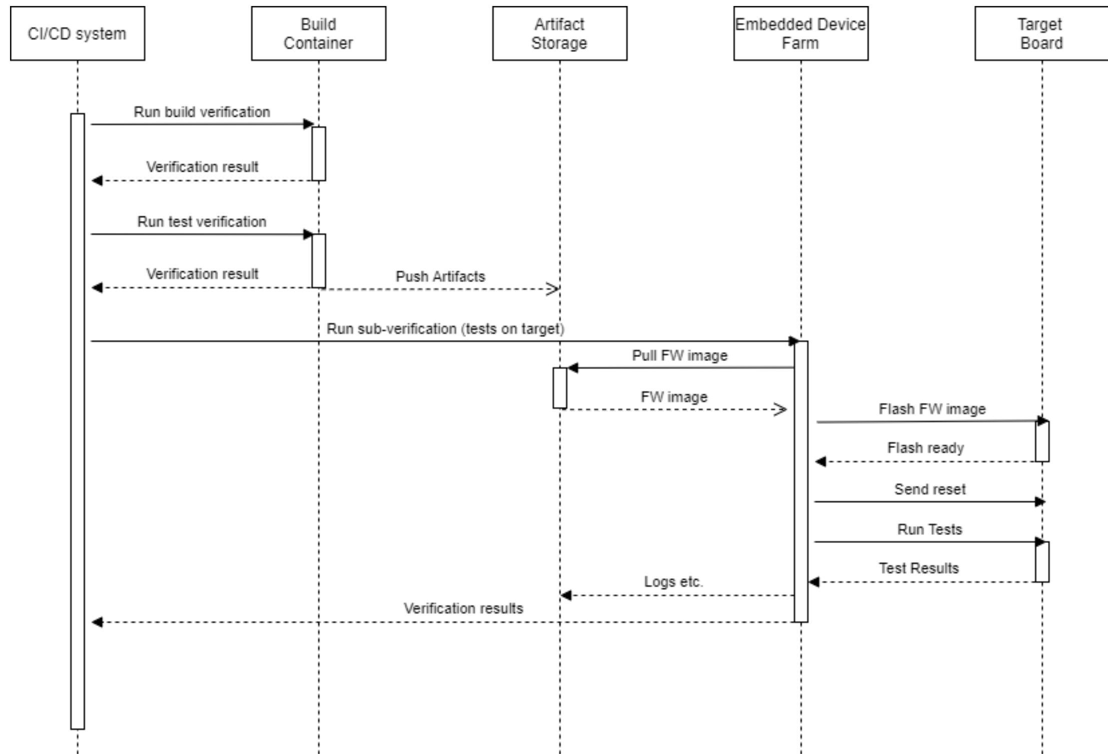

Figure 3. Typical verification sequence when using embedded device farm.

Embedded device farm was also planned to be aware of the details as well as the status of each target board connected into it. Details include for example architecture of the board, peripherals connected to it, and the commands used to interact with the board. That information is needed by automation to be able to successfully interact with the board. Status information on the other hand is needed to be able to schedule verifications to available targets. In LAVA, master instance implements a dedicated scheduler to fulfil this requirement. In LAVA, manual connections go through the same API which is used by CI/CD service to communicate with device farm. That makes the scheduler aware of the manual connection sessions also.

4.5 Implementation

Once the platform to be used to build the embedded device farm had been selected and the architecture for the system defined, the implementation phase could be started. This chapter describes the integration of LAVA system to be part of the existing CI/CD service. The section is divided into subsections to realize the chronological progress of the implementation.

4.5.1 Standalone phase

As always during familiarizing with a new technology, it is desirable to start from small implementation incrementally adding complexity to soften the learning curve. This “start small” mindset is also emphasized on LAVA documentation. To execute this in practice, the first goal was to bring up the selected device farm platform as a standalone solution. Even though LAVA implements a scalable master – worker architecture, it can be completely run on a single PC. This approach was used during this initial phase of implementation and a dedicated PC was acquired for the purpose. Debian 11 was selected to be used as OS in the PC because it is recommended in LAVA documentation. LAVA also offers official Docker images for both master and worker to detach from OS limitations. As this stage however, native installation was selected for simplicity.

First, LAVA server was installed and configured appropriately. LAVA server is performing in manager role in the architecture. LAVA master consists of web interface, database, scheduler and so-called lava-server-gunicorn daemon. Web interface is implemented using Apache web server, uWSGI application server, and Django web framework. Web interface is used to interact with LAVA system via UI or API. Database is implemented using PostgreSQL. Scheduler is implementing the functionality of LAVA. It is looking for changes in the database, checking the status of the connected targets, and executing the jobs on free targets. lava-server-gunicorn daemon on the other hand is the part of the application which handles the communication to worker nodes.

Setting up the server itself was fairly straightforward job, but some additional configuration was needed to expose the server to the internal organization network. For user authentication, LAVA supports multiple options: Lightweight Directory Access Protocol (LDAP), django-allauth, and local accounts. LDAP is commonly used protocol amongst the industry to enable authentication. django-allauth is a Django specific system used to authenticate with third party accounts including for example Google and GitLab. In this PoC, local accounts were used because no need for more advanced authentication in this phase was not identified. LDAP was planned to be assessed in further development of the system.

Once the master was up and running, a single worker was configured on the same machine. Worker consists of lava-worker daemon, dispatcher, and device under test (DUT). Worker daemon is handling the communication with master. Dispatcher is communicating with DUT and operating the jobs queried by master. Even though DUT is considered to be part of worker, it can also be seen as a separate block. Figure 4 represents the overview of LAVA system architecture. In LAVA documentation, the term DUT is used to refer to target hardware board. Worker was connected to the master instance by defining the master server uniform resource locator (URL) for the worker.

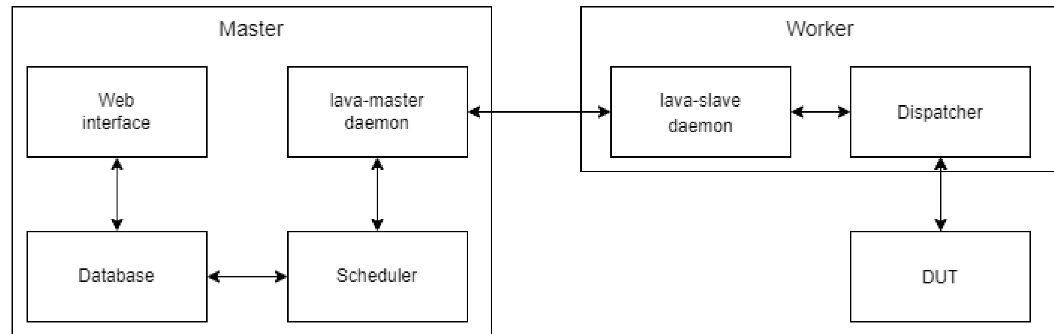

Figure 4. LAVA architecture overview.

LAVA comes with a set of predefined known device types. In practice this means that certain board types are supported by default. However, it is also possible to add completely new device types to LAVA, although it is not always a quick job. Bunch of precise configuration information about the board is needed, for example boot addresses, boot loader type, and console device. Sometimes changes to the dispatcher source code is also needed when adding new device type, which might make the integration more complicated. During this PoC, a known device type was selected to be used to keep the project in the scope of MVP. Raspberry Pi 3 Model B+ single board computer was selected to be used in the PoC due to high availability, and existing support by LAVA. Target was connected into the same network with worker host machine and SSH protocol was used for the communication between the worker machine and the target board.

Even though there was a built-in support for Raspberry in LAVA, some device specific configuration was needed to be applied. In LAVA system, configuration of a device is usually separated into three layers. The most general level of configuration is called device type, it includes general information of the board, such as boot addresses, boot loader details, and processor architecture. Secondly, for each individual device, there is a configuration file called device dictionary. This file holds more specific details of the particular board such as commands used to connect to the device and to power cycle the device. Both device type and device dictionary are defined using a jinja2 template format. Third phase of configuration defines more universal details of the device such as name used to identify the device and the worker which the device is connected into. There is not a configuration file for this stage, but the details are defined using lava-server CLI and information is stored in the database. During this configuration, device-type and device dictionary files are also bind to the device.

Ability to remotely switch power supply of target device is often needed during test execution on an embedded hardware target. In this PoC, Phidget Interface Kit 0/0/4, an USB controlled relay board, was used to control the power supply of the Raspberry device from the host machine. A small python helper script was written to simplify the usage of relays into single Linux terminal commands. The methods to turn power on and off as well as to power cycle the device were implemented into the script. Appropriate commands were then configured to LAVA device dictionary file. LAVA supports

basically any type of remote power switching device, as long as it can be controlled from host machine's Linux terminal.

For each device type in LAVA, a special test job called health check needs to be implemented. The purpose of the health check is to ensure that every board connected to the system is kept in a working order, and it is ensured by automatically running a minimal set of tasks on all the targets on regular basis. If the system detects a device is not able to pass the health check, it will be put into offline state to prevent actual test jobs to end up into possibly malfunctioning target. Health checks are defined in YAML format, and the schema is similar to the actual LAVA test definitions. In this PoC, a health check including boot and smoke testing was implemented for the Raspberry Pi target. The health check ensures that the device can be booted remotely and that a minimal set of Linux terminal commands can be executed on the target. The health check was set to run periodically every 24 hours.

As mentioned, in addition to dedicated API, LAVA also implements a web interface. At this stage of the implementation, the interface was available within the organization network. Web interface view showing recent jobs is presented in Figure 5. Web interface has features to operate and administer LAVA instance. It can be used to examine, monitor, and edit the devices connected to the system. Test jobs can be executed and tracked through the web interface. In addition, it has the documentation for the API, which can similarly be used to interface with the LAVA instance. In this implementation, the API will be used rather than the graphical web interface, because LAVA will mainly be used by automation systems, not human beings. LAVA project has also implemented a CLI application called lava-cli which can be used to interact with LAVA instance. lava-cli is a python-based utility which interacts with LAVA over the same API that the web interface uses.

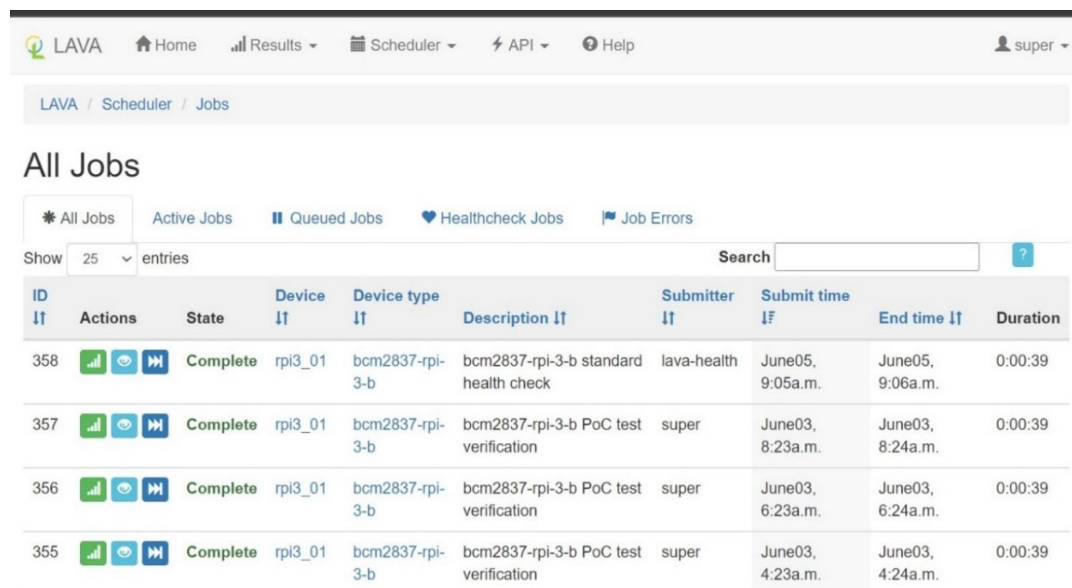

Figure 5. LAVA web interface jobs-view.

A requirement for the system was to enable configuration for the system as code. As mentioned, the device specific configuration in LAVA is maintained using jinja2 format. The health checks as well as the actual test definitions are stored in YAML files. At this stage, a dedicated git repository was created to hold all the necessary configuration files.

4.5.2 Connecting to the existing system

Once LAVA had been brought up as standalone solution with an embedded target connected into it, it was time to connect it into the existing system to be as part of CI/CD loop and to enable the actual usage scenario. Three different design options were considered to be used on this stage of the implementation. First option was to communicate with LAVA directly from existing build containers. This approach would have been very simple, but too much valuable resources would have been reserved just to keep up connection between systems during testing which is sometimes very time consuming. Another approach was to integrate separate but minimal and scalable container executors which would handle the communication between the existing system and LAVA manager via LAVA API. This would also have been a simple option, but still unnecessary computing resources would be reserved just to keep up the connection.

Third option and eventual resolution was to connect the LAVA master itself to the existing environment as executor having capability to parallelly run multiple verifications. The connection is based on SSH protocol and is established automatically by the CI/CD service. As a benefit compared to earlier two options, no additional components need to be added into the system because the master needs to exist anyway. The same manager instance can also be connected to different environments to share the same device farm across different projects. This is a very simple yet effective solution. Also, according to requirement definition, the manager was planned to be containerized on further development, which would add additional scalability for the system. Figure 6 presents the overview of the system defining also the connection methods used on different levels.

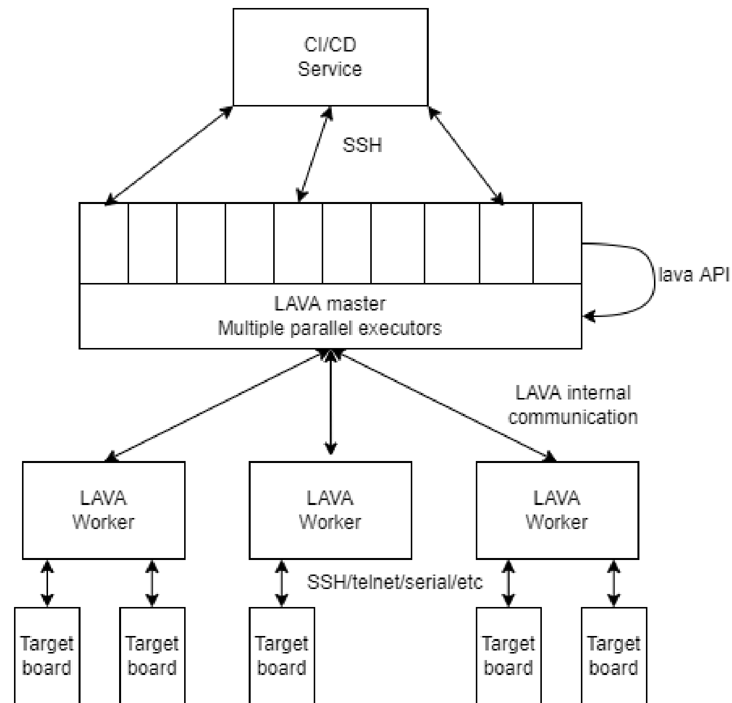

Figure 6. System connection overview.

On practical level, the test jobs are passed from the CI/CD service to device farm via LAVA API using lava-cli application in the verification steps. The target testing related verification steps are assigned to the device farm executor which in this case was the LAVA master. lava-cli has an interactive CLI, which makes it possible to submit jobs to LAVA based on YAML defined test configurations and follow the logging in real time from the command line. Using lava-cli was a natural approach to enable the interaction between the existing system and LAVA. It is simple yet reliable solution to use the API to easily submit and follow the test jobs in LAVA. However, lava-cli commands do not return any exit codes, so additional mechanism was implemented to determine the target test result on the existing CI/CD service. A small helper script was enough to do this job. At this stage, lava-cli was used with general manually defined user identity. On production system, a dedicated user identity for each project could be created to differentiate the jobs from different projects and possibly to restrict access to certain targets.

The existing CI/CD service comes with a CaC interface for defining the verifications. This PoC approach did require no modifications to that interface, although, on further development, special steps for the device farm verifications could be implemented to harmonize the user experience. At this stage, a demonstration verification containing basic smoke testing on target device was defined into CI/CD service. In addition to standard LAVA smoke testing, a separate test based on simple shell script was implemented to better demonstrate the actual use case.

The verification flow across different executors was defined using device-specific labels. First, on the verification definition, executor label was defined to make the

verification to be executed in the device farm executor. Secondly, on the test definition, target device type or label was defined which enables the device farm to select correct target device for the test. The verification configuration and the test definition files were stored into the same repository which had been created earlier during this implementation. In the production environment, different projects could have their test definitions on their own repositories, which would make it easier to maintain the system in large scale. Also, target testing logs were configured to be stored as artifacts via the existing system to demonstrate the actual use case.

Even though only simple testing was implemented at this stage, LAVA supports diverse testing methods. In practice, LAVA test definition YAMLs consist of three actions: deploy, boot, and test. LAVA has clear reference documentation for these three actions. On the deploy action, it can be selected what kind of operating system will be deployed into the target. Everything from device tree blob to root filesystem can be separately defined to be downloaded from online source. Also, deployment to network bootable drive is possible. The boot action defines how the device is booted during testing. For example, boot loader type and the needed commands can be selected. Login commands and the expected prompts can be defined. The actual tests are defined in the test action. Tests can be defined in the YAML itself, or in separate git repositories which can be imported automatically into target. Tests itself can be defined using basically anything which can be run from Linux terminal, anything from simple shell scripts to more comprehensive testing systems. Also, interactive testing is supported by default.

4.5.3 Manual connection method

During initial requirement definition, a need for ability to manually connect to target hardware connected to the embedded device farm was recognized. The main goal of manual connection method is to let the developers utilize the same target hardware with the CI/CD service to for example perform debugging or manual testing. This would be beneficial for example by saving on hardware acquisition or by minimizing the configuration gap between “developer’s desk” and the automation system.

As mentioned on description chapter of LAVA, it implements a solution to above mentioned scenario also. In LAVA, manual connection method is called hacking session. Hacking session is initialized using a special type of test job, which sets up a SSH daemon on the target board for the developer to access the target. Developer’s SSH public key is handed as a parameter for the hacking session job. The job queues similarly as regular test jobs, which allows the hacking session to interact with the same scheduler as the CI/CD service does avoiding hacking sessions not to disturb any running jobs on the targets. Once the special job has been executed on the target, a SSH access is open for predefined public key owner. The session will be available for predefined period of time or until user manually ends it.

Even though LAVA already implements ability for manual connections, some adjustments for it were needed for this PoC. On the initial testing of the hacking session,

it was discovered that if SSH is used as primary connection method for the target and if new firmware image is not deployed on each boot, using hacking sessions will corrupt the existing connection configuration. In general, LAVA expects that a fresh OS is always deployed to the target for each new test, which is not the case in this PoC. However, with some modifications to the hacking session start, initialization, and stop scripts, the connection was able to be established and the existing connection was not disturbed.

To harmonize the user experience of the system, a dedicated manual verification to the existing CI/CD service was defined which initializes the device farm manual connection. Basically, the verification queries user's SSH public key and the target board type as parameters and then communicates with LAVA via lava-cli to set up the hacking session for selected target type. Once the manual connection is available, a command to connect to the target is printed out in the verification log by lava-cli. Figure 7 presents the flow of setting up the manual connection method for selected target device. This approach has couple of benefits. First, developers do not need to familiarize with new system and its UI. This also eliminates the need to enable user identification to LAVA for all the users because they do not need to interact with it directly.

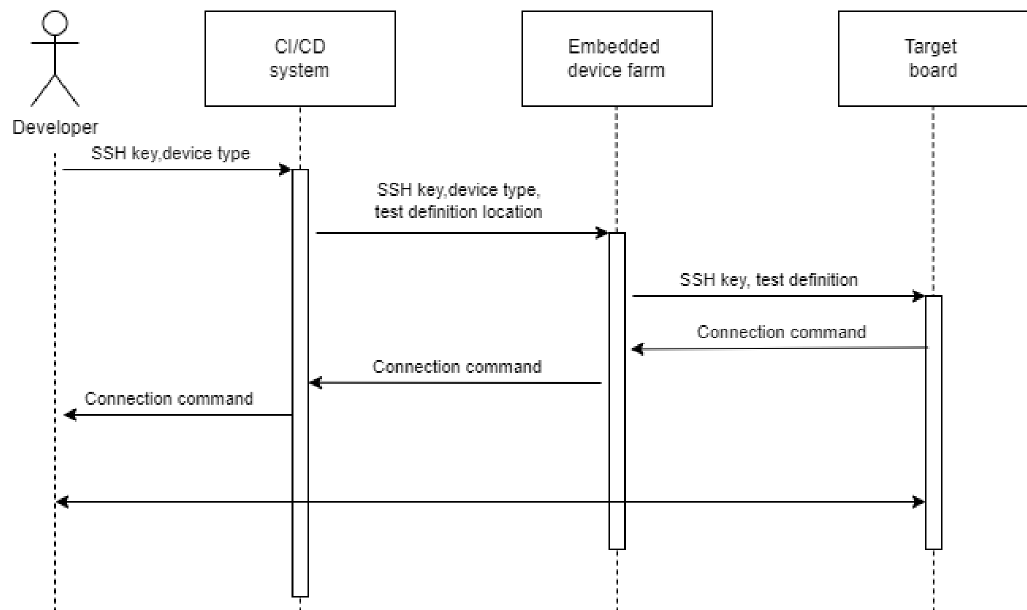

Figure 7. Manual connection method sequence.

4.5.4 Testing

To verify that the PoC functions properly and meets the requirements scoped for it, a set of functional tests were defined and conducted. Appendix 2 presents the tests defined for the system in table format. Each test case is identified with unique identifier in format Tx.y, x presenting the requirement ID from Appendix 1 which the test is related to and y

being just a running number. For each test case, specific description was defined explaining what is being tested, why it is done, and how the test is conducted. In separate column, expected outcome of the test was defined to be able to determine whether the test was successful. Each test case was conducted for the system, and lastly, result of each test case was presented in the table using simple pass or fail judgment.

The test cases cover the whole functionality of the system from connections between different blocks to scheduler operation and manual connection. All the functional features of the system were tested as well as different kind of usage scenarios. To mention some individual test cases, T2.5 tests the long-term stability of the system and proves that the system can be run successfully over longer period of time. On T6.1 and T6.2, remote power switching of the device was tested to verify that the target can be powered on and off during testing successfully. T9.1, T9.2, and T9.3 are verifying the operation of manual connection method.

Not all of the system requirements were such that functional tests for them could have been conducted, which is the reason why there are gaps in the identification numbering of the test cases. Storing configuration files into version control system or documenting the system are examples of non-functional requirements. Also, some of the requirements were not in the scope for the PoC phase so they were left out from testing, for example RQ7.

4.5.5 Documentation

Importance of proper documentation was highlighted already on requirement definition phase. Requirements for documentation included initial user guide, architectural description, and design description. User guide was written for the future users of the system to understand the functionality and needed configuration steps. User guide describes the procedure to create a verification which utilizes the embedded device farm to the CI/CD service and interpret the results. Accordingly, manual connection method was also presented and usage of it instructed in the user guide. Some additional usage tips were also listed in the user guide.

Architectural and design descriptions were combined into single document describing the system design verbally along with descriptive drawings. Separate setup guide containing precise steps to setup LAVA system was also produced to support the future setup of the system. Setup steps needed for both worker and master instance were introduced. In addition to all the written documentation, a presentation on the system was given for the CI/CD team to share information about conducted PoC and future plans for the system.

5. DISCUSSION AND FUTURE WORK

The goal of this thesis was to implement a PoC solution of a unified embedded device farm system to streamline the usage of embedded hardware targets as part of existing CI/CD pipeline. A system called LAVA was selected to be used in the implementation. This chapter discusses the implementation in terms of achievement of the goals, challenges and risks faced, and limitations and future development of the system.

All the requirements set for the PoC phase were fulfilled during the implementation. In addition, some of the scoped-out requirements were also implemented. The system was brought up and it was working as expected.

During the implementation, there were some challenges also. Initially LAVA documentation seemed to be very comprehensive, but during the implementation it was found out to be little bit inadequate, scattered, and obsolescent on some parts. There were parts in which documentation was lacking important information or the information was obsolete, which made the initial set up harder and consumed a lot of time during setup phase. This is probably also the reason why LAVA project is writing completely new version of the documentation, but unfortunately it is not yet ready. Another minor challenge was with setting up manual connection method. SSH is not designed to be the primary connection method in LAVA system which caused some challenges on the setup. Also, LAVA expects the firmware image of the device to be always freshly deployed before testing which was not the case in the PoC. Overall, being an open-source system, not all the corner cases have been taken into account in LAVA design.

There were some risks identified during the implementation. Some of the risks were concerning the PoC phase itself, while most of them were about implementing the system in larger scale in the future. The very first identified risk was the possibility for too large scope for the project. In the beginning of the thesis, it was quickly realized that the system could get very complex. However, the risk was mitigated by focusing strictly on producing an MVP solution only. Future risk in general is the addition of additional block between the existing solution and device targets. Computer systems always need maintenance and knowledge from the personnel in responsible of the systems. On the other hand, the device farm replaces some machines and configurations which has usually been manually maintained with an automated system setup, which should reduce the maintenance related problems and overhead.

Some future risks are also specific for using LAVA as a base for an embedded device farm. For example, LAVA is very much concentrated on kernel testing, which is well visible on the system design. Therefore, it might get complicated to implement an enterprise level embedded target device farm using LAVA. Also, if project happens to have custom hardware targets as part of the system it might be time consuming to integrate them as part of the system.

Being a PoC solution means there are limitations on the system which have been acknowledged. Even though the system was successfully brought up in sandboxed environment as MVP solution, it is evident that it could not directly be turned into

generalized solution. The scope of the PoC was intentionally limited to evaluate if it would be possible to implement an embedded device farm for the organization in first place. In order to further evaluate the suitability of LAVA system as base for the device farm, further implementation on a production project would be needed.

Being a PoC also means that there was a lot of work left for the future. While defining the requirements for this PoC, further development was already kept in mind. Basically, all the necessary future work for the future development is listed on the requirement table in Appendix 1. To mention some of the bigger topics left for further development, automatic deployment of the system, ability to add additional peripherals, and scaling the system were scoped out in PoC phase. LAVA already covers most of the things left for the future, so mainly it would only be about integration of the features of it into the system, but there would also be some other tools needed. For example, to automatically deploy LAVA system, some additional tooling would be needed. Containerization of the master and possibly also worker instances would make it easier to automatically deploy the system itself. For automatic deployment and configuration management across the system, some automated framework such as Puppet [40] or Ansible [41] could be used.

Taking all aspects into considered, it would be beneficial to similarly evaluate some other system represented in section 4.3 for the purpose. This way the pros and cons of both LAVA and the other system would be more clearly visible. labgrid for example was initially evaluated almost as potential system for the job as LAVA. Taking into account the challenges discussed in this chapter, labgrid might not have similar problems. For example, it is not concentrated on Linux kernel testing, rather introducing only platform to connect embedded hardware into CI/CD loop.

All in all, this thesis proves that it is no wonder why target hardware is commonly seen as a challenge factor hindering implementation of continuous practices in embedded software domain. Environments and the targets vary a lot between organizations, and even within the organizations between different projects. This is due to the fact that it is challenging to implement a flexible yet complete and unified system for this purpose.

6. CONCLUSION

The term continuous practices has gathered a lot of attention during last two decades in the field of software engineering. Continuous practices refers to various technologies used to continuously integrate, deliver, and deploy software changes. This thesis presented the practices as well as the benefits of implementing them and the common challenges on the way especially in the scope of an embedded systems domain.

One common challenge on the way of implementing continuous practices in embedded systems domain being the connection of embedded target hardware as part of CI/CD loop was addressed in this thesis. The main outcome of the thesis was an embedded device farm PoC system which makes it easier to connect embedded hardware targets to an already established CI/CD service in a unified manner. Requirements for such system were defined, different platforms were reviewed to be used in the implementation, and a platform called Linaro Automation and Validation Architecture (LAVA) was selected to be used. Architecture and system design for the system were defined, and LAVA was successfully added as a part of the existing CI/CD loop.

Raspberry Pi embedded target board was connected to the embedded device farm and verifications using the target were defined using existing CI/CD service. Also, manual connection method was added to enable parallel usage of same hardware by both individual developers and the CI/CD service.

The system was evaluated, and the conclusion was that the PoC implemented with LAVA could possibly be extended into a production environment. However, LAVA was found out to be designed very kernel testing centric and some other challenge factors were also identified. Therefore, before turning this PoC implemented with LAVA into a production environment, this thesis suggested to similarly evaluate some other system discovered in this thesis for the purpose. In any case, as the scope for this thesis was to create a PoC system, a lot of work was left for future development.

7. REFERENCES

[1] Beck K., Beedle M., Van Bennekum A., Cockburn A., Cunningham W., Fowler M., Grenning J., Highsmith J., Hunt A., Jeffries R., Kern J., Marick B., Martin R. C., Mellor S., Schwaber K., Sutherland J. & Thomas D. (2001). Manifesto for agile software development.

[2] Stahl D., Martensson T. & Bosch J. (2017). Continuous practices and devops: beyond the buzz, what does it all mean? In 2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA) (pp. 440-448). IEEE. DOI: 10.1109/SEAA.2017.8114695.

[3] Fowler M. (2006). Continuous integration. URL: https://martinfowler.com/articles/continuousIntegration.html. Accessed 31.5.2022.

[4] Lwakatare L. E., Karvonen T., Sauvola T., Kuvaja P., Olsson H. H., Bosch J. & Oivo, M. (2016). Towards DevOps in the embedded systems domain: Why is it so hard? In 2016 49th Hawaii international conference on system sciences (HICSS) (pp. 5437-5446). IEEE. DOI: 10.1109/HICSS.2016.671.

[5] Vassallo C., Zampetti F., Romano D., Beller M., Panichella A., Di Penta M., & Zaidman A. (2016). Continuous delivery practices in a large financial organization. In 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 519-528). IEEE. DOI: https://doi.org/10.1109/ICSME.2016.72

[6] Dakkak A., Issa Mattos D. & Bosch J. (2021). Success Factors when Transitioning to Continuous Deployment in Software-Intensive Embedded Systems. In 2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA) (pp. 1-9). IEEE. DOI: 10.1109/SEAA53835.2021.00025.

[7] Parnin C., Helms E., Atlee C., Boughton H., Ghattas M., Glover A., Holman J., Micco J., Murphy B., Savor T., Stumm M., Whitaker S., & Williams L. (2017). The top 10 adages in continuous deployment. IEEE Software, 34(3), 86-95. DOI: 10.1109/MS.2017.86.

[8] Yasar H. (2021). Expanding DevSecOps to Embedded Systems; Is it possible? Carnegie-Mellon Univ. URL: https://apps.dtic.mil/sti/pdfs/AD1122795.pdf. Accessed 5.6.2022.

[9] Chen L. (2015). Continuous delivery: Huge benefits, but challenges too. IEEE software, 32(2), 50-54. DOI: 10.1109/MS.2015.27.

[10] Hilton M., Tunnell T., Huang K., Marinov D. & Dig, D. (2016). Usage, costs, and benefits of continuous integration in open-source projects. In 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 426-437). IEEE.

[11] Itkonen J., Udd R., Lassenius C. & Lehtonen T. (2016). Perceived Benefits of Adopting Continuous Delivery Practices. In ESEM (pp. 42-1). DOI: https://doi.org/10.1145/2961111.2962627.

[12] Rodríguez P., Haghighatkhah A., Lwakatare L. E., Teppola S., Suomalainen T., Eskeli J., Karvonen T., Kuvaja P., Verner J. M. & Oivo, M. (2017). Continuous deployment of software intensive products and services: A systematic mapping study. Journal of Systems and Software, 123, 263-291. DOI: https://doi.org/10.1016/j.jss.2015.12.015.

[13] Gupta R. K., Venkatachalapathy M. & Jeberla F. K. (2019). Challenges in adopting continuous delivery and devops in a globally distributed product team: a case study of a healthcare organization. In 2019 ACM/IEEE 14th International Conference on Global Software Engineering (ICGSE) (pp. 30-34). IEEE. DOI: 10.1109/ICGSE.2019.00020.

[14] Laukkanen E., Itkonen J. & Lassenius C. (2017). Problems, causes and solutions when adopting continuous delivery—A systematic literature review. Information and Software Technology, 82, 55-79. DOI: https://doi.org/10.1016/j.infsof.2016.10.001.

[15] Shahin M., Babar M. A. & Zhu, L. (2017). Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. IEEE Access, 5, 3909-3943. DOI: 10.1109/ACCESS.2017.2685629.

[16] Benmoshe I. (2014). How to Calculate the ROI of Continuous Delivery. Zend Corporation.

[17] Henzinger T. A., & Sifakis J. (2006). The embedded systems design challenge. In International Symposium on Formal Methods (pp. 1-15). Springer, Berlin, Heidelberg. DOI: https://doi.org/10.1007/11813040_1.

[18] Barr M. & Massa A. (2006). Programming embedded systems: with C and GNU development tools. O'Reilly Media, Inc.

[19] Ernst R. (1998). Codesign of embedded systems: Status and trends. IEEE Design & Test of Computers, 15(2), 45-54. DOI: 10.1109/54.679207.

[20] Engblom J., Girard G. & Werner B. (2006). Testing Embedded Software using Simulated Hardware. In Conference ERTS'06.

[21] Trappe W., Howard R. & Moore R. S. (2015). Low-energy security: Limits and opportunities in the internet of things. IEEE Security & Privacy, 13(1), 14-21. DOI: 10.1109/MSP.2015.7.

[22] Olsson H. H. & Bosch J. (2020). Going digital: Disruption and transformation in software-intensive embedded systems ecosystems. Journal of Software: Evolution and Process, 32(6), e2249. DOI: https://doi.org/10.1002/smr.2249.

[23] Northwood C. (2018). The Full Stack Developer: Your Essential Guide to the Everyday Skills Expected of a Modern Full Stack Web Developer. Apress. DOI: https://doi.org/10.1007/978-1-4842-4152-3.

[24] Hoda R., Salleh N. & Grundy J. (2018). The rise and evolution of agile software development. IEEE software, 35(5), 58-63. DOI: 10.1109/MS.2018.290111318.

[25] Baraniuk C. (2021). Why is there a chip shortage? URL: https://www.bbc.com/news/business-58230388. Accessed 2.4.2022.

[26] Rahman A., Partho A., Morrison P. & Williams L. (2018). What questions do programmers ask about configuration as code? In Proceedings of the 4th International Workshop on Rapid Continuous Software Engineering (pp. 16-22).

[27] Perforce Software Inc (2020). Configuration as code: how to streamline your pipeline. URL: https://www.perforce.com/blog/vcs/configuration-as-code. Accessed 31.3.2022.

[28] Embedded Linux test systems (2022). URL: https://elinux.org/Test_Systems. Accessed 31.3.2022.

[29] Linaro Limited (2022). LAVA. URL: https://www.lavasoftware.org/. Accessed 31.3.2022.

[30] LAVA WIP documentation (2022). URL: https://lava.readthedocs.io/en/latest/. Accessed 4.4.2022.

[31] Pengutronix, Luebbe J. & Czerwinski R. (2022). labgrid documentation. URL: https://labgrid.readthedocs.io/en/latest/. Accessed 31.3.2022.

[32] r4d - rack infrastructure control system (2022). URL: https://github.com/ci-rt/r4d/blob/master/README.rst. Accessed 2.4.2022.

[33] HMS Industrial Networks AB (2022). Hottest project README documentation. URL: https://github.com/hms-networks/hottest/blob/master/README.md. Accessed 2.4.2022.

[34] STAF Project (2022). Software testing automation framework (STAF) documentation. URL: http://staf.sourceforge.net/. Accessed 2.4.2022.

[35] Testlink project README (2022). URL: https://github.com/TestLinkOpenSourceTRMS/testlink-code/blob/testlink_1_9/README.md. Accessed 2.4.2022.

[36] Opentest documentation (2022). URL: http://arago-project.org/wiki/index.php/Opentest. Accessed 2.4.2022.

[37] Sony Corporation (2017). Fuego test system. URL: http://fuegotest.org/. Accessed 2.4.2022.

[38] Sangorrin D. (2019). How to integrate Fuego automated testing tool in your CI loop. URL: http://fuegotest.org/ffiles/How_to_integrate_Fuego_into_your_CI_loop-Toshiba-Daniel-Sangorrin.pdf. Accessed 2.4.2022.

[39] Timesys Corporation (2022). Embedded Board Farm. URL: https://timesys.com/solutions/embedded-board-farm/. Accessed 2.4.2022.

[40] Puppet (2022) Puppet. URL: https://puppet.com/. Accessed 17.5.2022.

[41] Red Hat, Inc (2022). Ansible. URL: https://www.ansible.com/. Accessed 17.5.2022.

8. APPENDICES

Appendix 1. Embedded device farm system requirements

Req ID	Short description	Additional information	Use case	Priority	In PoC scope
RQ1	Connection to target boards from device farm				
RQ1.1	Linux terminal connection to Linux based target boards from device farm	• Via SSH	Communicate with embedded Linux targets through device farm	High	Yes
RQ1.2	Serial connection connection to target boards from device farm		Communicate also with embedded non-Linux targets through device farm. Sometimes also used for logging.	Medium	No
RQ1.3	Connection with boot loader		Communication with boot loader is sometimes needed to for example select the firmware image the device is booted on or to modify other booting properties.	Low	No
RQ2	Device Farm is connected to existing CI/CD service setups		Use target hardware as part of existing CI/CD setup		
RQ2.1	Device Farm is connected to existing CI/CD service provider	• Specific labels to run verifications on device farm targets • Automated connection between farm and existing service	Enable verification execution from existing CI/CD setups on target boards connected to device farm	High	Yes
RQ2.2	Allow users to run any kind of verification on selected target board connected to farm		Device farm should not limit what testing system projects use, just to offer infra "on bottom"	Medium	Yes
RQ2.3	CI/CD service communicates with device farm scheduler to execute verifications on target boards	• This is valid in case scheduler is not implemented as part of the CI/CD service instead • See also RQ5	Schedule verifications from existing service to hardware targets	Medium	Possibly
RQ3	Configuration of device farm properties		Configure the properties of the device farm effortlessly		
RQ3.1	Configuration definition via YAML or similar format		Configuration as Code to more easily maintain consistency	High	Yes
RQ3.2	Configuration via dedicated git repository	• Possibly integrate to be part of existing configuration repository?	Maintain the configuration under a version control system to achieve traceability	Medium	Possibly
RQ3.3	Automatic deployment of Device Farm	• Verifications for the configuration repository to check and deploy configuration	Avoid manual deployment steps	Low	No
RQ4	Configuration of target board properties		Enable projects/developers to easily attach new targets to the farm.		
RQ4.1	Enable configuration of all the needed target board properties via device farm configs	• Device name • Other device properties such as device architecture, CPU type, boot actions • Labels to distinguish different kind of boards	Configure the farm to "know" the boards connected to it	High	Yes
RQ4.2	Configuration definition via YAML or similar format		Configuration as Code to more easily maintain consistency	High	Yes
RQ4.3	Configuration via dedicated git repository		Maintain the configuration under a version control system to achieve traceability	Medium	Possibly
RQ4.4	Automatic deployment of target hardware configuration	• Verifications for the configuration repository to check and deploy configuration	Avoid manual deployment steps.	Low	No

RQ5	Scheduler for target board resources				
RQ5.1	Scheduler is aware of every target board connected to the farm	• By device labels		High	Yes
RQ5.2	Scheduler is aware of the state of every target board connected to the farm	• Connected/not • Verification running on the board or not	Device farm needs to be aware on the state of the targets connected to it in order to be able to schedule verifications to targets without overlapping.	High	Yes
RQ5.3	Scheduler offers an interface for CI/CD service as well as for individual developers to "reserve" boards	• See RQ9 for specifics for manual connection	Allow both automation systems and developers to use same targets without overlapping.	Medium	Yes
RQ5.4	Scheduler can queue verifications if all boards with the specified label are reserved		Enable running the device farm in the production environment where verifications come and go.	Medium	Possibly
RQ6	Ability to connect other lab peripherals to the system		Communicate with needed lab peripherals from verification scripts.		
RQ6.1	Ability to connect remote controlled power supply for target board	• Enable usage of it via verification definitions	Power on/off target boards during verifications	Medium	Yes
RQ6.2	Ability to connect remote controlled reset switch for target board	• Enable usage of it via verification definitions	Reset target boards during verifications	Medium	No
RQ7	Support network booting target boards	• This might be more towards customer to implement for their verifications, but still good to take into account while designing the farm • This is related to RQ1.3	Booting target board using network located file system is sometimes necessary during testing for example		
RQ7.1	Ability to boot target board using network file system			Medium	No
RQ8	Possibility to single worker machine per lab setup		Get rid of multiple host machines running in single lab needing additional maintenance efforts.		
RQ8.1	Configuration of worker machine parameters via YAML or similar format	• Name of the lab • Peripherals connected • Boards connected	Configuration as Code to more easily maintain consistency	Medium	Possibly
RQ8.2	Possibility to connect all the needed targets and peripherals into one machine	• For example network switches, USB switches, debuggers, power switches, etc.	Peripherals are needed to communicate with target boards. Switches needed if there are multiple boards connected to single worker	Medium	No
RQ8.3	Containerized lab worker setup	• Container with necessary environment for worker nodes	Containerization makes it easier to setup needed environment for worker on any machine. However, it has downside of making it harder to communicate with physical devices.	Medium	No
RQ9	Manual connection to target boards		Enable developers to manually connect to target boards connected to the farm for debugging purposes for example. This will allow to use the same targets in both manual testing and in CI/CD loop.		
RQ9.1	Interface to manually connect to target boards connected to farm		Straight forward process for developer to manually access selected target.	Medium	Yes
RQ9.2	Ability to select communication method for manual connection	• Serial/SSH/etc	Necessary if there are multiple connection methods available for single board.	Low	No
RQ9.3	Manual connection is in sync with scheduler		Needed to avoid disturbing automation environment verifications and other manual sessions	Medium	Possibly

RQ10	Possibility to scalable distributed system setup		Enable scaling into enterprise wide system. Eventually, enable enterprise wide target board pool.		
RQ10.1	Separate "manager" instance(s)	• Possibly duplicated for fail-safety	Manage connections between worker nodes and CI/CD environment.	High	Yes
RQ10.2	Containerized manager instance(s)		Enable scalability and adaptability	Medium	Possibly
RQ10.3	Worker nodes on labs	• Target boards in the labs connected to workers • See also RQ8		High	Yes
RQ10.4	Possibility to connect a worker anywhere within organization network	• Worker communicates with the manager	Enable to setup "lab" anywhere inside organization network, also on developer's desk	Medium	No
RQ10.5	Unlimited ability to scale up worker nodes	Manager(s) needs to be able to handle dozens of workers	Enable connecting all the needed labs to the device farm	Low	No
RQ10.6	Unlimited ability to scale up amount of target boards	• Worker needs to be theoretically able to handle dozens of target boards to be connected to it • Physical factors such as USB and ethernet connections will however become a limiting factor • See also RQ8	Enable connecting all target boards in one lab to single worker machine	Low	No
RQ11	Documentation		Provide documentation for customers to set up the system as well as for developers to be able to understand the system design.		
RQ11.1	Initial user guide			High	Yes
RQ11.2	Architectural description	Descriptive drawings with additional verbal explanation		High	Yes
RQ11.3	Design description			High	Yes

Appendix 2. Embedded device farm functional tests

Test case	Test description	Expected outcome	PASS/ FAIL
T1.1	Connection between device farm and target board is tested by running lava-server command to start connection to board from LAVA	lava-server successfully establishes connection and device terminal access is gained	PASS
T2.1	Connection between CI/CD service and device farm system is tested by sending single lava-cli command to LAVA master from instance connected to CI/CD service	lava-cli command returns successful exit code	PASS
T2.2	Connection between CI/CD service and device farm system is tested by running a verification on device farm using on CI/CD service	The verification runs successfully on device farm master	PASS
T2.3	Labelized connection between CI/CD service and device farm system is tested by running a verification on device farm using a dedicated label on CI/CD service	The verification runs successfully on device farm master	PASS
T2.4	Automated connection of device farm master to CI/CD service is tested by restarting both services 10 times	Connection is automatically established after restarts	PASS
T2.5	Long term stability of the connection between CI/CD service and device farm system is tested by running periodic verification every second hour from CI/CD system on device farm connected target for one month	The verification runs successfully over the time period	PASS
T4.1	Target board property configuration is tested by setting valid board configuration via lava-cli	Configuration of the device is successfully changed	PASS
T4.2	Target board property configuration validation is tested by using lava-cli configuration checker with both valid and invalid configuration	lava-cli reports configuration validity correctly	PASS
T4.3	Long term stability of target board configuration is verified by setting up periodical LAVA health-check job to run once a day for one month	Health check runs successfully furing the time period	PASS
T5.1	Device farm scheduler is tested by submitting 10 subsequent test jobs from CI/CD service to the device farm target board	Verifications run successfully on systematic order on target	PASS
T5.2	Device farm scheduler is tested by submitting test job from CI/CD service to the device farm target board which is set offline	Verifications waits for the target to become online and fails after timeout if the target does not appear online	PASS
T5.3	Device farm scheduler labelization is tested by submitting 10 subsequent test jobs from CI/CD service to the device farm target board by target label	Verifications run successfully on systematic order on correct target	PASS
T6.1	Remote power switching of the target board is tested via lava-server utility	Power can be switched on and off using appropriate lava-server arguments	PASS
T6.2	Remote power switching of the target board is tested via verification defined in CI/CD service	Power can be switched on and off during verification using approprite LAVA definitions	PASS
T8.1	Worker configuration is tested by setting up valid configuration files and setting up worker via lava-server	Configuration of the worker is successfully changed	PASS
T8.2	Worker configuration validation is tested by using lava-server configuration checker with both valid and invalid configuration	lava-server reports configuration validity correctly	PASS
T9.1	Manual connection setup is tested by submitting manual connection job from CI/CD service to the device farm target board	Verification runs and prints out a command to connect to target	PASS
T9.2	Manual connection interfacing with scheduler is tested by submitting manual connection job from CI/CD service to the device farm target board while there is already a job running on target	Manual connection waits until target is available and then sets up manual connection	PASS
T9.3	Manual connection is tested by setupping manual connection from CI/CD service with SSH keys and using the provided command to connect to the device	Provided SSH key can be used to connect. Connection parameters are cleaned from target after connection is closed	PASS