Tampere University

Valtteri Yli-Länttä

DOCKER CONTAINER ENVIRONMENT FOR SOC SW DEVELOPMENT

Master Thesis
Faculty of Information Technology and Communication Sciences
Timo Hämäläinen
Esko Pekkarinen
March 2021

ABSTRACT

Valtteri Yli-Länttä: Docker container environment for SoC SW development
Master's Thesis
Tampere University
Master's Programme in Electrical Engineering
March 2021

Software projects' size has been growing significantly throughout the years and they require more initial conditions for developer team to be able to work on them independently. Containers meet these requirements and it is one of the reasons for their popularity. Other features provided by containers are ease of use, high performance, and portability. Developers have begun to use containers to pack tools and dependencies, which in turn speeds up the software projects' development and build environment setup.

The aim of this thesis work is to create a container-based development and build environment for system-on-chip software department, which is estimated to include 200 developers. The environment is used for multiple different products, where each individual product requires personalized software development kit and related dependencies. During the thesis work, two surveys were conducted for the system-on-chip software developers. The first survey focuses on the current state of the department's development environments and opinions on the new container environment. The second survey was held after the completion of the project, focusing on the success of the objectives and future development features in the environment. After the development environment is done, it is integrated into a continuous integration process, which provides automated product testing.

The department's current development environment and the results from the first survey revealed the need of this project work, and the fact that developers are willing to use the new container environment. During the implementation we pay special attention to create a solution with ease of use and comprehensive documentation. The main part of the implementation is an initialization script, which on behalf of the developer creates the container, moves the user to the container and sets up the environment. This way the developer only has to clone the repository and execute the script with desired parameters. The results of the second survey were positive and showed that the project work was successful. The reception was generally favourable and no major problems arose. In addition, an estimation on the saved work hours in the department by the implementation was made. It is estimated to save up to 1400 work hours on each product's setup, when every developer in the department has performed one environment setup.

Keywords: Docker, Containerization, System-on-Chip, Software, Continuous Integration

TIIVISTELMÄ

Valtteri Yli-Länttä: Docker-konttiympäristö järjestelmäpiirien ohjelmistokehitykseen
Diplomityö
Tampereen yliopisto
Sähkötekniikan DI-ohjelma
Maaliskuu 2021

Ohjelmistoprojektien kokonaisuus on kasvanut huomattavasti vuosien saatossa ja ne vaativat yhä enemmän alkuehtoja, jotta yksittäinen kehittäjätiimi voi työstää niitä. Konttiteknologia vastaa tähän tarpeeseen, joka on myös yksi syy sen yleistymiseen. Muita konttien piirteitä niiden menestymiseen ovat helppokäyttöisyys, korkea suorituskyky ja siirrettävyys. Ohjelmistokehittäjät käyttävät kontteja työkalujen ja riippuvuuksien pakkaamiseen, joka nopeuttaa ohjelmistoprojektien kehitys- ja käännösympäristön alustamista ja jakamista.

Tämän diplomityön tavoitteena on luoda konttipohjainen kehitys- ja käännösympäristö järjestelmäpiiri (engl. system-on-chip) ohjelmisto (engl. software) -osastolle, joka koostuu arviolta 200 kehittäjästä. Ympäristössä työstetään useita eri tuotteita, joista jokainen vaatii kohdennetun joukon työkaluja ja niistä koituvia riippuvuuksia. Kaksi kyselyä toteutettiin osaston kehittäjille diplomityön aikana. Ensimmäisessä kyselyssä tiedusteltiin osaston nykyisten kehitysympäristöjen tilannetta ja mielipiteitä uudesta konttipohjaisesta kehitysympäristöstä. Toinen kysely pidettiin ympäristön valmistumisen jälkeen, joka keskittyi ympäristön onnistumiseen asetetuissa tavoitteissa ja mahdollisiin jatkokehityskohteisiin ympäristössä. Kehitysympäristön ollessa valmis, se otetaan käyttöön jatkuvan integraation (engl. continuous integration) prosessissa, joka tarjoaa automatisoidun tuotetestauksen.

Osaston kehitysympäristön nykyinen tila ja ensimmäisen kyselyn tulokset osoittivat, että projektin toteutukselle on tarve ja kehittäjät ovat valmiita ottamaan uuden konttipohjaisen ympäristön käyttöön. Huomiota pyrittiin kiinnittämään toteutuksessa erityisesti helppokäyttöisyyteen ja kattavaan dokumentaatioon. Toteutuksen ydinosana on alustusskripti, joka automaattisesti luo käyttäjän puolesta kontin, siirtää käyttäjän konttiin ja asettaa ympäristön valmiiksi. Tällöin kehittäjän ei tarvitse tehdä muuta kuin hakea skripti arkistosta (engl. repository) ja suorittaa se haluamillaan parametreilla. Toisen kyselyn tulokset olivat positiivisia ja osoittivat projekti työn onnistuneen. Vastaanotto oli pääsääntöisesti suotuisa, eikä suurempia ongelmakohtia noussut esiin. Lopuksi tehtiin arvio toteutuksen työtuntien säästöstä osastolle. Arvion mukaan 1400 työtuntia säästyy jokaista tuotetta kohden, kun jokainen osaston kehittäjä on kerran alustanut uuden ympäristön.

Avainsanat: Docker, Konttiteknologia, Järjestelmäpiiri, Ohjelmisto, Jatkuva integraatio

PREFACE

I want to thank my colleagues, managers and supervisors for their guidance and support during the thesis work. Your time and help have been irreplaceable.

I would also like to express my gratitude to my family, friends and fiancée. You have always been able to cheer me up and encourage me to believe in myself. Thank you.

Tampere, 16. March 2021

Valtteri Yli-Länttä

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

CI	Continuous Integration
GCC	GNU Compiler Collection
GID	Group ID
HW	Hardware
IC	Integrated Circuit
IP	Intellectual Property
LINSEE	Linux Software Engineering Environment
NFS	Network File System
OS	Operating System
RMM	Reuse Methodology Manual
RTL	Register-Transfer Level
SDK	Software Development Kit
SoC	System-on-Chip
SW	Software
UID	User ID
VCS	Version Control System
VLSI	Very Large-Scale Integration
VM	Virtual Machine
VMM	Virtual Machine Monitor

.

1. INTRODUCTION

A build environment is a necessity when considering building any software (SW). The build environment can be thought of as a space which contains the necessary features and tools for development, such as specific software development kit (SDK), operating system (OS), repositories, and mounts. These requirements may vary more if continuous integration (CI) is implemented [1]. CI lightens the developer's manual workload as it provides automated steps for laborious tasks. The use of CI requires its own implementation and increases the number of requirements and tools. The build phase becomes challenging when it must be available on multiple different platforms. In SW department's use case, the platform is virtual machine (VM), laptop, cloud server or physical server. All these platforms might not have the same OS: common ones are Linux distributions, Windows and macOS. After initial setup, maintaining these environments becomes the next major challenge. As there are multiple different platforms, having up-to-date and comprehensive documentation tends to prove difficulty. In addition, updating the build platforms and making large scale adjustments can be time consuming at the later stages of the project.

The objective of this thesis is to implement a uniform environment for system-on-chip (SoC) SW development and CI. The target is to produce development environment with all the required tools and dependencies for SoC SW department's developers to work on specific projects. Docker containers were chosen for this task. Containerization is a method where software is running in isolation. The implementation must be usable by developers who have no previous experience with Docker. Docker is a commonly used free container management system which functions on all major platforms [2].

At the beginning of the thesis project, a survey is conducted. In the survey, questions regarding developers' current build environments are asked. The survey is intended to assist in the development process as it reveals current problem scenarios. In the practical part of this thesis, an initialization script is written to create a Docker image. The image holds all the required basic tools needed in the build environment and makes it effortless to update. After creating the image, the initiation script creates the container which the developer can now use as development platform. Once the container is implemented, it can be taken into use by the CI workflow, to automate product testing. The

thesis will also go through how to make a proper documentation. The thesis goes through some challenges of Docker, containers, and different setups. After the implementation a second survey is held where developers are requested to compare the new environment to the old ones. This provides information on how the project succeeded. Once the container environment is functioning correctly, it is integrated into a CI process. This provides the developers automated product testing, saving time from repetitive tasks.

Finding solutions to the problems with current work platforms allows SW developers to efficiently focus on actual work tasks instead of manually setting up the product dependencies. Updated and comprehensive documentation makes the usage of containers fast and easy. It will also ease newcomer's adaptation to new work tasks.

The thesis is structured in six chapters. Chapter 2 contains the theoretical background on key subjects. After that section, the reader should be familiar with the terminology and technology used in this thesis. In addition, related work is discussed and why Docker containers were chosen for the project. In chapter 3 the main problems and challenges are discussed and analysed, and what solutions are planned for them. First, the current situation of the SoC SW department is investigated and the data collected from the survey is analysed. Then the challenges faced during the container environment build phase are discussed. Chapter 4 goes through the design and implementation of the container build environment. Here the solutions to the previously mentioned challenges are gone through in detail and explained the decisions made along the way. In chapter 5 results are discussed. This includes the user feedback gathered via the second part of the survey. Chapter 6 concludes the thesis and discusses possible further development of the build environment.

2. SYSTEM-ON-CHIP SOFTWARE DEVELOPMENT

In this section the relevant technologies and methods of the thesis topic are introduced. At first the current state of SoC technology is examined in section 2.1. SoC technology is focusing more on the SW than the HW (hardware) aspect. In section 2.2 general SW development workflow is discussed. This includes build process, build environment, SoC SW development, build automation and reproducibility. CI in its own is considered in section 2.3.

In section 2.4 the focus is on Docker, containers, and virtualization. Here containers are introduced and how Docker can be used to manage them. In addition, comparison between containerization and virtualization is done to get a clear view of their pros and cons. Then in section 2.5 Yocto's development usage and its umbrella projects are discussed. Finally, in section 2.6, related work regarding containerizing build environment is analysed and similarities to this project are inspected.

2.1 System-on-chip

An SoC can be specified as an integrated circuit (IC) which contains multiple independent very large-scale integration (VLSI) designs which forms an operational application to the chip [3]. The predefined cores in SoC are integrated components, which usually include microprocessors, central processing unit (CPU), graphical processing unit (GPU), large memory arrays, audio and video controllers and so on. These cores are referred to as intellectual property (IP) blocks. Cores can be separated in two classes depending on their nature: soft cores are in form of synthesizable register-transfer level (RTL) description, and the hard cores have been optimized for performance and a certain process [3]. The pay-off between these solutions are flexibility, performance, time-to-market and portability among others. Cores with more design specific attributes tend to have better performance and shorter time-to-market, but they have less re-usability, flexibility and have higher cost [3].

Depending on the nature and scale of the SoC project, there might be third party IP blocks in use. In this case the customer usually offers few premade IP blocks for the deliverer company to build the SoC around, or the customer orders few IP blocks from the deliverer. Since all the IP blocks communicate with each other via communication

channels, it is critical for the SoC developers to have precise documentation of the IP blocks [4].

SoC design is becoming faster due to the reusability of IP blocks. The fact that SoC designers can also use third party IP blocks speeds up the time-to-market process. Currently widespread design flow method called the reuse methodology manual (RMM) is used in SoC industry [4]. The main concepts of RMM can be seen in Figure 1. The RMM design flow starts with overall system specification which includes non-functional physical features and functional aspects (step 1). From this specification the development continues to behavioural model (step 2) and is then refined and tested (step 3). Once the initial model is properly tested, manual HW and SW partition is performed to establish communication protocol between HW and SW IP blocks (step 4). The next two steps are performed in small cycles: a HW model and a SW prototype are developed (step 5) and then co-simulated to see if the design matches expectations (step 6). Once the simulations validate the design, last specifications for the HW and SW IPs are performed (step 7) [4].

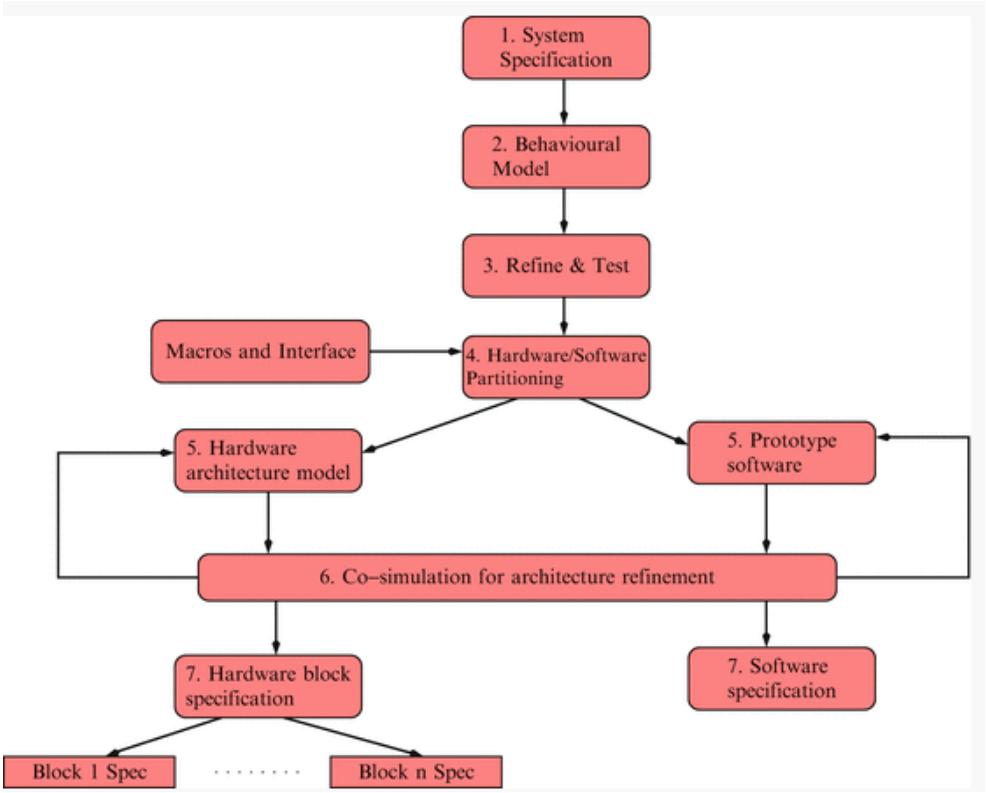

Figure 1. RMM SoC design flow [4]

The use of IP blocks takes mainly place in phases 4 and 5. Despite the fact that RMM has multiple phases and a single SoC can have large amount of individual IP blocks, the most time-consuming phase is verification with 50-80% of total design effort [4]. Generally, the verification phase consumes more time in safety-critical systems.

2.2 Software development

SW life cycle consists of requirements, analysis, design, implementation, testing and iterations [5]. There are multitude of tools and methods to help developers with these steps and to achieve desired goals.

In the build process of the SW life cycle, developers produce code, execute program, test program, and repeat these steps as many times as necessary. The result of this build process is the application, and its features are tracked with release number. The build process has variety in its steps depending on the used programming language and tool choices. These steps may include code compilation, software packaging, creation of databases, executable installer and more. Once the application has all required functionalities and it passes the test cases, it is ready for deployment.

For the developer to be able to run the SW build process, their build environment must meet certain requirements. These requirements involve a compiler, libraries, tools, and an OS. The correct choice of compiler is tied to the used programming language. For example, GNU Compiler Collection (GCC) includes front ends for C, C++, Objective-C, Fortran, Ada, Go and D, and on top of that contains libraries for the previously mentioned programming languages [6]. In addition of the default libraries, certain builds might use libraries available online or made in-house. The build environment also needs all the tools used in the build process. These tools might be used for SW compilation process, documentation, debugging and many more tasks. The developer has to take into account the tool versions, since different versions of the same tool might not work in the same way. This may cause unwanted results or even build crashes. Lastly, the same outcome is likely not achieved if the OS is switched from Windows to Linux distribution even if the same set of tools are available. For this reason, it is critical for the build environment to have correct OS.

Traditional SW development differs in some ways from SW development for embedded systems. The most significant difference is the real-time requirements. In embedded SW, it is common that the correctness of the application computing is dependent on result and timing. Real-time system interaction is implemented with the help of events and interrupts [7]. In addition, some differences with the traditional build process compared to

embedded build process can be seen in Figure 2. Traditional build process functions with the help of OS, whereas embedded process does not necessarily require it. In embedded SW the key steps in converting the source code into binary image are compiling using a compiler, linking with a linker and relocation by locator. The compiler processes the human-readable source code and translates it for the processor. In embedded SW build process, a cross compiler is used, as the product platform is almost without exception different from the compiler's platform. In the next step, a linker is used to tie together all object files produced by the compiler, to form a single relocatable program. After relocatable program is generated, a locator is used to determine physical memory addresses in the embedded process. The locator converts the relocatable program into a executable binary image, which then can be programmed into ROM or flash device. Lastly, the developers have to take into account the uniqueness of each hardware platform. Factors to consider are system initialization, processor interfaces, load distribution, resource allocation and real-time system timing requirements [7].

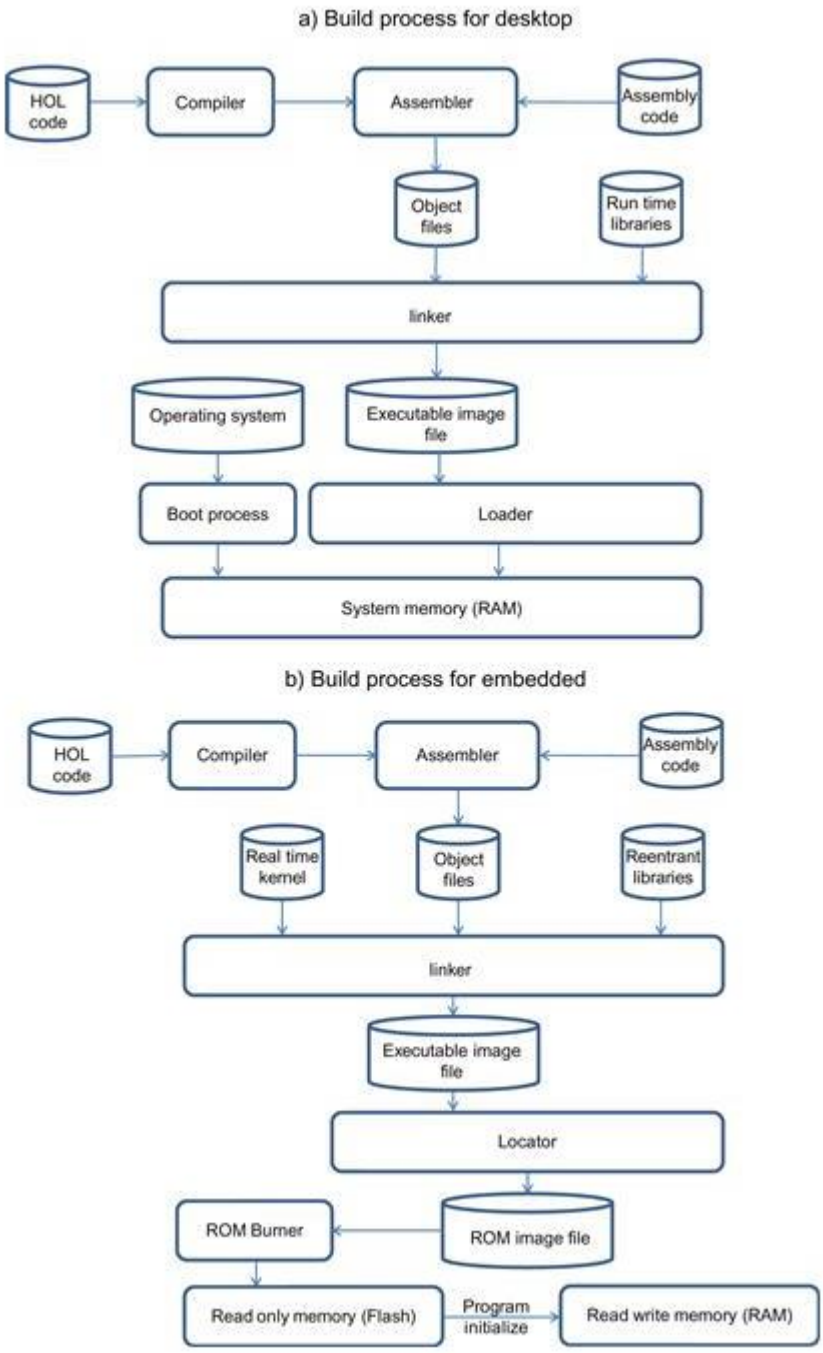

Figure 2. a) traditional SW build process and b) embedded SW build process [7]

Setting up the whole build environment manually can be a great deal of work. This task can be lighten, for example by using premade scripts. Scripts of this kind can be used for setting environment variables, correct tool versions, fetching SW source code and more. To be able to fetch source code, Version Control System (VCS) of some kind is required. VCS additionally provides access to every version of the stored projects, and eases management as well as coordination of projects. Collaborate tools are supported

by VCS, which is useful as larger development project entities require many developers. Currently very popular VCS tool is Git [8].

With a functional build environment developer can begin to build the SW. Even with the build environment done, the remaining steps can be tedious and time consuming. Manual deployment leaves room for human errors: steps can be done incorrectly or completely missed. Proper solution to avoid this kind of situation is build automation. Build automation holds within build integration, automated tests, code analysis and deployment [9]. With build automation the variation between builds is diminished, removing possibility of human error. Furthermore, using build automation speeds up the build process based on how many steps it has, since the time taken from the developer to execute individual build steps is removed.

Reproducibility is described as the ability to duplicate the results of a former study using the same materials [10]. In SW development, reproducibility guarantees the ability for a developer to reproduce certain steps and end in the same result in their own development environment. Efficient methods to ensure reproducibility in SW development are previously presented build automation and precise documentation. The documentation should cover at least environment factors, such as OS and tool versions, used libraries, setup methods and parameters used. Reproducibility is also important in case of bugs. When bug occurs to a client or other developer, detailed description of the malfunction is required for the developer to be able to reproduce and fix it.

Important part of build automation and a great tool to help reproducibility is test automation. Automated tests help to ensure the correct functionality of the program and prevent the developer from breaking the code-under-test. Any developer and the CI server should be able to run the entire automated test suite basically without any setup [9]. The duration of the test sequence should not be too lengthy since it will increase the SW build time and consume developers' active time.

2.3 Continuous integration

CI is a SW development method, where all developers commit their code changes to same branch instead of local branches, possibly multiple times a day. The main goal of CI is to remove unnecessary time spend on integration and speed up the debugging process. When multiple developers work simultaneously on same code, it is quite common for something to break. With smaller changes and automated test suite, debugging becomes much easier and malfunctioning code is less tedious to locate [11].

One of the key aspects of CI is the ability to keep the mainline branch free of defects. This means, every new code change has to result in a successful build. Malfunctioning mainline prevents developers from committing new changes, since they will have the same error [11]. Even with the help of automated tests, locating a defect might take some time. A commonly used method to prevent defects from getting into the mainline is to run a private build. By running a private build, the developer will get latest available version from version control repository and integrate new changes to it. After applying changes, a build with all the unit tests must be ran. Testing changes proactively reduces the number of defects in the mainline and therefore reduces the time spend on fixing malfunctions [12].

CI efficiency can be improved with the help of CI tools, and the first important step by developers is to choose the tool which fits their needs. Some of the important qualities are code quality analysis, build acceleration, security, and code coverage [12]. Figure 3 represents a typical CI flow from a developer's perspective. First, the developer makes changes to the latest code version and runs a private build. After passing unit tests locally developer can commit changes to version control repository. Code commit triggers CI server to automatically fetch and build the SW with the latest changes. CI server then forwards the build result to the development team and keeps everyone up to date.

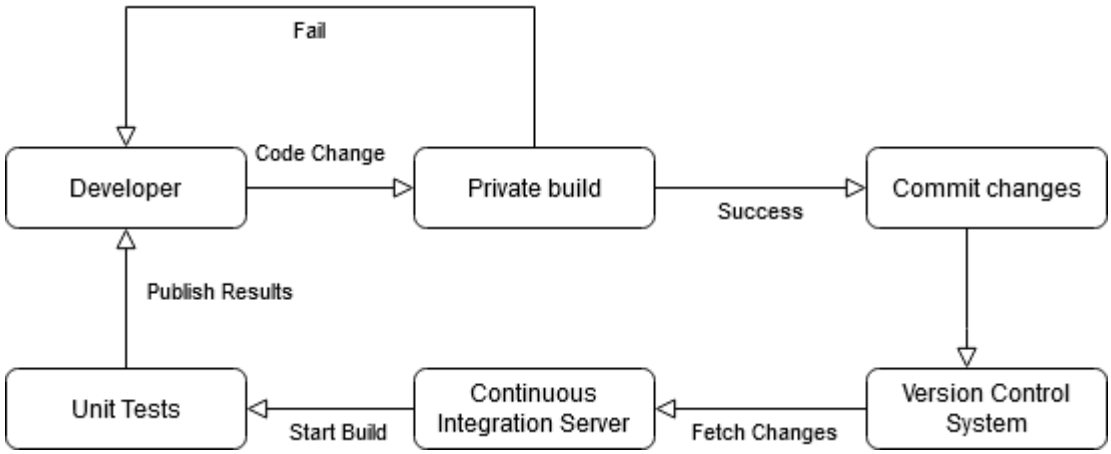

Figure 3. Basic CI implementation

2.4 Virtualization and Docker

Virtualization is a method where SW layer is used to create a virtual version of physical HW with the same inputs, outputs and behavior. In other words, virtualization enables the usage of completely different environment than directly provided by the original HW system [13]. In this section on virtualization, two technologies are discussed: virtual machine monitor (VMM) and containers. Some of their features are inspected and pros and cons are compared. After discussing virtualization, a closer look is taken at open source containerization engine Docker and process technology features which make Docker suitable for the thesis work.

VMM can encapsulate the whole VM's SW state and (re)map VMs to HW resources. A VMM must work as a high priority SW and it typically runs alongside or under the OS, as it is intended to work as an isolated duplicate of a real machine [13]. High priority SW can generally perform any operation the HW supports.

Figure 4 represents two different types of VMMs and shows their main differences. VMMs have two basic architectures: Type 1 is usually referred to as a bare metal VMM and Type 2 as hosted VMM. Bare metal VMM is installed as the main system on the hardware, giving it full control over any VM. Hosted VMM is either above or alongside the host OS, allowing them to share drivers.

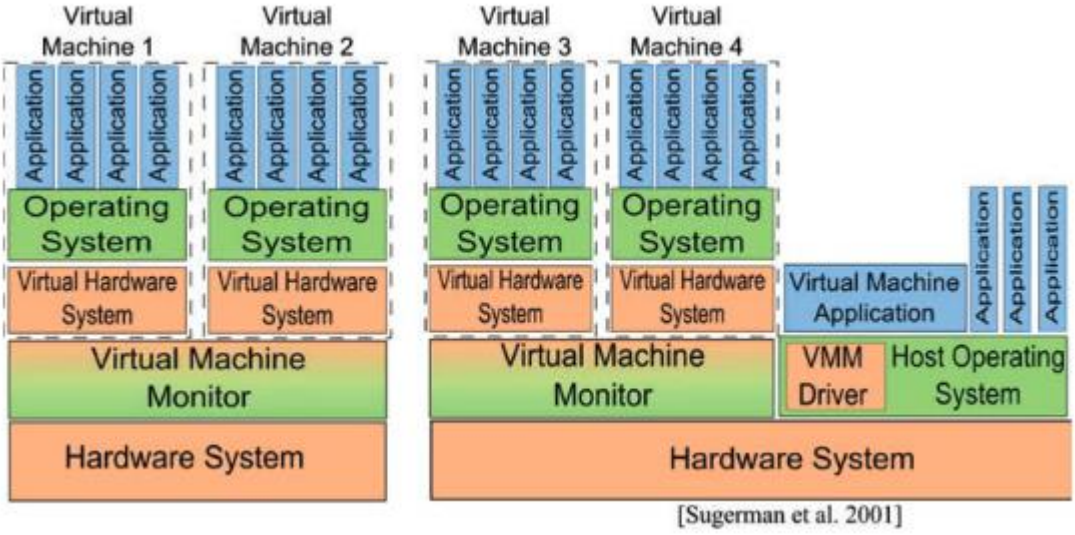

Figure 4. Two architecture types of VMMs, left is bare metal VMM and right hosted VMM [13]

Bare metal VMM functions as the main boot system on the HW. Therefore, the VMM operates on highest priority, meaning it has full control over all VMs using it. Hosted VMM is installed on the HW's OS, which results in a bit more complicated structure. Hosted VMM can share drivers from host OS, meaning the VMM system does not need HW drivers and is able to use VMs in the current environment. As a result, it is not necessary to migrate the host OS to multiple boot arrangement [13].

VMM is commonly used to stabilize bare metal servers, SW debugging and guest OSs. When VMM is not relying on the HW OS, system virtualization makes it possible to install multiple instances of different OSs on a VMM, allowing multiple guest OSs on each physical system [13]. This kind of setups are common in data centers.

Containerization is a good alternative for virtualization, for containers are newer lightweight technology compared to VMM. Containers use the same shared OS kernel as the host machine and can run several individual processes in each of the containers. Container-based design has smaller disk images than VMM, due to the shared kernel [14].

The main difference between VMM and containers are their way to virtualize and isolate. The relation between traditional VM and container can be seen in Figure 5, where container is isolating applications at OS-layer, while VM-based isolation is done by the OS. VM requires installation of OS to be able to run applications, meaning the resources must be isolated to separate guest OSs. By contrast containers are isolated on OS-layer and are built on top of Linux primitives [15].

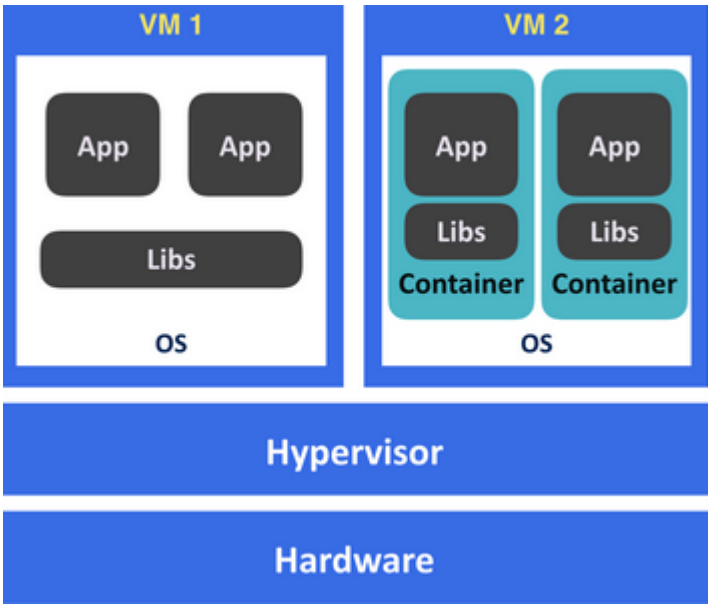

Figure 5. Relation between VM (left) and container isolation (right) [15]

.

Containers consist of multiple building blocks, where namespaces and control groups are the most important. Both of these blocks are Linux kernel features. Namespaces enable logical partition, providing processes different system resources and isolation between containers. Control groups on the other hand enables isolation and control over CPU, memory and I/O [15].

Both virtualization technologies, VMs and containers, have their own pros and cons. Table 1 represents comparison of their major attributes and gives a preliminary perspective of where each of these technologies are well suited. VMMs virtualize HW and device drivers, and run OS on top of the virtualized HW, causing overhead. Containers are able to avoid this overhead by using OS-layer isolation [14]. Containers can achieve native performance level as there is no need for them to simulate HW. Shared kernel and OS libraries provide high density virtual instances and reduced disk image size since OS is not included in the image [14]. Being lightweight allows containers to start very fast, especially when compared to VMs. Container bootup may take only few seconds, while VM might take multiple minutes.

Table 1. Comparison between VMs and containers [16]

Virtual Machines (VMs)	**Containers**
Represents hardware-level virtualization	Represents operating system virtualization
Heavyweight	Lightweight
Slow provisioning	Real-time provisioning and scalability
Limited performance	Native performance
Fully isolated and hence more secure	Process-level isolation and hence less secure

Sharing kernel also causes few downsides for the container approach. Containers cannot isolate resources as well as VMMs, as the host kernel is exposed to the containers, and might cause security issues [14]. Another limitation of kernel sharing is the inability to run Windows-based containers on Linux host machine. Windows host is able to run Linux containers, as Windows uses Linux VM for the container. As a payoff, this solution reduces the container's performance level to the VM level [14].

Docker is a containerization engine for automated packaging, shipping and deployment of SW applications. As an open source application with specified user-friendly design and cross-platform nature, Docker has become a very popular container engine.

Docker uses images in the same concept as VM: Docker image is an illustration of a system. As the main difference, VM image can have running services, and Docker images do not hold kernel within. As a result of these properties, Docker image is much lighter than VM image, but requires Linux running on host machine. If the host machine does not have Linux as OS, Linux kernel can be provided, for example, by Hyper-V manager. Despite their differences, a Docker container can perform similarly as VM instance [16].

Docker uses Dockerfiles to set up Docker images. Dockerfile is a text file written in domain specific language, which defines instructions of architecture and build order of the Docker image [16]. The Dockerfile is optimal to be stored in VCS, since any developer can build identical Docker image from it and it requires very little space. For its human-readable form, Dockerfile can be easily adjusted to suit developers' specific needs, for example, if additional tools are needed in the container environment.

Docker image must originate from a base image and can inherit as many images as required for preferred behavior, as visualized in Figure 6. Docker images are formed by data layers, and the data layers reflect instructions from the Dockerfile. These layers provide efficient Docker image storing, as each unique layer is stored once and can be used by multiple images. Therefore, layers improve Docker image build time, since already formed layers can be used to other images.

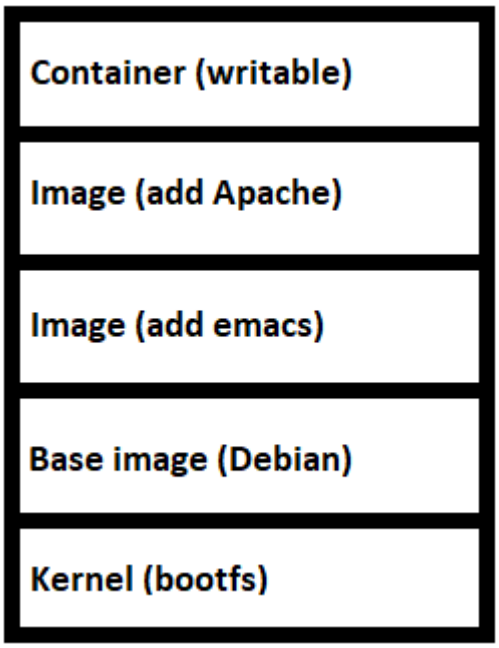

Figure 6. Docker container build [altered from source 16]

Usually the base Docker image works as an OS, for example in case of Linux, the base image is one of Linux distributions. Docker images are used to build the Docker container [16]. Once the container is running, it is still possible to install more packages just like in a regular Linux distribution.

Docker Hub is a namespace repository for Docker images. These repositories allow developers to distribute Docker images within project team and customers. Docker Hub has become a popular option for companies to store their Docker images: it is free to use and its commands are very similar to Git [17].

2.5 Yocto

Yocto is an open source collection of tools, templates and methods for development and deployment of custom Linux-based systems for embedded applications using any HW architecture. During the production phase, Yocto's tools are in use, and previously built utilities and other SW components are reused while building new applications, libraries and SW components. The Yocto tools setup their own build environment, toolchain and utilities to lower the required host SW dependencies [18]. The discussion in this section will focus on projects under Yocto, such as BitBake and OpenEmbedded, followed with development usage of Yocto.

Poky acts as a reference distribution of Yocto, and it provides metadata, tools and mechanisms to build Linux SW stack platform-independently. The essential parts of Poky are

OpenEmbedded core and BitBake tool, in combination with default set of metadata [19]. OpenEmbedded core is a set of recipes, classes and files for multiple different OpenEmbedded systems, including the Yocto. Yocto project is built in layers, where OpenEmbedded core functions as the base layer. The layer format provides several advantages: it helps to extend functionalities, contain errors and ease making changes to the build [19][20].

BitBake is a task execution engine that parses Python and shell script code, and takes care of the complex dependency constraints. At high level, BitBake can read metadata, determine required tasks and run these tasks in parallel. BitBake has few attributes those provide it the ability to function properly: recipes, configuration files, classes, layers and append files [21].

To manage the SW built flow, BitBake uses recipes, which are files with .bb extension. Recipes provide BitBake package information, dependencies, source code specifications and more detailed information. Self explanatively configuration files define numerous configuration variables that control the build process of the project. The configuration files define and include machine, distribution, compiler, common and user configurations [21].

Class files provide valuable information for files containing metadata. They share information regarding standard tasks: compilation, configuration, unpacking, fetching, installation and packaging. In larger projects, other classes usually override and extend these features according to their own needs. Yocto projects have numerous features and customizations, and layers provide opportunity to isolate them from each other. Dividing entities into separate layers provide modular system, for example keeping support for target specific machine in its own layer, making adaptation to future changes fluent. Finally, append files are used to override and extend recipe information. BitBake assumes all append files to have a matching name with recipe files. Append files allows developers to create project-specific specifications without modifying generic recipes [21].

Yocto’s development process begins with creating recipes and images, for which Poky has to offer few ready-made recipes with simple functionalities that developers can use for testing and development. With the help of BitBake, images can be created from recipes without additional intermediate steps. Before and during the write and test phase of the application, multiple different libraries and tools are required. Despite the addition of libraries and tools during the project, developers need test environment adequate to the final version to match toolchain compatibility and to avoid behavioural changes. Poky

offers a solution to this, as it can generate SDK packages that can be installed on any machine [18].

There are two different kind of SDKs Poky can create: generic and image-based. Generic SDK provides developer with basic toolchain with cross-compiler, debug tools, libraries and header files. The use of the generic SDK is mainly suitable for kernel and bootloader development and debugging, but usage of image-based SDK is still highly recommended, as it is more fit for the application's specific requirements. Image-based SDK is created from custom made image and it contains all dependency libraries, tools and HW architecture of the image [18].

Regardless of which SDK is chosen, its usage in the development flow can be seen in Figure 7. The SDK can be installed on separate machine or development environment from Yocto, allowing developers to work independently. Once the developer has finished with compilation and tests, the changes can be integrated to an image. Once the image is ready and shared with Yocto project machine, the image can be rebuilt to produce new modified image [22].

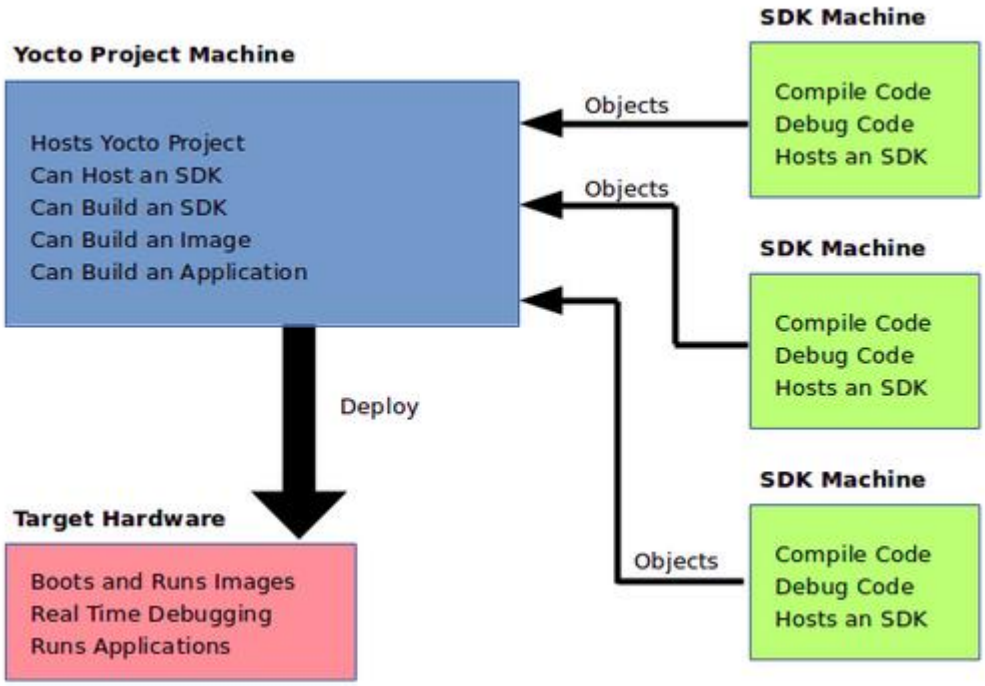

Figure 7. SDK in development process [22]

2.6 Related work

Cases of building coherent development environment using Docker and containers are not uncommon, as multiple references can be found from the internet. However, it is quite rarely specifically mentioned and gone through in use cases, if the case does not precisely process it. Finding scientific texts on this subject can prove challenging, but there are plenty of coding instructions, blog posts and articles that deal with setting up Docker development environment. In addition, there is literature available regarding Docker on a more general level.

At first a book in [23] is examined, where containerization and Docker are incorporated into development workflow. Docker helps to mitigate unnecessary steps in the workflow, by giving developers better access in the build environment. This reduces the amount of support need, as developers have user permission to solve their own problems. In addition to providing coherent build environment, Docker guarantees efficient testing, as the test case artifact can be reproduced in the production and developer environments. These features are answers to the needs of this project work, as the goal is to make coherent build environment which can also be used in CI.

Then, a look at the book in [24] is taken which goes through Docker's basic use cases, features and setup. There are examples of implementation models which can be efficiently done with Docker: development and build environment, shareable local Docker containers and Jenkins CI. These fit this project's needs and show that the planned implementation can be done with Docker. The simple and precise instructions provided by the book are good for getting started with the subject and they help to make efficient solutions in different steps. Importantly, the book demonstrates that Docker is well suited for the project. Docker does not only speed up the development environment setup and workflow, but it offers synergy with CI testing. As CI process often requires software installation, test running and clean up, it may turn out to consume large amount of resources. Docker however provides cheap deployment and cleanup, which is an advantage in this project.

A blog post in [25] discusses challenges their development team is facing and introduces Docker and containers as a way to relieve them. Their problems mainly consist of a large number of required tools, dependencies and variation of OSs, causing tedious development environment setup and cross-platform issues. The above problems are also reflected in the onboarding process. Last issue mentioned in the post is developers running integration tests on shared platform and not locally. Running tests locally would allow developers to execute them on different versions and branches. The test environment

setup has similar difficulties as development environment setup. As a solution the blog post introduces containers as development machines. Docker containers have same OS and toolset, developer only has to setup Docker, and developer can apply desired version and branch to the container for testing. Despite containers good qualities, the blog post mentions drawbacks: use of bind mount removes necessary dependencies. Some workarounds are discovered and introduced with the cost of setup time which is directly proportional to the size of the project.

Another blog post in [26] takes a position from a slightly different angle: it reveals what problems they have encountered during and after setting up the Docker container environment and what solutions they found and used to fix them. The challenges are focused on cross-platform usage, write access to volumes and running containers on the same host port. Docker requires Linux to function, therefore developers using Windows and Mac need some way to run Linux in their environment. The blog post introduces Vagrant and boot2docker as a solution, for they can create a minimal Linux VM for running Docker. Docker is capable to mount host filesystem into the container as volumes, which can cause access issues. On Windows the folders get world-writeable Linux permissions of 777, but on Mac and Linux, the file ownership and permissions do not automatically work. On the host system, the User ID (UID) and Group ID (GID) are different than in the container.

In this thesis, the file ownership problem has been fixed by adding the local user's UID and GID to the container, providing access to all mounted files. In the blog post the problem is solved with Vagrant feature, which easily provides the ownership to the mounted folder. In the blog post, the development team is working on web applications, what for they expose port 80 for HTTP connection to the container. Unfortunately, only one container can be bound to a given port, which becomes impractical when large quantity of containers are in use. Given workaround for this problem is to run each application in VM via Viagrant. The solution should be used at its own discretion on a case-by-case basis, as the use of VMs removes some benefits of the Docker.

An article in [27] goes through a case with similar need of build environment and the steps how they used Docker to fulfil those requirements. In the article, their build environment requires large number of specific programs, libraries and modules with certain tool versions on top of correct OS. The reproducibility of such environment is the main challenge, as there is a need to share analysis methods and programs with other users and partners. Docker provides a platform-independent reproducibility of containers and transforms the challenging environment setup into a simple set of commands. The initial

stage of the project is very similar to the article's, for a reproducible development environment must be implemented to ease the setup process and to remove compatibility issues. This project takes the implementation a step further than in the article, as the setup script does the entire environment initialization on behalf of the user.

As can be seen, there are plenty of similar cases and solutions implemented as the thesis project. The transition from a similar initial situation to the same implementation model provides assurance that the correct solution model has been chosen for the project. In addition, having many sources and models of how to implement the design makes the development process much easier. Other container technologies like Docker are available, but since other departments in the company have already done implementations with Docker, it is wise to choose Docker. Even when looking at the situation without the internal initial condition, Docker seems like the best solution. Docker supports all aspects of the project's implementation, while providing comprehensive documentation, variety of examples and user friendly environment. When inspecting other container technologies, such as LXC, they seem to lack some of these features. With LXC, the environment setup process and performance stay very similar, but there is not as good support for Jenkins CI as with Docker [28]. Other considerable difference is the lack of examples and guides. If developers run into problem scenarios, they must be able to find solutions easily online, as they are not expected to have previous experience with the technology. There are no notable downsides in using Docker, and bringing other similar technology might cause problems in the future, for example, if the operating models of two different departments are combined.

3. PROBLEM ANALYSIS

In this section the focus is on the problems and challenges in the current development environment and build process. First, the current situation is gone through covering currently used technologies by developers, development workflow and build environments. After this the survey for the developers using the build environments is discussed. The survey mainly inquires developers about their current development environments and their experiences with them. Lastly, build environment requirements are described as well as setup process, and they are reflected to the Docker environment's preconditions.

3.1 Current state

This section presents in detail the current and upcoming state of technology in SoC SW department. In addition to familiarization with the technology, some of the major challenges the developers have to face on daily basis is gone through.

There are few primary development environments in use in the department: Linux Software Engineering Environment (LINSEE), Oracle VM VirtualBox and personal work laptop. They are all used for development, but their main differences can be seen in Table 2. Environment description gives a brief description of the initial state. OS indicates which operating system is in use, e.g. Red Hat Enterprise Linux (RHEL). Lastly, maintained by informs which party is responsible for the maintenance of the service.

Table 2. Currently used SoC SW development environments

Environment	**Environment description**	**OS**	**Maintained by**
LINSEE server	Programming environment with preset tool packages	RHEL	IT
Oracle VM VirtualBox	VM in own use, requires full setup by developer	RHEL	IT
Personal work laptop	Company tools, requires own development environment setup	Windows, Linux	Developer

LINSEE is a programming environment for SW development on RHEL platform accessible by Secure Shell (SSH). LINSEE contains 3rd-party and in-house tools, those are suitable to assist developers with SW architecture, design, implementation, unit testing, building and CI. The services are provided and maintained by IT with the assistance of representatives from main user groups. As pros, the LINSEE environment has vast variety of support services, modularity, license management and version-control for tools and toolsets. Developers can get assistance in the environment setup, and individual developer can install or update tools independently to suit their own needs. The cons however overrule the pros: tool management is difficult, testing for dependencies is tedious, development lacks compatibility, RHEL is outdated and the environment lacks documentation regarding proper environment setup.

The toolset management is implemented with setseeenv command, which delivers tools in package. As the developers can customize these packages for their own liking, there is no single toolset for all developers. An initial setup of this sort causes differences in the development environments, causing compatibility problems. There may be significant differences between different tool versions, meaning developers with different toolsets might not be able to run each other's products. Running into such malfunction can be time consuming and tedious. There is no tool provided for version and dependency testing, leading developers into manual work. LINSEE environment has only support for

RHEL, which has outdated tool versioning. Lastly, the LINSEE environment lacks documentations. IT only covers the basic steps, such as how to access the environment and how to activate toolsets. Since individual developers can customize their tool versions, it is overwhelming to keep track of each separate development setup. In addition, the impact can be seen when developers start with new projects and when newcomers begin their first projects.

Oracle VM VirtualBox provides personal VM on RHEL platform for the developer's own use. IT provides guide for the VM setup, but everything installed on the VM is taken care by the individual developer. Having root access in the VM gives developers freedom to choose their own tools and versions to use, however it might be time consuming and troublesome for less experienced developers. Oracle VM also has some of the same problems as LINSEE: developers do not have uniform development environment, VM supports only RHEL and there is lack of documentation.

A personal work laptop is provided for each employee. The laptop has Windows 10 as default OS, but developers have the possibility to run any Linux distribution. However, Linux is not the recommended OS for many company tools are easier to use on Windows. The developer needs access to Linux distribution to be able to build product SW. Without using the provided solutions in Table 1, developer with Windows OS on their laptop must come up with their own solution, for example containers. Regardless of the time it takes to setup such environment from scratch, the major problem is the same as with LINSEE: as there is no uniform toolset versioning, the development environment is not compatible with other developers.

The current main tools for build automation and CI used in SoC SW department are Gitlab, Gerrit, Zuul and Jenkins with its plugins. In this setup, Gitlab and Gerrit work as the code review and version control system, Zuul as the project's gating manager and Jenkins as CI automation server. All these tools are connected together and are part of developer's workflow.

Gerrit and Gitlab are used as tools for code reviewing, and from a CI's point of view their functionality and purpose is the same, giving the opportunity to inspect them simultaneously. Both Gitlab and Gerrit manage Git repositories and provide more advanced project management and code review. The code review functionality works as follows: whenever a developer makes a change into a repository, the code goes through automated tests to check correct functionality, and other developers have to evaluate and grade the changes. If both automated tests and developers' reviews pass the code is accepted and

the changes submitted, or if either aspect of code review does not pass, the changes are rejected, and developer has to fix the defects [29][30].

Jenkins is an open source application for CI and build automation, and it is used for building and testing projects. There are tremendous number of plugins available for Jenkins those provide wide range of features, such as static analysis, unit tests, unit test line coverage, code complexity and so on. With the help of plugins, Jenkins can also communicate with VCSs and be integrated with servers, clusters and containers [31].

Zuul is an application which functions as a gate between source code repository of the project and CI service, such as Gerrit and Jenkins. Zuul's functionality revolves around pipelines, those are workflow processes which can be applied on one or more projects. There are multiple different workflow processes, and few of the common ones are check, gate and post types. Check pipeline determines which tests to run on newly submitted changes, gate pipeline controls automated merging when tests pass, and post pipeline usually updates project information, such as documentation. Once pipeline has executed its whole queue, the pipeline's reporter triggers giving results of all the jobs. Top level modeling of Zuul's connection to Gerrit and Jenkins can be seen in Figure 8 [32]. In the workflow, developers fetch and push code from Gerrit. Code change in Gerrit will trigger event, which Zuul will notice and respond accordingly. Zuul server interacts with Gearman server and Zuul merger. Gearman is a protocol for executing tasks on distributed workers and it communicates with Jenkins.

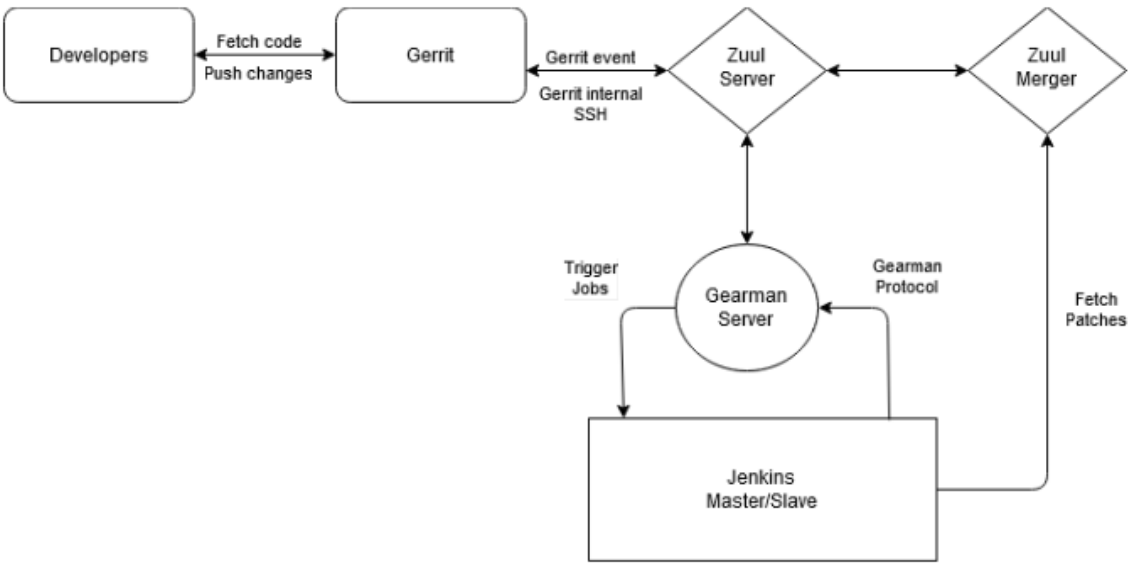

Figure 8. Zuul workflow

In SoC SW organization there are few basic principles regarding Jenkins jobs and Zuul implementation. As developers have to create their own Jenkins jobs, it is important to keep them as simple and re-usable as possible. Simple and small jobs ensure that there is no unnecessary overlapping, and re-usability provides possibility to use same Jenkins jobs for different Zuul pipelines. As Zuul is being used, it is attempted to maximize the test execution control to it. In practice this means shifting as much control from Jenkins to Zuul as possible.

The workflow in SoC SW department is currently going through changes as it will become more like Figure 8. The workflow model is already in use by other departments in the organization, giving a readymade operating model. The basic principles are as follows: developers work on the same repository which is conventionally the master branch. When developers are ready with their implementation, they run local tests. Successful code is then pushed to Gerrit, that will trigger test suite in Jenkins, which will run basic tests. The changes have to be also reviewed by other developers. The reviewers can be chosen automatically by Gerrit or by the developer who made the changes in the first place. The reviewers will evaluate the and give it a grade from -1 to +2, giving indication how well it is done. If the reviewers grade the change +2 and it passes all the Jenkins tests, Zuul will be triggered and more advanced test suite is run. In the case where multiple changes are made simultaneously, Zuul uses change queue to ensure that multiple changes have not broken each other. If the more comprehensive tests are executed successfully, Zuul will submit the changes to the master branch in Gerrit, finishing the workflow. At any stage of the workflow a failure or a negative review will stop the process and inform the original developer who made the changes. Then the developer can adjust the code and restart the process.

3.2 Survey

To get a better view of the current state of the provided build environments and tools, a survey for the whole SoC SW department was held. The survey was sent to approximately 200 developers, whose job descriptions are generally trainees, engineers and leaders. The survey's main focus was to gather information on the development environments those the developers use, what strengths and weaknesses the environments have and how the environments could be improved. As another perspective in the survey de-

velopers were asked if they have any earlier experience with Docker and general attitudes towards new Docker development environment. In this section the results of the first survey are discussed.

The survey was sent via email through common distribution list to whole SoC SW department. In the end only 11 answers were given, but enough data was collected to make reasonable conclusions. The survey had six open-ended questions and it was done completely anonymously so that the respondents could give as real and honest answers as possible. The survey's original frame can be seen in Appendix A. The survey's subjects and the thoughts behind them are the following.

1. What build environment have you been using (LinSEE server, oracle VM, container, other -> what)?

   At first, basic information regarding the used build environment was collected, which could be LINSEE server, oracle VM, container or basically anything else. Knowing this is important for the following questions and it provides direction of the most popular build environment.

2. Pros and cons of your current build environment.

   Developers' opinions regarding the currently used build environments were asked. It is essential to know which properties work well and which areas need improvement. Such information may be helpful in the future decision making while building the new Docker environment. The information also clarifies why certain build environments are used more than others, and why some developers have built their own environments from scratch.

3. Have you manually set-up the build environment (compilers, host tools, etc)? If yes, give a time estimation how long this took.

   Since the new Docker environment eliminates the need of manual set-up, information was gathered on how many in SoC SW department has had to manually initialize the build environment. In addition, an estimation was asked on how long the set-up took, to get a better picture of how many working hours the new environment might save from the developers. This question also revealed the average amount of critical issues during set-up.

4. How the current build environment could be improved?

   The given suggestions give direction towards what the developers wish from their development environment. It also helps to ensure that any important features or problem areas were not overlooked. In addition, the desires for the fea-

tures that the new Docker environment covers provide reassurance that the project is important and timely.

5. Do you have any earlier experience with Docker?

   This question provides an overview of how familiar the concept of Docker and containers are in SoC SW department. This gives the first impression of how challenging the deployment of the new environment might be and what kind of reception this project might get, since new implementations may not be very welcome.

6. What expectations you have from the new Docker development environment?

   This question gathers the features and functionalities desired from the Docker environment and project. The answers help to focus more on key aspects and possibly cut out a few unnecessary features. In addition, there is feedback on this issue, with negatives playing a more important role. They help to locate the biggest pain points as well as threats, and possibly negate them during launch with demo presentation or question and answer section.

As the answer provided were open-ended, the nature of the answers had some variation. In general, the answers were clear and precise, so nothing had to be left to interpretation or omitted. Next, the answers are gone through and some analysing is done.

Based on the first question, two most popular development environments are LINSEE server and Docker containers. Few developers mention using oracle VM, but it never was their only environment. IT does not provide support for container environment, meaning developers have initialized them themselves. However, it is known that some developers are sharing their container setups within smaller teams.

Developers clearly highlight similar strengths and weaknesses in the environments. Mentioned pros and cons are gathered to Table 3. Data was collected from each said environment in the survey, and only topics those were mentioned more than once were taken into account.

Table 3. Pros and cons of the development environments

Development environment	**Pros**	**Cons**
LINSEE server	- Can be left running over night - Has prebuild Kernels and buildroot - Good availability	- Needs configuration - Long setup time - Sometimes overloaded - Missing centralized tools
Oracle VM	- No need to configure	- Different variations of Linux distributions and tools - Slow - Need to compile Kernels
Docker containers	- Resettable environment - Can be run anywhere - CI is using same containers - With mount able to use own editors	- First creation and configuration is time consuming - Authentication issues

The answers are not much different from the points discussed in the previous section. LINSEE's biggest flaw seems to be tedious and time-consuming setup. There are mentions that this is especially evident with newcomers. A couple of new perspectives worth noting are LINSEE's slowness due to large user numbers and the ability to leave tasks running over night. Since the LINSEE servers are available for every developer and it is one of the more popular build environments, it seems possible that the servers get crowded often. As a good feature larger tasks can be left running over night, which optimizes the use of working hours. On the other hand, a large simultaneous start-up of heavy tasks at the end of workday can significantly congest servers and affect the efficiency of those working in other time zones. Efforts have been made to resolve this

problem by providing ‘nice’ parameter, which prevents individual builds from hogging all resources.

Based on the survey responses, Docker containers are already in surprisingly high use by individual developers. Voluntary use of containers shows that developers are comfortable with the use of Docker and are familiar with the technology. The good features of Docker containers can be summarized as being flexible, self-controllable and usable on any platform. In addition, a lot of attention was given to the opportunity to use any editor via mount. As main con the developers mention the difficulty of initial setup, meaning Dockerfile and other possible steps.

Most of the developers have done a manual setup for the build environment they use, and there are only mentions about doing so for LINSEE and Docker. On LINSEE when everything works as intended the setup time takes around one working day, but some developers mention running into trouble which caused the setup to take on average two weeks. A clear and up-to-date documentation on how to do this setup would probably reduce these problem cases significantly. On container’s side, developers using a ready-made base container environment have had to install some tools to their environment, which takes around few minutes. One responder has made the whole container environment in small steps over a couple of months, but no real time estimation can be given from this case. Based on the responses, containers are faster to setup than LINSEE environment.

The improvement suggestions had lots of similarities. Regarding LINSEE server, many developers mention moving over to Docker would be smart. These answers might be biased since the developers should be aware of the intentions of taking Docker into use and some of them state later that they do not have earlier experience with Docker. Other suggestions were to raise performance capacity, use same tool versioning and create extensive documentation. Since the new Docker development environment is partly replacing LINSEE, it will provide all of these improvements, if enough developers will run Docker outside of LINSEE servers to reduce its load. For currently used containers, developers wish for clear instructions and possibility to perform multiple different tasks in a single container. Both of these features are provided in the new environment, suggesting a good starting point for the project.

The acceptance of the new build environment appears to be generally positive and the expectations are realistic. Most of the developers have little to none experience with Docker, which clearly reflects on the given answers. Most of the developers wish for a well-documented, easy to use and uniform environment. Some negative feedback was

also given, where the developers were afraid of unnecessary complexity. The new environment should provide everything desired and remove the need to do complex setups.

3.3 Preconditions for the docker environment

To have a better understanding of the choices made during the project, next some of the preconditions are discussed, before going through the implementation process of the project. At this point it is known that the project will be implemented with Docker containers, as it fits well the conditions and it is being used by other departments in the organization with good results. Table 4 summarizes the relevant problem scenarios in the current environments and shows planned actions for them. The columns in table 4 show the scenario, its current problem and provided solution. This chapter will go through topics in table 4 and analyse them.

As discussed earlier, the major problems with current build environments are maintenance and usability issues. The build environment has swelled into a complex and poorly documented system that led into compatibility issues. In addition, running the build environment on the developer's personal laptop is time consuming, since the developer has to search for the project specs, set dependencies and still possibly debug compatibility issues. The goal is to create a new environment which will eliminate as many of these problems as possible.

Table 4. Problems in current build environments and solutions for them

Scenario	**Problem cases in current environments**	**Solutions**
Maintenance and usability	Lacking documentation Compatibility issues Swollen and complex	New build environment
Tool and environment setup	Time consuming Error prone	Uniform toolset Automated setup
Documentation	Inadequate Differences between environments	Documentation covers whole setup process Step-by-step instructions Guide for basic debugging
Changing between projects	Requires manual actions from developer	Automated project swap with script

For the developers to be able to build the product, the development environment requires Linux OS, specific set of tools and environment variables. The first criterion is to provide environment with uniform toolset for all developers. Since the setup process can be time consuming and tedious, automated setup process is provided, where the user does not have to perform any kind of initialization. The new development environment should be agile, as well as accessible from any environment, such as developer's laptop. Being able to grant every developer root access without security threats would streamline the working experience.

As the current development environments lack good documentation, it is one of the main objectives to have comprehensive documentation for the new environment from the beginning of the implementation process. The new development environment should have non-existing learning threshold, referring that developers with no previous experience with the technology should be able to use it. The developers are provided with specific step-by-step instructions on how to get started in the new environment. In addition, instructions for fixing common problem situations are needed.

A single developer can be part of multiple different projects simultaneously, meaning they must work with different toolsets and dependencies. Therefore, it is important that an easy way for the developers to change between the projects being worked on is provided. From workflow's perspective, currently SoC SW department has separate environments for development and CI. As CI flow uses Docker containers in Kubernetes cluster, a single environment for both development and CI is implemented. The environment setup has to be executable from Jenkins jobs for the CI process to function properly. This means the automated setup process must work without user input by providing parameters for the Jenkins job.

4. DESIGN AND IMPLEMENTATION OF DOCKER ENVIRONMENT

In this section the implementation process for the new Docker container environment is gone through and how solutions for the problem situations discussed in the earlier sections were found. In addition, it is discussed how the survey feedback from section 3.2 is taken into account during the implementation.

SoC SW department already has a VM build environment in place, so a different approach was taken. Docker was chosen as main technology in the project, and containers were used in the implementation of the build environment. Containers basically function as lightweight and agile VMs with the ability to get the developer into the build environment with a single command. In addition, Docker has been successfully used in other departments and it is currently the most supported container runtime engine. Additional confirmation and approval regarding Docker was received from few developers in the SoC SW department, as they have previous experience with Docker and containers.

In section 4.1 Docker image's and container's creation process is discussed and the thoughts behind the decisions. Docker image is created from a Dockerfile, which is simplistically a human-readable instruction file. Dockerfile generally contains dependencies, tools, scripts and user handling for the container. The actual container is then created from the Docker image with desired flags and settings.

The most important aspect of the project's implementation is discussed in the section 4.2: creation of the initialization script. Developers have to be able to start and use the container with ease without any previous experience with the technology. As the environment has several prerequisites, the Docker commands become quite challenging. For this reason, a start-up script which the developers can use to launch the container is created. The start-up script should be easy to use and the developer must be able to choose the project to be worked on. In the 4.2 subsections a closer look is taken at developers personal OS, user handling in the container, directory mounting, environment variables and clean-up steps.

Finally, the documentation process is discussed in the project in section 4.3. The documentation is kept in SoC SW departments internal sharepoint, where all the developers have access. The documentation provides developers with steps on how to start the container environment, what flags they can set, basic debugging tips and more. As time goes the development environment will change and improve, and future projects might

bring new challenges. This will create the need to update documentation and the initial plan for this is gone through.

4.1 Docker image and container

Using Docker in project usually begins with the Dockerfile, where the first step is to decide on the base image. Current development environments use RHEL and one of the goals is to get rid of it. Intention is to use updated Linux distribution which supports required tools and needs. Ubuntu bionic beaver is chosen, since SoC SW department already had its base image. The Ubuntu image is stored in artifactory to ease the use of it. As Ubuntu can provide same features and toolsets as RHEL those are required, no compatibility issues should appear.

Next step is to add instructions to the Dockerfile to provide the Docker image with required tools and other functionalities. During this project there was no need to create a new Dockerfile from scratch as SoC SW department had a sufficient one with all required tools ready in the git repository. A multi-stage build functionality was used in the Dockerfile, which allows to start a new build stage from where the old Dockerfile ends. Figure 9 represents the way Dockerfile installs tools and how multi-stage building is being used in the project. In this use case the image size is not too big, a bit under 2.0GB, and the image build time is only few minutes, so there is no need to chain the instructions. Multi-stage building enables multiple different variations of the base image. In this project two different images are used to match developer’s needs: one contains native SDK and another project specific SDK.

Once the Docker image is ready, any number of containers can be created from it. Developers working in the LINSEE server can use same image to launch own containers, so no excessive image building has to be done. As there is the possibility for multiple developers to use the same Docker image, no user specific functionalities are added inside the Docker image.

```
...
RUN apt-get -y update
RUN apt-get -y install apt-utils
RUN apt-get -y install gawk
RUN apt-get -y install wget
RUN apt-get -y install git-core
...

FROM soc-sw-docker... AS builder

USER root

...

FROM builder AS nativesdk

...

FROM nativesdk AS targetsdk

...
```

Figure 9. Tool installation in Dockerfile and multistage building from it. Three dots represent missing parts of code

4.2 Initialization script

Simple Docker commands can be demanding as they may have dozens of detailed flags and options. Since significant portion of SoC SW department's developers have no earlier experience with Docker, an initialization script was created which will create required image, container and setup for the developer. As the project's implementation process went forward, there came need to pass the container with more and more data. Mainly the data is provided to the container environment via volume mounting, artifactory and environment variables. Dockerfile can handle package's and environment variables, but it is good to keep the Docker image as generic as possible. For this reason, this is done during the script, for what Docker has efficient commands. With these requirements the container environment setup becomes rather complex, making the script necessary for even experienced users. In this section the script's implementation and decision process is discussed, from developer's personal environment all the way to the clean-up steps.

4.2.1 Operating system

Docker can be used with all major OSs: Windows, Linux distributions and macOS. Developers in SoC SW department mainly use Linux and Windows, meaning the script has to function on both OSs. There is no plan at this moment to provide support for macOS,

but if individual developers wish to use it, they can use the initialization script once they have a container up and running. Since Windows and Linux cannot unambiguously use the same script format, separate scripts for both environments were created: bash shell script for Linux and batch script for Windows. Both script types work on the same principle, they execute series of commands stored in a plain text file with proper file extension. This setup requires a bit of extra work as the changes and new functionalities have to be done twice.

There are practically three different starting positions where the developer uses the environment: personal laptop running Windows, LINSEE server running Linux and Linux environment chosen by the developer, such as VM or personal laptop. The initialization script, Dockerfile and other necessary files are stored in git repository, where all the developers in SoC SW department have access. LINSEE is the most straightforward to use from developer's perspective, as it already has Docker preinstalled, leaving developer with only the responsibility to request user permissions to use Docker, and to fetch code from repository. Once the developer has fetched the files to their workspace on LINSEE, they run the bash shell script in the cloned folder, which setups the container environment. If developer works in Linux environment outside of LINSEE, they have to install Docker before they are able to use it.

On Windows environment there are few more additional steps before developers can begin to use the setup script. First of all, the implementation is currently using newest Docker version 19.03.5 and there is no guarantee that the setup functions as intended with lower versions. In order to be able to install particular version, the Windows OS has to meet certain requirements. As stated before, Docker needs Linux VM to run the containers. This is supported via Hyper-V manager, which can be installed or activated from the Windows control panel. In addition, Windows container features must be enabled for everything to function correctly. After developer has done this initial setup, they can fetch the files from the git repository and run the initialization script to setup the container environment.

4.2.2 User handling

Linux has strict file and folder access control, which is bound to current user's UID and GID. As multiple volume mounts are performed from host machine to the container, it is necessary to make certain that the user has access to all the files in the chosen mount directory. Upon file creation Linux sets by default the creator as owner, which can be later changed. Each file can permit different users and groups different access rights. To ensure smooth user experience and proper file access for the user in the container, a

user is created with same UID and name as the host machine user and provide sufficient GUI.

The container creation from the Docker image and user handling is demonstrated in Figure 10. The Dockerfile base layer provides a generic GUI which will be used in this case. For the script to be able to add the desired user into the container, it has to be provided with the information. This functionality is easy to add to the script: current user's information can be obtained with commands id -u and id -u -n, since the desired user is executing the script. After the user has been added to the container, the container has to be connected as that user in the desired group.

The user is granted with full control over the container to avoid the need of outside assistance during development process. For this reason, the user permitted with sudo rights. Sudo command allows user to execute commands as other user, in this case the superuser. This allows the user to install packages, maintain files and directories, or perform any other tasks which requires superuser rights. In the script, this is done by adding the user information into the /etc/sudoers.d/ directory with no password requirements.

```
docker container create --attach STDIN -t
--name ${container_name} --entrypoint /bin/bash ${image_name}

docker container start ${container_name}

user_create_cmd="useradd -g ${group_name}
-G users,sudo -m -s /bin/bash -l -u ${user_id} ${user_name}"

sudo_create_cmd="echo '${user_name} ALL=(ALL:ALL)
NOPASSWD: ALL' > /etc/sudoers.d/${user_name}"

user_permission="chown -R ${user_name} /home/${group}"

docker container exec -u root ${container_name} /bin/bash -c
"${user_create_cmd} && ${sudo_create_cmd} && ${user_permission}"
```

Figure 10. Initialization script's container creation and user handling

4.2.3 Mount

Mounting enables replication of filesystems from the host machine to the container environment, providing easy access of local files inside the container. The usage of mounting enhances user experience, as it allows developer to continue work progress from previous environment and to use local development tools. In this section, different forms and cases of mounting in the project are discussed, and how they are implemented.

The first necessary mount implementation is to provide the user with local work directory inside the container. This allows the developers to use familiar local tools despite the container's OS is Linux and provides easy way to get desired files inside the container environment. At first, a decision has to be made on how much data is mounted to the container. Easy solution would be to mount the project repository directory where the start script is located. However, this would bring unnecessary files to the container and could cause confusion, which led into giving the user the freedom to define the local mounted working directory location. This mount path inquiry is done in the beginning of the start script, where the user has to provide the absolute path of the local work directory. The user has a responsibility to know how to use this wisely, as mounting large directories can become quite slow, and mounting directories out of user's permissions can cause unwanted behaviour. Regardless of the chosen local mount directory, the mount path inside the container is hard coded to be /home/mount_folder. Bind mount was chosen in this use case, for it provides good performance. The mounting is done during the creation of the container, with command:

```
--mount type=bind,source=${localmountdir},target=${containermountdir}
```

The working directory is not obligatory to mount, but the developers are asked for a mount path during the script, for them to be able to save their work process locally.

The projects' code is stored in Git repositories, and as the container environment has to function as a standalone development environment, a smooth use of Git must be provided inside the container. On top of installing the latest version of Git into the container, the user's Git settings and credentials has to be acquired from the host machine into the container. All the requisite information is in .gitconfig file and .ssh folder. The ssh folder holds within the user's private SSH key, which permits the user to authenticate as itself while performing Git commands. Without SSH private key, the developer cannot perform Git push or pull commands via SSH protocol, and https authentication requires unwanted extra steps due to identification. The .gitconfig file contains user specific information, such as user name and email address. This information is used to identify the author of Git commits, i.e. code changes. As the developer has to be able to make code commits

from inside the container, this information must be provided. Getting this information inside the container causes a small complication.

On Linux environment the default location for .gitconfig file and .ssh folder is the user's home directory, at /home/<username>/. To get these files to the container, decision to bind mount the user's whole home directory to the container in the same path was made:

```
-v /home/${user_name}/:/home/${user_name}/
```

As discussed in the section 4.2.2, since the user inside the container has same UID as in the host machine, the access rights function properly. The decision behind mounting the whole home directory was to simplify the initialization script and to possibly provide the user with other useful files, such as .bashrc, which contains user specific aliases and other initializations those take action in the container. On Windows, the script asks the user for the path to the Git files for mounting. By default, their location is at C:\Users\<user_name>.

Mounting user's home directory raised inconveniency in the LINSEE servers: as the directory is an NFS (Network File System) the mounting process is slower. Developers report that the mounting process takes occasionally long period of time compared to the rest of the container's setup process. There is no intention to add an extra flag to the initialization script to tell which environment is being used, since it would make the script usage slower and tedious. The nature of this problem is complicated, as it only occurs on one developer platform and usually in more crowded time window. At the end, an end user oriented solution was made: the problem case was documented and the developers were provided with a step-by-step solution on how to remove the NFS mount, and how to get useful files from user's home directory to the container.

4.2.4 Environment variables

The last step in setting up the container environment for development usage is setting environment variables for specifically chosen project. Environment variables determine the way certain processes behave and alter tool versioning. The containers are used for developing multiple different projects, each having its own specific SDK.

At first, a project specific SDKs must be generated and stored in a location where they can be fetched into the container. The SDKs can be updated almost daily, which requires a strict tracking of the latest version. Such work becomes laborious if done manually, which gave the idea to use Jenkins to automate the task. One Jenkins job was written, which uses Docker container to generate given project's SDK and then pushes it into the artifactory. Different project's pipelines are used in Jenkins to trigger the Jenkins job with

project specific parameters. The Jenkins builds are triggered daily at midnight to keep the SDKs up to date, and only the latest version of the SDK is stored in the artifactory.

The environment setup requires multiple steps once inside the container, and as the container must be fully initialized for the developer, a separate script was made to be executed in the container. The script's main functionality can be seen in Figure 11. At the beginning of the script user gives input to determine which project is currently being worked on and the script fetches appropriate SDK from the artifactory. If the SDK already exists, only environment variables are set without fetching, and if user gives input which is not a recognizable SDK, error case echo is only given. After the fetch is done, the script unpacks the SDK, sources environment variables and sets proper kernel configurations depending on the kernel version. After the script has successfully run, the container environment is ready for the developer's use.

```
read -p "Select desired SDK: " SDK
SDK="${SDK,,}"
SDK_path=$SDK

SDK_exists=${PWD}/SDK/${SDK}
if [ -z "${SDK}" ]; then
    SDK="empty"
elif [ -d "${SDK_exists}" ]; then
    SDK="exists"
fi

case "$SDK" in
...

...
# Environment setup

ENV_VAR=$(find ${PWD}/SDK/${SDK_path} -name 'environment-setup-*') &>/dev/null
source ${ENV_VAR}

MACH=${ENV_VAR#${PWD}/SDK/${SDK_path}/environment-setup-}

VERSION=$(find -L ${PWD}/SDK/${SDK_path}/sysroots/${MACH}/usr/src/kernel
-name 'System.map-*' -type f -printf '%f\n') &>/dev/null

VERSION=${VERSION#System.map-}
VERSION=${VERSION%%-*}

if [ "$(printf "%s\n" "5.4.44" "$VERSION" | sort -V | head -n1)" = "5.4.44" ]; then
    # Version is greater than or equal to 5.4.44
    ( cd ${PWD}/SDK/${SDK_path}/sysroots/${MACH}/usr/src/kernel
    ; make oldconfig modules ) &>/dev/null
else
    # Version is less than 5.4.44
    ( cd ${PWD}/SDK/${SDK_path}/sysroots/${MACH}/usr/src/kernel
    ; make silentoldconfig scripts ) &>/dev/null
fi
```

Figure 11. Container's environment setup script without SDK fetching

After the functionality of the environment setup script was verified with manual testing, it was added to the initialization script. The script is copied to the container and sourced in bash session. The setup script runs successfully, however once inside the container, a severe setback occurs: the environment variables are not set correctly. After a brief inspection, the problem is located in docker exec command, for it runs each command in separate instance, and environment variables have to be set in each session. Docker has option –env, -e for setting environment variables with syntax VAR=foo, but in this case it does not help. The SDK fetched and used exports these environment variables within inner scripts. These scripts contain numerous lines of code, if clauses and product specific settings, making it incompatible with the Docker option.

In the end a workaround solution was found by adding the environment setup script's sourcing to the containers .bashrc file, instead of executing it during the container setup. The .bashrc file is a shell script which will be executed when bash is started, i.e. when the developer gets inside the container. With this change the environment variables are set properly and the container development environment is fully set.

A new problem raised from the solution, regarding the earlier implementation decision: as the user's home directory is mounted to the container, its .bashrc file is mounted from there. This leads the user having the changes in their .bashrc file on their local machine, which is not desired. Research revealed that there is no simple solution for excluding single files from directory mounting and adding already existing file in mounted directory causes it to be overwritten. The situation is corrected by removing the home directory mounting, and in its place, copy required files from the home directory.

4.2.5 Clean-up

To exit the container development environment user simply has to run exit command, which will close the bash inside the container. Normally the container instance will persist after exiting the container, but in this case it is necessary to be removed to avoid congesting the host machine. The removal will be done in the initialization script with following commands:

```
docker container kill ${container_name}
docker container rm ${container_name}
```

Only the directories those were mounted inside the container preserve data changes made during the development.

Based on the feedback from the developers, several of them wish to keep their container after exiting the instance. The container setup itself does not take long time, but the SDK

download and extraction require a couple of minutes. An adjustment to the initialization script was made to solve the problem by adding an option to create a new image from the container's changes. In the implementation, the user is asked for a commit tag, where blank input means no commit will be done. The actual commit is done with the following command:

```
docker commit ${container_name} ${image_name}:${given_tag}
```

Now the developer has to start the container from this newly created tagged image manually, or by customizing the initialization script to use the new image. Developers can commit as many images as they please, which, if misused, can cause trouble by consuming large amount of space.

Providing the developers with the possibility to commit their containers required few adjustments to the setup. Each time the developer enters the container environment, the environment variables have to set with the setup script, but the SDK download cannot trigger. This can be avoided by adding a simple if clause check to see if files already exist. In addition, it was noticed that the now useless part of the SDK takes over 200 MB disk space. In individual cases this would not be a considerable amount, but in case of LINSEE server, if hundreds of developers create several instances of personal containers, it starts to consume a significant amount of disk space for no good reason. For this reason, the unnecessary part of the SDK is removed at the end of the setup script.

From time to time there is a need to update the Dockerfiles, especially in the early stages of this project. Reasoning behind this might be the need of new tools, OS update or requests by the developers. Dockerfile update forces to remove the old Docker image and replace it with a new one. The removal of the old image requires a termination of all containers dependent on it, meaning every developers' container is removed. Due to this, it would be desirable for the developers not to rely too much on individual container, but to use them as short-term environments, or at least use the mounted directory as work directory. Despite this assumption, the developers have to be given a notice of upcoming update to give them time to prepare. This can be done via internal communication channels, for example one week before update.

4.3 Docker environment in CI

Once the container setup functions as intended in developer's use, it can be implemented into the CI process. The automated setup and building are done with Jenkins job. There are two different Jenkins jobs: one is used for bitbaking specific product and then it runs a test set for the produced image, other for testing cmake functionalities.

The Jenkins job's functionality is written to pipeline script. At first the Jenkins job creates a Kubernetes Pod which starts a container defined by Docker image. This project has not been integrated to Kubernetes, what for a Docker image available for SoC SW department's use is being used. Once inside the container, the setup files are fetched from Git. A container can be launched inside of a container, enabling the use of this implementation. This project's container can now be launched. During the container launch, all user input parameters must be given in advance, as the pipeline is not interactive.

First the bitbaking pipeline is examined. When the job is triggered, multiple parameters have to be set, determining product, test set, bit file version, machine and more relevant variables regarding the pipeline script. With these parameters, the Jenkins job can bitbake correct image in the first stage of the pipeline. If the image is produced successfully, second stage is started, where defined test set is executed. The results of both stages are presented after the Jenkins job has finished execution.

In the second pipeline which runs cmake, only product name has to provided when the job is triggered. The pipeline generates makefiles using cmake command and executes them with make command with different sets of exported variables. Again, the results are provided after the Jenkins job has finished execution.

Currently only manual pipeline triggering is available for both jobs, and the developers using the pipelines have to acquire the required parameters by themselves. Small adjustments to the setup scripts had to be made, as errors rose during the pipeline executing those did not appear in manual setup. In addition, the pipeline had to be defined to use bash shell to function correctly in this use case. Once the Jenkins job has finished its execution, Kubernetes automatically terminates created Pods.

4.4 Documentation

Now that the initial version of the container development environment for the SW developers has been created, the project distribution, maintenance and documentation has to be taken care of. The project distribution mainly involves the spreading of awareness of the project's existence. Announcements via the internal communication channels has been made regarding the beginning and completion of the project, but still only a small part of the department uses it. Desirably developers find and use the container development environment and share user experiences with one another, to get all developers to use it and remove compatibility issues.

The project requires active maintenance, especially in its early stages. It is expected that the developers report bugs, unnecessary features and new requirements. Significant

changes require documentation updates to keep the developers up to date. The maintenance includes updating and adding tools, as well as meeting future product requirements.

The developers need to be provided with comprehensive and detailed documentation regarding the container environment, initialization script and Docker commands in general. A large proportion of SoC SW developers have little to no previous experience with Docker, which makes the documentation particularly important. In addition, developers tend to be critical towards new arrangements and systems, and a favourable reception can be achieved with wide and user friendly instructions.

The main documentation is stored in Microsoft SharePoint, as it is used in the company, and a README.md file is provided in the project's Git repository with quick introduction and a link to the main documentation. The documentation is divided into five sections: introduction, container setup, manual container setup, how to use tagged images and managing images & containers.

In the introduction it is stated what and who for the project is, what version of Docker is currently being used, where the container can be used and key contact members for problems and feedback. This is meant to be a compact presentation of the project's essential contents. The introduction is followed with steps on how to setup the container environment, those can be seen in Figure 12. Here, the developer is provided with precise instructions for every phase of the setup, from code fetching to the exiting of the container. Unambiguous commands can be copied straight from the documentation, and options as well as examples are given for commands that require user input. In order to avoid any ambiguity, explanations are given for different stages and the container's relevant directory structure is given. Such precise instructions should be evident even for developers with less experience with Docker and UNIX.

The next section, manual container setup, explains in detail each required step to get the container environment up and running using the command line. This guidance is necessary, as some developers are going to make their own changes to the Dockerfile or container initialization. This allows the easing of their workload and clarify the functionality of the initialization script to those who are interested. This guide is also related to the next part of the documentation, how to use tagged images. Tagged images can only be used manually or by adjusting the original initialization script, for which these instructions are provided.

```
...
After cloning the repo, start the initiation script.
There are two scripts, one for both Windows and Linux OS.
Use the one you are currently working on.

The script will ask which SDK to use,
if you wish to save your container as image,
path to your mount folder, and volume mount name.
Answer accordingly:

$cd sdk-docker/docker_SDK/

$./linux_init_script.sh OR .\windows_init_script.bat

Use Target or Native SDK (T/N): n

Want to keep the built and modified Docker image for latter use?
If yes, give tag for it. If not, press return. : <tag>  #for example ylilantt_image

Give path to mount directory or press return if mount not needed:
<path to mount folder> #for example /var/fpwork/ylilantt/mount_folder

Give volume name or press return if mount not needed.
Volume can be existing one or new. In latter case it will be created. :
<volume name> #for example ylilantt_volume
...
```

Figure 12. Step-by-step instruction on how to setup the container

In the final part of the documentation commonly used Docker commands are introduced. The exact commands are provided with possible examples with explanations what they do and in what situations they might prove useful. In addition, a link to the Docker's documentation is given. Developers who only use the provided initialization script might not find this section as useful as the developers who begin to experiment with Docker. The main purpose of the section is to provide developers with general instructions for solving trivial problem situations. This should help to reduce the amount of unnecessary issue ticketing and contacting, which in turn would hamper the implementation process.

5. RESULTS AND DISCUSSION

To help evaluate the Docker container development environment implementation, data is gathered to compare SoC SW department's current state with the old one. Measurements are made of how long time it takes to setup the container environment in different scenarios. The results are proportioned with the survey feedback examined in the section 3.2. In addition, it is reviewed whether the implementation has met the expectations of the mentioned survey. The new survey that was held for the departments developers who have used the project's environment is discussed, and their satisfaction and opinions are measured. At the end of this section, the project's items those require development and future development plans are discussed.

5.1 Setting up the container environment

The container environment has currently three different initial scenarios worth measuring: first setup without cache or Docker image, setup from ready Docker image, and setup from tagged image. The developers have to setup the container environment a few times a day if they work on multiple different projects and disconnect from the containers at the end of the day. As stated earlier, Windows uses Linux VM for running the container, leaving its efficiency to VM level. Therefore, potential performance differences between Linux and Windows setup process are compared.

Tasks those are more or less constant regardless of the user are collected in Table 5. Outside of Table 5, the first step in the whole setup process is to get to the documentation page and read through necessary parts. The time it takes for an individual developer to read the documentation is not measurable, but it can be stated, that the documentation is reasonably short, and it takes less than five minutes to read it.

Table 5. Time comparison between different container setups

Windows (W) and Linux (L) scenarios	**User input**	**Image and container build**	**SDK setup**	**Clean up**	**Total**
L First setup	60 s	210 s	60 s	5 s	335 s
L Image setup	60 s	10 s	60 s	5 s	135 s
L Tag setup	200 s	0 s	5 s	5 s	210 s
W First setup	60 s	240 s	150 s	10 s	460 s
W Image setup	60 s	10 s	150 s	10 s	230 s
W Tag setup	200 s	0 s	5 s	10 s	215 s

First setup describes a scenario, where the user runs the initialization script for the first time, without having built the base image from which the image is inherited. This case occurs only in developers' personal environment during the first build, or when the images are updated and need to be removed. In image setup, the user has already prebuilt the image the container environment is using. This is the most common situation and the default state from where the script is run. The last case is tag setup, where the user creates an image from a container instance and uses it to setup the container environment. The same scenarios are presented for both Linux and Windows.

The columns show the time it took to execute particular task. User input includes time estimation of code fetching and user's terminal input to the point when the initialization script starts running or the user is in the container. Image and container build indicate the time it takes to build the image, create the container and get user inside it. SDK setup holds within the SDK fetch, execution and environment variable setup. Lastly in the clean up, the time it takes from the user to leave the container once they perform exit command is measured, and total shows the sum of all operations.

The test case scenarios in Table 5 were reproduced five times to get more realistic time estimates, but there still are varying time influencing factors, such as machine load and

internet's download speed. In tag setup the user input time is much higher than in other scenarios, as the container have to be manually launched from the new tagged image. The portion of this step is significantly reduced after first setup and can be further reduced by creating a script to do the setup. The docker commit command is relatively slow, approximately 120 seconds, and it has to be done for each individual project separately, as the SDK has already been set in the environment. From the image and container build times it can be noticed, that the creation and transition of the container does not take long, and the building of the image is the heavy part. In tagged case this takes no time at all, as the container is created and entered in the manual stages which is taken into account in the previous step. As the image can be reused as many times as desired, this step is nearly instant in long term use. In the SDK setup the first prominent difference between Windows and Linux OS can be seen. However, this does not cause significant inconvenience, as the slowest case on Windows still took less than 8 minutes to perform. For tagged images, the SDK has already been downloaded and unpacked, leaving only the need to source the environment variables.

From Table 5 results it can be observed that the fastest consistent container setup takes around 2 minutes to perform. This execution time can still be significantly lowered by optimizing the tagged image setup. The user input time can be under 10 seconds, meaning the whole setup process will take under 20 seconds. Such a short initialization time improves the user experience, as it does not leave the developer waiting.

Overall, the results are promising, and the improvement compared to the previous situation is considerable. There is no specific data of the previous environment setup durations, but user recollection of it has been collected in the section 3.2. The developers reported that successful setup for one project took around 1 working day, and the worst-case scenarios up to 2 weeks. With the assumption no problems occur during either case, the development environment setup time was reduced from 7,5 hours to few minutes.

With these estimates, a rough financial benefit of the implementation can be calculated. A situation is inspected, where each of SoC SW department's developer builds a new project environment. As SoC SW department consists of approximately 200 developers, and the individual build time is rounded to save 7 hours, already 1400 work hours has been saved. To get a more realistic picture of the benefits, this result is compared with the development process of the implementation. If the background studies and the familiarization with the subject is left out, and only count the design, development, test and fix

parts of the project work, the completion took approximately two working months, meaning 330 working hours. Considering the time spent on the implementation phase, still over 1000 work hours were saved after one department wide project environment setup.

5.2 User feedback

To get an even better understanding of how well the implementation succeeded, a feedback survey was sent for the SoC SW department. This time only developers who have used the container environment can take the survey, reducing the amount of possible answers a bit. Aware of this, the implementation is introduced as the survey is sent and developers are encouraged to try it. The survey's agenda is to get improvement ideas, collect information on malfunctions as well as bugs, and to obtain general opinion on the functioning of the environment. In addition, the background knowledge of the developer is asked, as it is good to be taken into account in other answers.

As expected, so far only five answers have been received to the survey, which is not an issue as the questions are formed to be more of an individual entities. As more users begin to use the environment, response and feedback rates are expected to rise. The survey had six questions and one follow-up question depending on the answer given in question number 3. Again, the survey was held anonymously to get honest feedback, and the survey's original frame can be seen in Appendix B. Next, the thoughts behind each separate question in the survey are inspected, three first questions were multiple choices and rest were open ended.

1. Did you have previous experience with Docker?

   As the survey is held anonymously, at first a better view of the user's knowledge on the topic is acquired. This helps to reflect their answers given later and observe if novice or experienced users struggle with certain parts. Developers who have used Docker a lot may are expected to have stronger opinions on technical solutions, for they presumably have their own way of doing things.

2. How did you consider the first time setup/deployment?

   User experience on what kind of first impression the container setup gives is gathered. The environment setup is supposed to be simple and straightforward, and this question measures how well the implementation has succeeded in it. If the first use of a new method and technology is faltering, the reception will not be favourable.

3. Did you find the documentation clear and informative?

The documentation page is in its early stages and not yet well formatted, and developers are asked if the preliminary solution is going to the right direction. The necessary information should be in plain sight and essentials easy to find from the page. If the developer found the documentation to be incomplete, a follow up question with open ended feedback arises.

4. Pros and cons of the container environment.

   This question helps to evaluate which features are good and which need further improvement from developers' aspect. However, the more important area of focus is cons, as the well-functioning features tend to be left overlooked.

5. Improvement suggestions (setup, usage, documentation etc).

   The most important issue in terms of the project status, since the environment is up and running, further development concepts are needed. The purpose is to highlight critical matters from the previous question, and to possibly get concrete ways to approach named problems.

6. Open feedback

   Finally, opinions are gathered to see overall satisfaction on the new build environment. Here the developers can also give criticism and further suggestion on topics they did not find fit in the previous questions.

The survey's first couple of questions did not have individually any distinctive answers, as their intention were to gather background information to help analyse the following questions. However, there is a wide variation in responses, meaning information is received from different set of developers with different user experiences. In one developer's opinion, the documentation did not meet good quality. The reasoning for this was the lack of explanation behind the commands. The developer wished to have more information what the commands are actually doing and what for they are executed. The small details were consciously left out, as they could flood the documentation with a wall of text and make it unnecessary hard to read. A separate section in the documentation can be created for more detailed explanations, or if more of similar feedback is received, it can be added to the main section.

The common tone of the advantages in the new container environment were focused on effortless setup and being always identical. However, in the next few questions main focus is on improvement suggestions. As a bit of a surprise, there was a desire from developers to receive documentation instructions on how to compile a driver or application in the new environment. The compiling process is the same as before, but as a

solution a link is put into the instruction page on the documentation. Other improvement topic from Windows users was to get a more comprehensive mounting during initialization. Adding such functionality to the initialization script would unnecessarily complicate matters for users who do not need it. The intention was to provide the developers with enough material to allow them to personally customize the script to fit their needs, however without modifying the Dockerfiles. The last development proposal to consider was to move the image tagging to the beginning of the script, and make it so it will not launch the container. This would prevent an unstable situation where the developer closes the container session without manually exiting, for example by shutting down the computer, in which case the image commit is left in uncertain state. Making this change seems sensible and doing it is effortless. In addition, one bug report was given, caused by mounting same directory into multiple containers. There has been no time to investigate this further yet, but one possible solution is to change the mount and use volumes instead. Volumes have a Docker command

–volumes-from

which will mount the named volume from other target container to the new one during launch.

The goal is to take all feedback into account, and the survey will be kept open to get further feedback from the new users. The challenge is to find a solution to satisfy as many developers as possible, while simultaneously to keep the implementation simple and user friendly. While adjusting and developing the environment, it is important to remember to take into account that most of the developers have no previous experience with Docker, and some might even lack in UNIX skills. For these reasons, the maintenance and clarity of documentation is the key for well-functioning implementation.

5.3 Future development

The container development environment is used by SoC SW developers with different backgrounds, skills and intentions. For this reason, currently developers are requesting different implementation models for the same features. The goal is to gather as much feedback as possible, and to make the environment and its setup process simple and satisfactory for as many developers as possible. It is expected, that in the near future new small functionalities are added and improvement of old ones is done based on the feedback received. The project’s maintenance should not be too laborious, even when new products need to be implemented. SoC SW’s technical leader assigns a developer to be responsible of the project’s maintenance.

Currently the future implementation focus regarding the container environment is CI. The current Jenkins jobs do not cover all use cases and their usage can be unwieldy. All unambiguous and laborious tasks could be automated, to save developers' time and remove unnecessary repetition. In addition, the jobs can be changed to trigger from code commits to VCS, and if wanted, all available products can be tested from one commit. At this time the test results are shown in Jenkins application, which can be changed to better notification solution, such as automated email.

To further support the CI implementation, Kubernetes could be taken into use, to automatically setup containers from the Docker image. Kubernetes is used for scheduling and launching containers, and it can manage them in large scale. Kubernetes functions well with containers those are used for small number of actions, such as automated testing with CI pipeline [33].

6. CONCLUSIONS

Over the years, software development has become an increasingly complex entity. As the project grows in size, it becomes too laborious for a developer team to perform the whole coding process on its own. The developers have to take advantage of old projects, libraries, related work and so forth. These allow developers to perform tasks with significantly increased pace, while simultaneously causing difficulties. Now developers have to use a large portion of their time to set up correct development environment, including operating system, dependencies, code fetching and setting correct tools.

The main goal of this thesis work was to implement a uniform development environment for SoC SW department, with approximately 200 developers. The development environment has to be reproducible and the configuration must be able to be done for different products. A case study was conducted to evaluate the best technology to suit the need, and Docker containers raised as the best solution, for one reason Docker has been used by different departments in the company.

The purpose of the project is to solve the current problem cases in the SoC SW department regarding build coherency. The developers work on multiple different products and each product requires unique development environment. These development environment requirements mainly consist of packages, dependencies, tools and environment variables. Setting up such a complex environment takes a lot of time from the developer, is error prone and overall tedious. However, the main challenge is the compatibility issues, as the developers' environments may wary by tool versions or other small factors those are hard to detect. In addition, the complicated environment setup lacks documentation, and since there are multiple different ways to do the setup, it would be overwhelming task to update and keep track of them all.

The solution model starts with a Docker image, which is built from suitable Dockerfiles. The Dockerfiles contain all required tools and other base requirements, and they are stored in repositories for developers to openly access. The development environment is built from the Docker image, but as the setup process has several steps and can provide difficulty for less experienced developers, an initialization script is provided. The intention is to make the script simple and user friendly, with comprehensive documentation. Once the developer is in the container, selected product specific SDK is automatically fetched and set to finish the environment setup.

The goal of this thesis project was achieved, and the results were good. A reproducible development environment with specific requirements was created, which can be launched from different OSs. According to the information received, the initial development environment setup time was reduced from one or more working days to mere minutes. A feedback survey was conducted after the project was completed, to gather information regarding user experiences. Developers were mainly satisfied with the ease of use, and some areas for further development and bugs were reported.

REFERENCES

[1] M. Soni, I. Das, R. Youe, J. Dharmaraj, K. McGowan - Jenkins essentials: continuous integration, setting up the stage for a DevOps culture, Packt Publishing, 2015. Available: https://learning.oreilly.com/library/view/jenkins-essentials/9781783553471/toc.html

[2] G. Schenker, H. Saita, H. Lee, K. Hsu - Getting Started with Containerization, Packt Publishing, March 2019. Available: https://learning.oreilly.com/library/view/getting-started-with/9781838645700/?ar

[3] R. Rajsuman, H. Ling - System-on-a-chip: Design And Test, July 2000. pp. 3-7 Available: https://ebookcentral.proquest.com/lib/tampere/detail.action?docID=257584&pq-origsite=primo

[4] S. Roopak, R. Parthasarathi, B. Samik - Correct-by-Construction Approaches for SoC Design, Springer, New York, 2014. pp. 1-10 Available: https://www.springer.com/gp/book/9781461478638

[5] G. Tsihrintzis, M. Virvou, L. Jain - Multimedia Services in Intelligent Environments, Springer, Berlin, 2010. pp. 1-25 Available: https://www.springer.com/gp/book/9783642133541

[6] GCC, the GNU Compiler Collection, Visited on July 2020. Available: https://gcc.gnu.org/

[7] R. Oshana - Software Engineering for Embedded Systems, Newnes, April 2013. Available: https://learning.oreilly.com/library/view/software-engineering-for/9780124159174/?ar

[8] Git, Visited on July 2020. Available: https://git-scm.com/

[9] S. Ritchie - Pro .NET Best Practices, Apress, December 2011. Available: https://learning.oreilly.com/library/view/pro-net-best/9781430240235/?ar

[10] S. Goodman, D. Fanelli, J. Ioannidis - What does research reproducibility mean? June 2016. Vol. 8 Available: https://stm.sciencemag.org/content/8/341/341ps12

[11] M. Meyer - Continuous Integration and Its Tools, IEEE, May 2014. pp. 14-16 Available: https://ieeexplore.ieee.org/document/6802994?ALU=LU1050740

[12] P. Duvall, S. Matyas, A. Glover - Continuous Integration: Improving Software Quality and Reducing Risk, Addison-Wesley Professional, June 2007. Available: https://learning.oreilly.com/library/view/continuous-integration-improving/9780321336385/?ar

[13] M. Pearce, S. Zeadally, R. Hunt - Virtualization: Issues, security threats, and solutions, ACM Computing Surveys, February 2013. Available: https://dl.acm.org/doi/10.1145/2431211.2431216

[14] R. Morabito, J. Kjällman, M. Komu - Hypervisors vs. Lightweight Virtualization: A Performance Comparison, IEEE, March 2015. Available: https://ieeexplore.ieee.org/document/7092949

[15] H. Saito, H. Lee, C. Wu - DevOps with Kubernetes, Packt Publishing, October 2017. Available: https://learning.oreilly.com/library/view/devops-with-kubernetes/9781788396646/?ar

[16] P. Raj, J. Chelladhurai, V. Singh, S. Holla, O. Hane, N. Khare, R. Dua, V. Kohli, S. Konduri, R. McKendrick, A. Espinosa, S. Gallagher - Docker: Creating Structured Containers, Packt Publishing, June 2016. Available: https://learning.oreilly.com/library/view/docker-creating-structured/9781786465931/?ar

[17] Docker Docs, Visited on July 2020. Available: https://docs.docker.com/docker-hub/repos/

[18] O. Salvador, D. Angolini - Embedded Linux Development using Yocto Projects - Second Edition, Packt Publishing, November 2017. Available: https://learning.oreilly.com/library/view/embedded-linux-development/9781788470469/?ar

[19] A. Vaduva - Learning Embedded Linux Using the Yocto Project, Packt Publishing, June 2015. Available: https://learning.oreilly.com/library/view/learning-embedded-linux/9781784397395/?ar

[20] OpenEmbedded-Core, Visited on July 2020. Available: http://www.openembedded.org/wiki/OpenEmbedded-Core

[21] BitBake User Manual, Visited on July 2020. Available: https://www.yoctoproject.org/docs/current/bitbake-user-manual/bitbake-user-manual.html#bitbake-user-manual-intro

[22] Yocto Project Application Development and the Extensible Software Development Kit (eSDK), Visited on July 2020. Available: https://www.yoctoproject.org/docs/3.1.1/sdk-manual/sdk-manual.html#sdk-development-model

[23] J. Turnbull – The Docker Book, Turnbull Press, July 2014. Available: https://learning.oreilly.com/library/view/the-docker-book/9780988820203/?ar

[24] K. Sean, M. Karl – Docker: Up & Running, O'Reilly Media, Inc. September 2018. Available: https://learning.oreilly.com/library/view/docker-up/9781492036722/

[25] Using Docker Containers As Development Machines, Visited on July 2020. Available: https://medium.com/rate-engineering/using-docker-containers-as-development-machines-4de8199fc662

[26] Creating a Consistent Cross-platform Docker Development Environment, Visited on July 2020. Available: https://rollout.io/blog/cross-platform-docker-development-environment/

[27] K. Devisetty, K. Kennedy, P. Sarando, N. Merchant, E. Lyons – Bringing your tools to CyVerse Discovery Environment using Docker, Faculty of 1000 Ltd., 2016. Available: https://f1000research.com/articles/5-1442/v3

[28] K. Senthil – Practical LXC and LXD: Linux Containers for Virtualization and Orchestration, Apress, August 2017. Available: https://learning.oreilly.com/library/view/practical-lxc-and/9781484230244/

[29] A. O'Grady - Gitlab Quick Start Guide, Packt Publishing, November 2018. Available: https://learning.oreilly.com/library/view/gitlab-quick-start/9781789534344/?ar

[30] L. Milanesio - Learning Gerrit Code Review, Packt Publishing, September 2013. Available: https://learning.oreilly.com/library/view/learning-gerrit-code/9781783289479/?ar

[31] M. Soni - Jenkins Essentials, Packt Publishing, July 2015. Available: https://learning.oreilly.com/library/view/jenkins-essentials/9781783553471/?ar

[32] Zuul Concepts, Visited on August 2020. Available: https://zuul-ci.org/docs/zuul/discussion/concepts.html

[33] D. Rensin – Kubernetes, O'Reilly Media, Inc., September 2015. Available: https://learning.oreilly.com/library/view/kubernetes/9781492048718/?ar

APPENDIX

APPENDIX A: First Survey

1. What build environment have you been using (LinSEE server, oracle VM, container, other -> what)?
2. Pros and cons of your current build environment.
3. Have you manually set-up the build environment (compilers, host tools, etc)? If yes, give a time estimation how long this took.
4. How the current build environment could be improved?
5. Do you have any earlier experience with Docker?
6. What expectations you have from the new Docker development environment?

APPENDIX B: Second Survey

1. Did you have previous experience with Docker?
2. How did you consider the first time setup/deployment?
3. Did you find the documentation clear and informative?
4. What was unsatisfactory in the documentation.
5. Pros and cons of the container environment.
6. Improvement suggestions (setup, usage, documentation etc).
7. Open feedback