Tampere University

Olli Kanerva

DEPLOYMENT OF PROPRIETARY APPLICATIONS BASED ON OPEN SOURCE TECHNOLOGY FOR AUTOMATION PLATFORM

Master of Science Thesis
Faculty of Engineering and Natural Sciences
Examiner: Professor Jose Luis Martinez Lastra
June 2022

ABSTRACT

Olli Kanerva: Deployment of proprietary applications based on open source technology for automation platform
Master of Science Thesis
Tampere University
Master's Programme in Automation Engineering
June 2022

Industrial automation is ever-evolving field with latest technologies and devices. The importance of software is crucial in industrial settings and the latest trends are heading towards the implementation of software that is based on open source technologies. Bosch Rexroth has developed the latest platform that has implemented features that allow anyone to develop applications for their new ctrlX AUTOMATION devices. Cimcorp has had years of cooperation with Bosch Rexroth and is therefore implementing the new generation automation platform into their device portfolio in coming years.

This master's thesis was done for Cimcorp Oy and it examines how applications, that are created with open source technologies, are deployed in proprietary matter in industrial settings. Thesis work focuses on the development environments of the applications and also the actual creation of the application. Everything was done in virtual environments as the new automation platform also supports full virtualization of actual hardware devices. Work also includes a theory-based review of modern software development practices and environments related to that.

The results of this thesis included an sample application created with two programming languages, C++ and Python. Python version was created based on feedback of the C++ version so it contains slightly more wanted features. Best practices of developing applications for ctrlX were researched and tested. At the end of this thesis also the future of the created sample application in actual physical devices was discussed.

Keywords: ctrlX, automation, industrial software, robotics, open source technology

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

TIIVISTELMÄ

Olli Kanerva: Avoimen lähdekoodin teknologiaan perustuvien patentoitujen sovellusten käyttöönotto ja tutkiminen uudelle automaatioalustalle
Diplomityö
Tampereen yliopisto
Automaatiotekniikan maisterikoulutus
Kesäkuu 2022

Teollisuusautomaatio on jatkuvasti kehittyvä ala, jossa käytetään uusimpia teknologioita sekä laitteita. Ohjelmistojen merkitys on keskeinen teollisissa ympäristöissä ja viimeisimmät trendit ovat menossa kohti avoimen lähdekoodin teknologioihin perustuvien ohjelmistojen käyttöönottoa. Bosch Rexroth on kehittänyt automaatioalustan, joka sisältää ominaisuuksia, joiden avulla kuka tahansa voi kehittää sovelluksia uusille ctrlX AUTOMATION-laitteilleen. Cimcorp on tehnyt vuosia yhteistyötä Bosch Rexrothin kanssa, ja siksi se ottaa uuden sukupolven automaatioalustan osaksi laitevalikoimaansa tulevina vuosina.

Tämä Cimcorp Oy:lle tehty diplomityö tutkii, miten avoimen lähdekoodin teknologioilla luotuja sovelluksia otetaan käyttöön teollisissa ympäristöissä. Työssä keskitytään sovellusten kehitysympäristöihin ja myös varsinaiseen sovelluksen luomiseen. Kaikki tehtiin virtuaaliympäristöissä, sillä uusi automaatioalusta tukee myös todellisten laitteistojen täyttä virtualisointia. Työ sisältää myös teoriapohjaisen katsauksen nykyaikaisiin ohjelmistokehityksen käytäntöihin ja niihin liittyviin ympäristöihin.

Tämän diplomityön tuloksiin sisältyi esimerkkisovellus, joka on luotu kahdella ohjelmointikielellä, C++:lla ja Python:lla. Python-versio luotiin C++-version palautteen perusteella, joten se sisältää hieman enemmän haluttuja ominaisuuksia. Parhaita käytäntöjä ctrlX-sovellusten kehittämiseen tutkittiin ja testattiin. Tämän työn lopussa käsiteltiin myös luodun mallisovelluksen tulevaisuutta fyysisissä laitteissa ja sovelluksissa.

Avainsanat: ctrlX, automaatio, teolliset ohjelmistot, robotiikka, avoimen lähdekoodin teknologiat

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

PREFACE

My journey with Cimcorp started in 2019 when I got selected as a trainee for the final year internship at Satakunta University of Applied Sciences. At the end of 2019 I did my bachelor's thesis for Cimcorp Oy so I want to give my gratitude for them for giving me the opportunity to also do my master's thesis along side of my regular daily work.

Especially I want to thank Ari Stjerna for guiding me through the whole process and all the enthusiastic explanations of ideas that were behind the development work of ctrlX integration. I also want to thank Jani Tuomola for giving feedback on the progress along the way and of course the university instructor Luis Gonzalez Moctezuma for providing clear path on how the structure of thesis would look like.

Tampereella, 4th June 2022

Olli Kanerva

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

AI	Artificial Intelligence
API	Application Programming Interface
CPU	Central Processing Unit
CtrlX	New generation automation platform
DevOps	Set of practices that combines software development and IT operations
ERP	Enterprise Resource Planning
HMI	Human Machine Interface
IDE	Integrated Development Environment
IEC 61131-3	Third part of the industrial standard which specifies syntax and semantics of programming languages for programmable controllers
MES	Manufacturing Execution System
MLD	Bosch Rexroth's motion control system
OPC Unified Architecture	Platform-independent data transfer mechanism for moving data between servers and clients
OS	Operating system
PLC	Programmable Logic Controller
Snap	Software packaging and deployment system
VCS	Version Control System
WCS	Warehouse Control System developed by Cimcorp Group

1. INTRODUCTION

1.1 Background

Software has become a part of modern industrial systems and plays a key part in financial strategy of many companies so the importance of good, quality software is crucial. This leads to challenges where the complexity of the software is increasing and the different levels in ISA-95 "automation pyramid" (Figure 1.1) are becoming depended from each other. "Automation pyramid" is representing the current understanding of IT architecture in industrial settings. This model divides the manufacturing operations into different levels where each level represents a certain system, information or timeframe. Model has been defined in a standard model by International Society of Automation (ISA). [1]

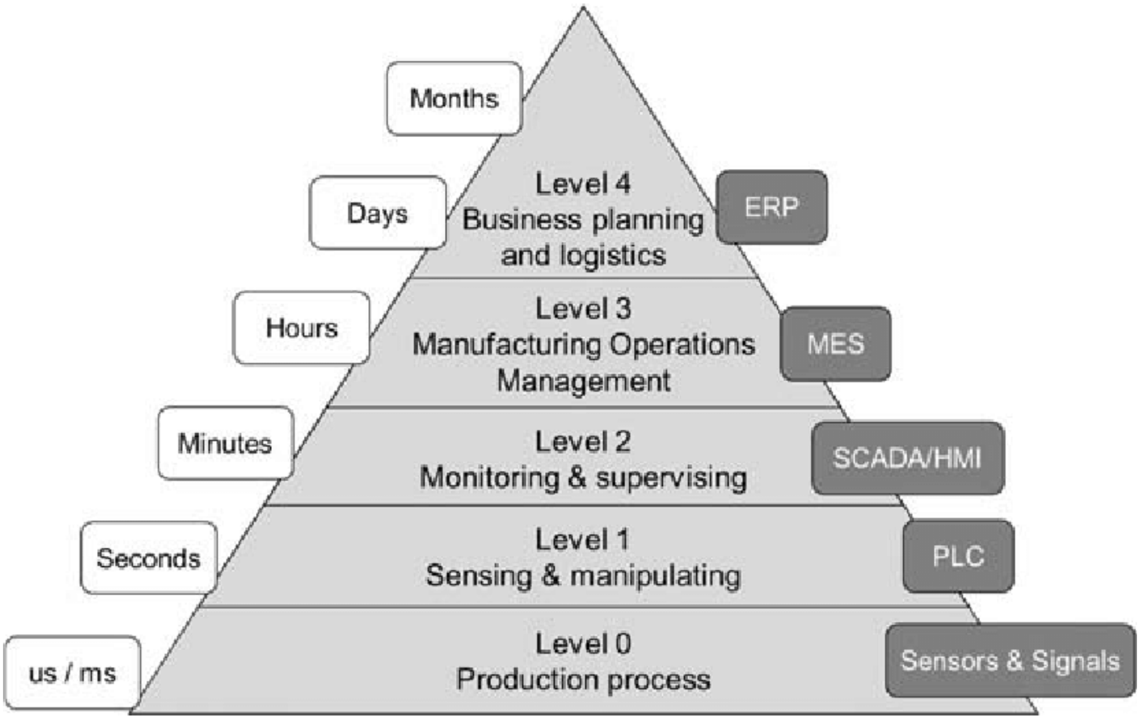

Figure 1.1. *ISA-95 automation pyramid model [1]*

For instance, in Bosch Rexroth's idea of future factories the levels are all connected to each other very closely (Figure 1.2). Data collected remotely from sensors located in the plant floor could effect the decision made in enterprise resource planning (ERP) system or tell the manufacturing execution system (MES) that the quality of the product is not good enough etc. Different kind of models, machine learning or AI can all be applied to the data and then be used in management level decisions.

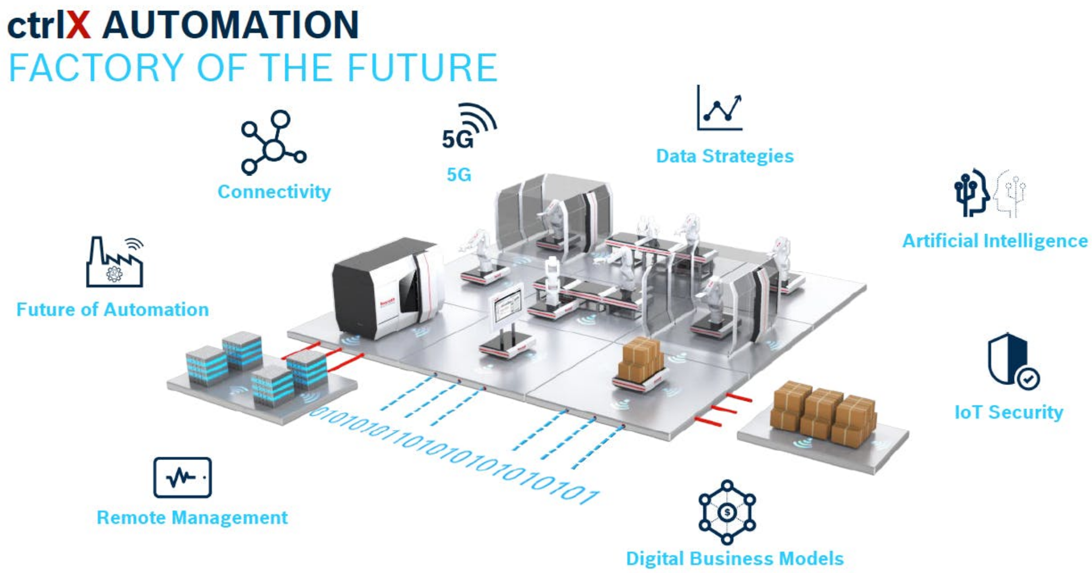

Figure 1.2. *Factory of the future concept idea by Bosch Rexroth [19]*

Cimcorp's and Bosch Rexroth's history together as business partners goes back for over 10 years. Bosch Rexroth has been the main supplier of the Cimcorp's control systems, more specifically delivering the MLD, drive-integrated motion control with PLC functions, and the proprietary software framework in Cimcorp's gantry robot solutions for a few generations. These two companies have had a deep integration of both software and hardware in lower level devices to achieve efficient system that can be used in higher level applications also provided by Cimcorp. Now when Bosch Rexroth started to develop the ctrlX automation platform, Cimcorp has been on board with development the entire time. [7]

As a lead customer, Cimcorp has a opportunity to influence the development of the ctrlX platform and a chance to develop their own systems and its integration to the ctrlX world at the same time. ctrlX integration has major potential for Cimcorp because the hardware size is much smaller, weights less and therefore helps Cimcorp's gantry robots to achieve faster cycle times. Software wise the transparency of open source code and the possibility of creating proprietary applications for specific needs is very intriguing for companies, such as Cimcorp, to further develop their product portfolio and deliver better products for customers. [19]

1.2 Problem definition

Software development and pipelines in Cimcorp Group are implemented roughly in two categories depending on whether the use of the software is in field level PLC devices or in the higher level Warehouse Control System (WCS). Development pipeline for both of these categories are slightly different because of the differences in programming languages and the supported environments. This thesis won't be focusing on the WCS side of software development or methods that much, but rather on the opportunities that the new generation devices and third-party software can produce with field level devices such as PLC and servo applications.

Bitbucket version control system, Docker, open source containerization platform and Jenkins, automation server to help on software development, are commonly used applications for WCS development in Cimcorp, so the open source technology isn't anything new for industrial cases. However developing and deploying snaps for industrial settings is not that well known area in house because before ctrlX automation platform there has not been a need for it. This is, because there has not been support for such third party applications in PLC devices. In short, snaps are application containers for mainly desktop, cloud or IoT use for operating systems that are running Linux kernel. For instance, Spotify, well known music streaming service, is packaged with all of its dependencies to one single installable snap so that it can be installed to many different Linux distributions via Snap store and thus is called a snap.[31]

Snaps can be developed with basically any programming language that there is. However, in the SDK, or the Software Development Kit, that is used to develop the snaps for the ctrlX platform there aren't as many supported, but still quite a few. Main one's that at least Bosch Rexroth is suggesting for most efficient use are C++ and Python programming languages.[11] Expertise in Cimcorp is in Java language because the WCS is created with it, so this brings us to the question, what are the pros and cons of different programming languages for snaps in ctrlX environment and how do they compare to each other. Is it possible for example to implement the snaps in Java, C++ or Python completely or is it necessary to make some hybrid version of different languages?

Snap interfaces between the ctrlX CORE, WCS and external systems, customers ERP systems for instance, is particularly interesting problem to wonder. Is it possible to find such interface module that all snaps developed by Cimcorp, no matter what they do, can be configured using the same interface standard so that they would operate on the same principle?

1.3 Objectives

The main focus in this thesis is on snap technology and on snap development with integration to ctrlX automation. Objective is to find best methods and workflow for snap development. Different interfaces to and from snaps are also particularly interesting for this thesis and will have a lot of focus on it. Research work on this thesis aims to provide an answer to these following research questions which are also defining the scope of the work:

1. What would be the standardized application interface when considering the usage in industrial automation domain?
2. How to use database functionality with proprietary snap application and how well does it perform in CtrlX environment?
3. What is the most convenient method for building a production version of a proprietary snap application?

1.4 Outline

This thesis is divided into 6 chapters; chapter 1 is the introduction chapter. It describes the document structure, establishes the problem and gives the objectives for the work. Chapter 2 provides the theoretical background and the state of the art concepts that are used and researched later in the thesis. Chapter 3 presents the proposed practices and methods that might help to achieve the objectives described in the chapter 1. Chapter 4 is the implementation chapter where the proposed approach described in the chapter 3 is implemented. Chapter 5 presents the results and discussions of the tests done during the implementation chapter to prove the competence of the proposed practises. Lastly in chapter 6, the conclusions of the thesis work are carried out with some discussions on future development that can be applied based on this thesis.

2. STATE OF THE ART

Nowadays, industrial production is moving towards flexible factories with short product life cycles and a need of rapid reactions to changes. This is why the rigid and complex automation production lines are in the past and open platforms with third-party hardware and software are coming in to play. This chapter of the thesis examines the current trends and the state of the art of industrial automation and tells more about the CtrlX AUTOMATION product family developed by Bosch Rexroth. [10] This chapter also studies current trends in software development and other related parts of the development pipeline that are used world wide in software development in industrial settings.

2.1 ctrlX AUTOMATION family

2.1.1 Hardware components

The heart of the operation in the hardware side of ctrlX AUTOMATION is the ctrlX CORE (Figure 2.1) which is powering the control systems as a whole. ctrlX CORE is Linux based control platform and has 64 bit quad-core ARM processor as a brain. ctrlX CORE can also be directly integrated to the servo drives which are called ctrlX DRIVES (Figure 2.1). ctrlX DRIVE is a drive system that consists of servo drives and servo motors. Servo drives are scalable and thus can be linked together to achieve more degrees of freedom depending on the application that it is used for. Servo drives are the powering factor of servo motors which generate the actual movement for the robot etc. These hardware components and their integration together are particularly interesting for Cimcorp because the physical size of the drives and CPU compared to the older generation is 50% smaller and for the servo motors 30% smaller. For instance, this reduction of physical sizes will decrease the weight of the Cimcorp's gantry robot solutions and thus the robot can be much more efficient regarding the speed and pacing of its tasks. [9]

The ctrlX AUTOMATION family also has more hardware components. The ctrlX HMI which are different size multi-touch displays for mobile or stationary use. The ctrlX IPC or the industrial-PC's are there to form the interface between the field level and the control level to ensure the reliable and secure transmission of large volumes of data. ctrlX IPC's come in different sizes and configurations depending on the needs of the user. IPC's are also used in Cimcorp to connect and control the gantry robot cells where the IPC receives and sends data to the Cimcorp warehouse control system (WCS) and also works as a instruction logger. These ctrlX IPC's are a solid alternative for Cimcorp to update and "future proof" their systems. [9]

The last two hardware components that the ctrlX family has are the ctrlX I/O and the ctrlX SAFETY (Figure 2.1). CtrlX I/O has the classic analog and digital input/output capabilities as well as an additional EtherCAT add-on to connect the whole ecosystem to the machine control level, for instance to the servo controls. The I/O has a ability to be directly attached to the ctrlX CORE and then work as a EtherCAT master. The ctrlX SAFETY on the other hand consist of SafeLogic and SafeMotion devices. SafeLogic is purely a safety control system while the SafeMotion is the safety drive system based on the ctrlX DRIVE. Bosch Rexroth promises 5 to 10 times faster communication on these SAFETY devices if they are connected together with ctrlX SAFETY$^{\text{link}}$ interface as a complete system compared to the classic fieldbuses.[9]

Figure 2.1. *ctrlX CORE, SAFETY and DRIVE [9]*

2.1.2 ctrlX software

Software side of ctrlX AUTOMATION family consists of four main software frameworks. These are called ctrlX WORKS, MOTION, PLC and IOT. ctrlX WORKS is toolbox for engineering of different automation solutions. From ctrlX WORKS, it is possible to gain access to all the necessary tools for industrial solutions. ctrlX PLC engineering can be opened from the side menu and straight go into developing PLC applications in intended environments or commission servo drives and motors with ctrlX DRIVES engineering. Although, regarding this thesis, the most useful part of the ctrlX WORKS is its ability to engineer automation solutions through virtual environment without any real hardware connected to the PC, with ctrlX CORE^{virtual}. The ctrlX CORE^{virtual} is promised to be identical software wise to the actual physical hardware and thus allow testing and commissioning of applications without actual control hardware. This concept of the so-called Digital twins, meaning virtual representation of objects or systems that is based on real-time data, is very beneficial when considering the efficiency of commissioning automation applications. Especially at times like these, where the component shortage is limiting the availability of actual hardware.[15]

ctrlX MOTION application is running on ctrlX CORE and therefore also on the ctrlX CORE^{virtual} and its purpose is to provide the motion functions for applications where there are one or more axes present. These application can be for instance Cartesian gantry robot solutions that Cimcorp provides. ctrlX MOTION can also be run controller-based or drive-based which means that it can either be in ctrlX DRIVE with integrated CORE or embedded to normal ctrlX CORE. [14] Figure 2.2 shows an example of how controller-based architecture with different ctrlX AUTOMATION hardware and software could look like.

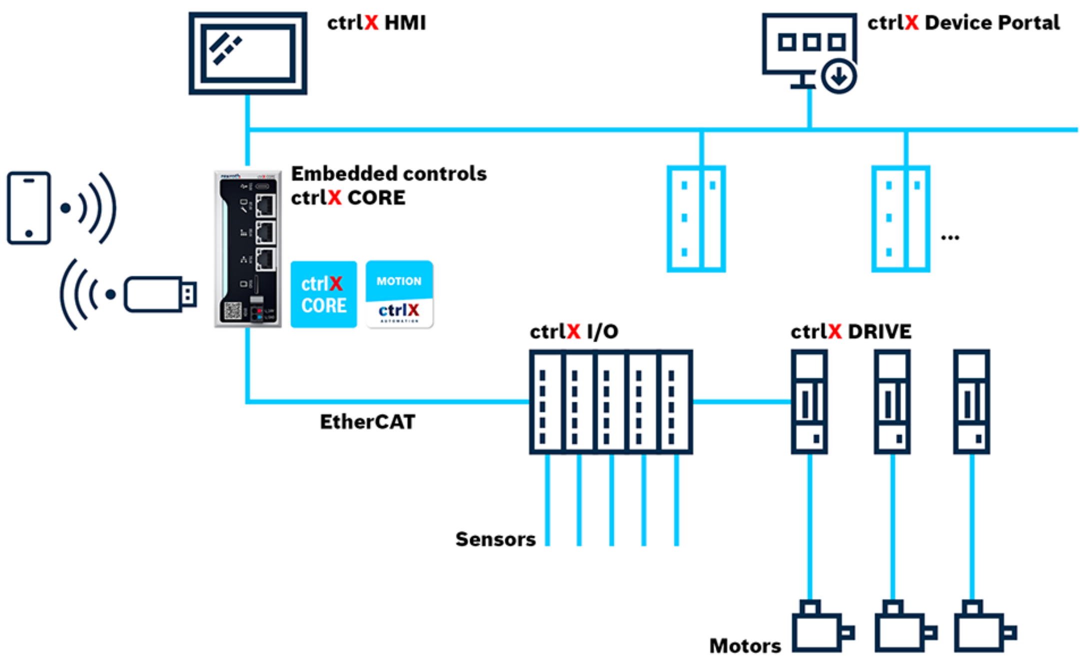

Figure 2.2. *ctrlX MOTION architecture example [14]*

ctrlX PLC is an application installed also to the ctrlX CORE for sole purpose of developing PLC applications for different purposes. Types of automation that it can be used for are general industrial process or assembly automation solutions or regular building automation to control lighting or air conditioning etc. IOT application on the other hand offers connection options and standards to different IOT devices via internet. [9]

Data Layer is also very important aspect of the whole ctrlX AUTOMATION family. Although it is not an installable application like the rest of the ctrlX softwares, it is a default feature in ctrlX CORE. Data Layer is a data broker of the ctrlX CORE, which means that every application installed to the CORE can have managed access to the real time or non-real time data available on the controller. Later in this thesis, Data Layer word will come out more because every interface or data exchange between CORE applications inside of the controller or from the outside, like from OPC UA, will have to go through the ctrlX Data Layer. [13] Also regarding the thesis, ctrlX Software Development Kit (SDK) is important concept because the example applications developed for this thesis is based on examples and codes from the SDK. SDK in short is software package delivered by Bosch Rexroth to help developers to create their own applications (or snaps) for the ctrlX CORE. More precise examples of SDK are provided later in the thesis. [11]

To summarize the software side of ctrlX AUTOMATION; MOTION, PLC and IOT are applications installed to the ctrlX CORE, CtrlX WORKS is a toolbox for engineering with real hardware or just in virtual environments, Data Layer is the broker of data between different applications and with SDK there are various possibilities to create own applications to the ctrlX CORE. In the Figure 2.3, there is an overview of how the ctrlX AUTOMATION family constructs.

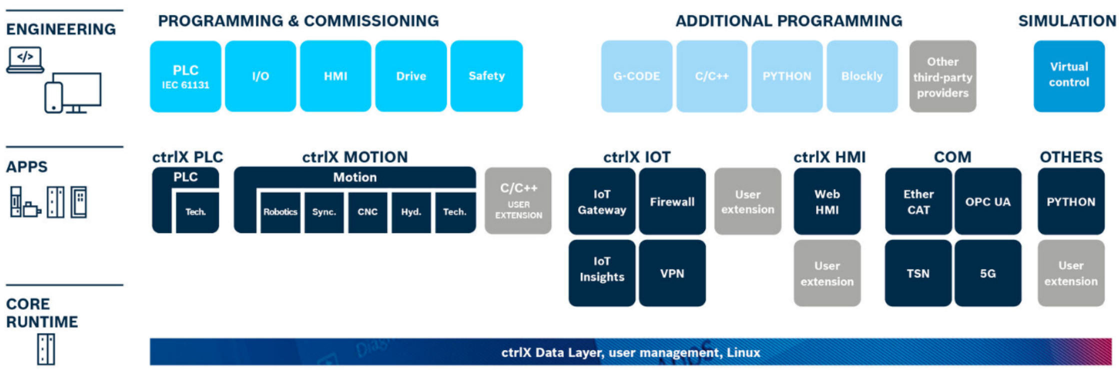

Figure 2.3. *ctrlX AUTOMATION structure [15]*

2.2 Software development

In industrial settings, corporations software architecture and deployment is nowadays molded into a certain format, mostly because of larger and larger customer base and the need to quickly adjust the certain project for a specific needs of customers. Software's are usually constructed in such matter that they are agile and easily adjustable with modularized standard functions for common functionalities and interfaces of certain basic purposes, and modified functions for project specific functionalities for unordinary cases implemented for customers needs. Common way of seeing the modern development and operations life cycle is shown in the figure 2.4. [17]

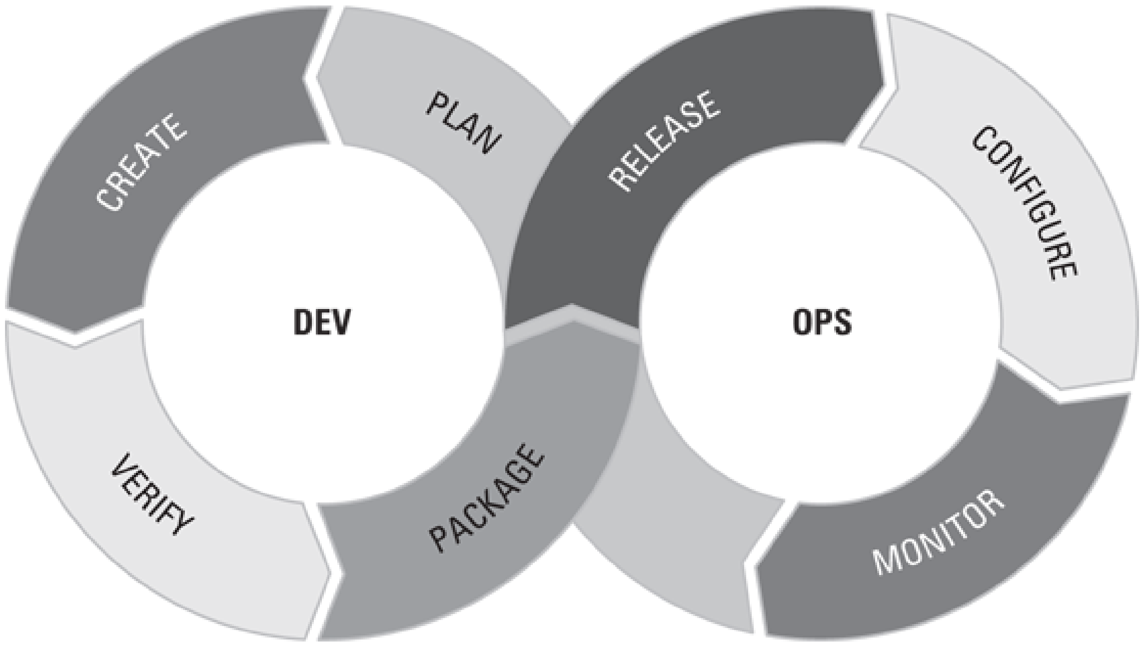

Figure 2.4. *Software development life cycle [17]*

Let's consider general model (Figure 2.5) of the so-called waterfall development pipeline that also Cimcorp has applied on their own software development pipeline for the WCS with slightly different steps on the way of course. There have been a lot of customer feedback and adaptation to that feedback over the years, but the main structure has stayed the same. The main components are design, production and installation. Loosely compartmentalized, in Figure 2.5 planning, designing and coding are corresponding to design in Cimcorp. Testing corresponds to production and deploying/maintaining to installation and commissioning. Testing in Cimcorp is not that unambiguous as just testing in production because it also contains virtual simulation of different parts of the systems, but overall these are proved to be effective methods and models to follow when creating software in industrial settings. Every corporation is modifying these steps based on their own necessities so not every mold is the same.

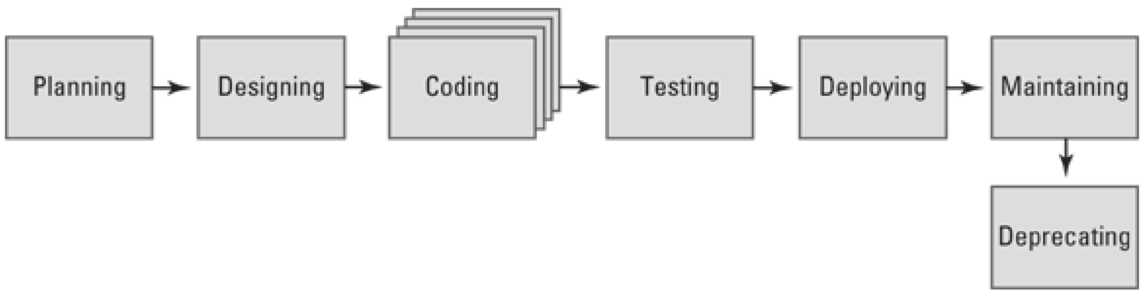

Figure 2.5. *Software development pipeline [17]*

As the new automation platform is entering it's way into the Cimcorp's products, it is necessary to research not so well-known application creation methods in-house and determine the best environments to develop those applications. This thesis is therefore trying to adequately research and implement modern software development methods for applications and also to establish a standard interface for the snap inside of ctrlX CORE which can then be used in the future, where there might come up more needs for applications which are useful in the ctrlX CORE.

2.2.1 Snap application development

As stated in the chapter 1, a snap is application package with the application itself and all the necessary dependencies bundled together to work in different Linux distributions. To clarify some similar terms usually used with the term snap; snapcraft is a tool or command to package the developed application in the snap format, in other words it is used to create the installable .snap files. Snapd is a background service that manages or allows users to use snaps and lastly snap store is a place where users can upload their own applications or download someones snap, similar to well-known App store or Google play, but for applications for Linux distributions. [31]

First step in any software development is to determine the programming language that is going to be used. It is usually chosen with the wanted end result in mind. Whether the need is for machine learning or web applications, Python is the one to go with, or if the application is needed to perform and execute functions more efficiently then C++ is the language to choose. In snap development there is no exceptions, first steps include the decision on the programming language to determine how is the snap essentially built. Snapcraft uses so-called plugins, or addons, to build the environment where the snaps are then created. Commonly used plugins, depending on the programming language, are cmake, python, catkin or conda for instance. These plugins are the basis for any snap because they lay the foundation of a snap and are often referred to an extension of the programming language so this must be chosen at the very beginning of development work. This thesis will examine development work with C++ and Python programming languages, so the build systems that are going to be used are cmake and python. [30]

Next important part of snap development is creating the "snapcraft.yaml" file that is used to define the crucial parts of the snap into a single file from where the plugin can then build the snap based on the information located in that .yaml file.[30] Bosch Rexroth have been kind enough to include examples and instructions on how these files are supposed to look like for ctrlX AUTOMATION environment, so the snaps can access the Data Layer and all the necessary locations. In figure 2.6 there is an example how the snapcraft.yaml file is constructed for example application hello.plc created for ctrlX CORE. [11]

Snap software development also requires more than just plugins and one description file; select the development environment, e.g. virtual machine, where the programming and building is happening, decide the preferred IDE where the applications are written, install or create required extensions for different languages, and also all the necessary installation for the ctrlX AUTOMATION applications for where the snap is going to be installed etc. These topics will be discussed more detailed later in the thesis.

```yaml
# Getting started
# See https://snapcraft.io/docs/getting-started
# Snapcraft references
# See https://snapcraft.io/docs/snapcraft-yaml-reference

name: sdk-cpp-helloplc
version: "2.0.0"
grade: stable
summary: Write PLC variables
description: |
  This app determines symbolic PLC variables from plc-program 'PLC_PRG' and writes in all variables of
  the type string 'HelloDeveloper', in all integers '1337' and in all floats the value '0.815f'.
base: core20
confinement: strict

# define how apps & services are exposed to host system
apps:
  helloPlc:
    command: helloPlc
    # interfaces to connect to https://snapcraft.io/docs/supported-interfaces
    plugs:
    - network
    - datalayer
    daemon: simple
    passthrough:
      restart-condition: always
      restart-delay: 10s


#  describes how to import, and potentially build, each required part of the snap:
parts:
  helloplc:
    plugin: dump
    source: ./generated/build

  deb:
    plugin: dump
    source: ../../deb/ctrlx-datalayer-1.7.5.deb

# describes which interface slots are supported by defining the plugs
plugs:
  datalayer:
    interface: content
    content: datalayer
    target: $SNAP_DATA/.datalayer

```

Figure 2.6. *Example of an snapcraft.yaml file from SDK C/C++ samples [11]*

2.2.2 App Build Environment

Application Build Environment is needed to develop CORE applications with the ctrlX SDK. This environment is Ubuntu 20.04 Linux operating system which at least Bosch Rexroth is suggesting to use for optimal performance. There are two different methods on how one can use such environment. Ubuntu can be directly installed to the PC or it can be installed to a virtual machine and then used from computer with Windows 10 as operating system for instance. This thesis will focus on the virtual machine side because the work laptops that are provided by Cimcorp are running Windows 10 and there are no physical components available for ctrlX devices so the virtual emulation of actual hardware is needed to be done on Windows operating system, therefore the virtual machine side is the needed aspect for building the snaps. [11]

There are also few possible methods for installing the virtual machines. Applications that Bosch Rexroth is suggesting to use as build environment are QEMU or VirtualBox, which both are free open source software's, but it is also possible to install Ubuntu distribution to other virtualization software like VMware which is in use in Cimcorp. Basically VMware and VirtualBox have the same operation methods regarding the use, with only difference that VMware requires a paid license. [11]

So what is the difference between the installations of QEMU and VMware/VirtualBox? For VMware/VirtualBox virtual machine you'll have to install an Ubuntu Desktop Edition operating system which is basically as close as possible to native Linux OS installation on ones personal computer. This is the preferred method if the developer wants to have full Linux experience, but is limited to use Windows host machine for some reason. QEMU virtual machine on the other hand is using Ubuntu Server Edition operating system to emulate the Linux OS. The server edition of Ubuntu does not support graphical user interface (GUI) and therefore lacks some utilities and applications that are provided for the desktop edition of Ubuntu. See the difference in figure 2.7 [11]

QEMU setup is also slightly different because Bosch Rexroth has used the Ubuntu Cloud-Init Technology which means that the virtual machine instance that is going to be installed will install a default user and the necessary tools to build snaps at the first start up. These cloud images that are used in the install process is a sort of operating system templates so every time there is a new install from one of these images, it will boot up using the same default users and settings installed. With VMware or VirtualBox, Users have to create their own users and password and install all the necessary tools and dependencies by themselves which might take slightly longer and take more effort. [11]

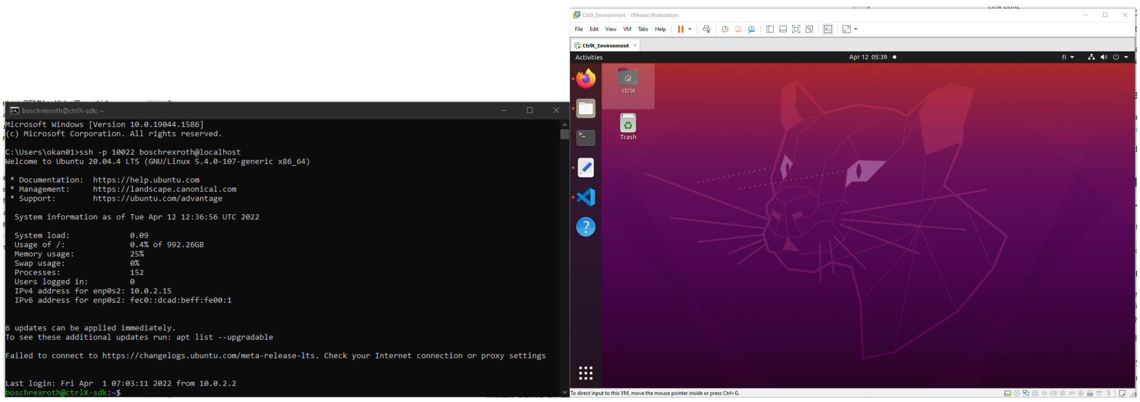

Figure 2.7. *Comparison of virtual machine user interfaces, QEMU vs VMware*

There are also some key differences in the settings when installing these environments. For the Ubuntu Desktop version, it is possible to in install a Multiarch, which is a method where operating system can have multiple architectures on the same machine. With Qemu, there is a option to only install one architecture at the time for that one instance of a QEMU virtual machine. For instance from the figure 2.7, Qemu window shows the line "Welcome to Ubuntu 20.04.4 LTS (GNU/Linux 5.4.0-107-generic x86_64)" which means that this operating system is running AMD64 architecture. The second architecture that is possible to have that is also supported by the ctrlX CORE is a AARCH64 architecture. This leads us to some more questions about these different, AMD64 and AARCH64, architectures, what do these terms mean and how do they differ from each other? [11]

Architectures are part of a abstract model of a computer that determines how the CPU is controlled by the software. These instruction set architectures (ISAs) are developed by different manufacturers for different purposes depending on the use case. AARCH64 or ARM64 is a common architecture used in mobile devices or other interconnected devices in Linux environments when the AMD64 or x86-64 is commonly found on personal computers or serves. Whether the application is developed to ctrlX CORE (AARCH64) or ctrlX CORE$^{\text{Virtual}}$ (AMD64), the architecture must be chosen based on the wanted end product and use case. [20] [2]

IDE's or integrated development environments are also one crucial part of the whole application build environment. This is the place where one can install the extensions, like the plugins mentioned in the chapter 2.2.1, to the build environment and then start writing out the wanted program. Popular and recommended IDE to develop ctrlX applications is Visual Studio Code. Visual Studio Code has a extensive library of extensions where it's possible to choose the wanted plugins to write and create programs with the preferred language. It is also recommended to use because of its support with SSH (Secure Shell) protocol which the QEMU virtual machine uses to communicate with the VS Code. VS Code can be installed to Windows 10 machines and from the VS Code there is possibility to get access to the files of Ubuntu virtual machine via SSH connection. From this connection there is a possibility to also download the files directly from the quest OS to the host computer with Windows OS. [11]

2.2.3 Programming languages

The two programming languages that are researched for ctrlX environment for Cimcorp are Python and C++, as mentioned earlier. The reasoning behind selecting these two languages are the expertise and familiarity in Python language in Cimcorp and for the C++, the original development language of ctrlX AUTOMATION. At least at the time of writing this thesis, ctrlX had some functions for the applications that were only available to be done with C++ and not supporting Python for different reasons. For instance, a real time access of data by either providing or accessing the data in Data Layer can only be done with C++, and not with Python. Real time data in this context means that the application developed with C++ language can theoretically run in same cycle times as the field busses. The cycle time means the time the CPU needs to execute the cyclic program, to update the inputs and outputs as well as for all parts of the program, see figure 2.8. These cycle times are measured with milliseconds and are slightly different based on the communication protocol used. Programming languages that are capable of running "real time" are for example the PLC codes based on the IEC 61131-3 standard and the C/C++ languages. [27]

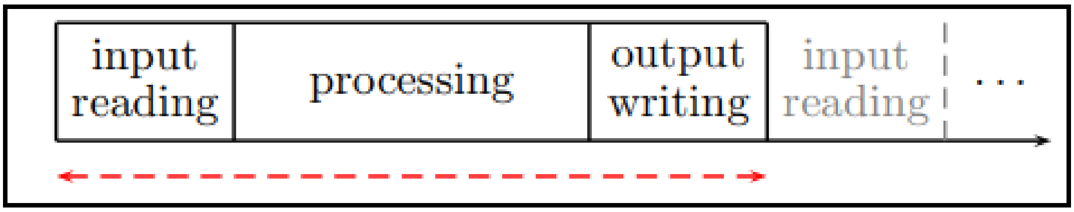

Figure 2.8. *Example of PLC cycle time [27]*

So how do these two languages differentiate from each other in the low level that causes some of these differences that was mentioned earlier? One of the main differences between programming languages is the execution model of the program. Most common ones are interpreted languages, such as Python and compiled languages, such as C++. Compiled programming language means that the one or multiple source code files are precompiled with a compiler into a executable file that is translated into machine code, which the computer can then execute without intermediary support system. This makes the program much more efficient and faster to execute than the interpreted languages. Interpreted language, like Python, requires some sort of support system, or what is called the interpreter, to be able to execute the program within a required virtual environment. Interpreted programs are not so efficient, but they can work well cross-platform when the compiled languages need a specific platform where they are complied for. [18]

How does a compiler know, which source code files needs to be linked together to create the executable file with compiled languages? For instance, with ctrlX software development kit the recommended tool to create and build the executable files is called CMake. CMake is a open-source tool to build, test and package software for compiled languages, such as C++ and it can be added as extension to the Visual Studio Code IDE. CMake building process always starts from the file called CMakeLists.txt that is needed to be added to the folder of the project. As the project size and the complexity of the program increases, so does the length of the CMakeLists.txt. Basic program only requires three lines of code; version of CMake that is used, project name and the source code file names. If the built dependencies are then going to be used in the creation of snap application, it requires some additional settings to be configured to the CMakeLists.txt file. [8]

With Python language the execution of program is done via the interpreter. Python interpreter is installed to the machine where the program is wanted to be executed and when the file is invoked on the console of said machine and the correct interpreter is set. For instance,

```
#!/usr/bin/env python3
```

is the first line that is read on the invoked main.py file which then specifies the correct interpreter, python3 in this case, to be used from the correct location where the installation of python is located. [32]

Snaps developed with Python also require one additional file in addition to the snapcraft.yaml file. This file, setup.py in ctrlX AUTOMATION environment, is configuration file where the additional scripts and installation requirements are specified. Somewhat similar functionality to the previously mentioned CMakeLists.txt, a file where the dependencies are configured.

2.2.4 Software compilation and snap building

C++ is a compiled language which turns human readable source code into machine code as mentioned in the previous section 2.2.3. This process of generating machine code consist of sequence of complex tasks with every step involving different tools. These tools are nowadays encapsulated into a single tool for simpler and more user friendly interface in mind though, but the overall phases are described in the figure 2.9. Preprocessing is intended to use to prepare the source code files for the compilation process. For instance, preprocessor is commonly used to with "#include" directive to include header files (.h ending in filename) where some functions or classes used by the program are defined etc. [18]

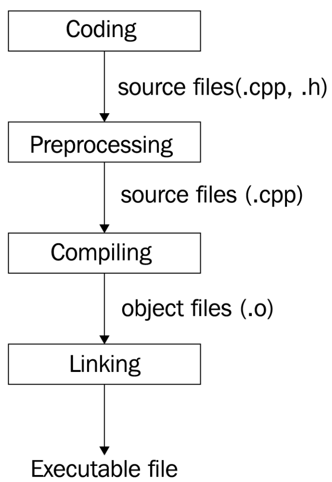

Figure 2.9. *C++ Program-building phases [18]*

Let’s look at the Compiling phase from the 2.9 slightly more closely in figure 2.10 because the methods used in the first three steps in this phase are also used with interpreted programming languages, but more on that later.

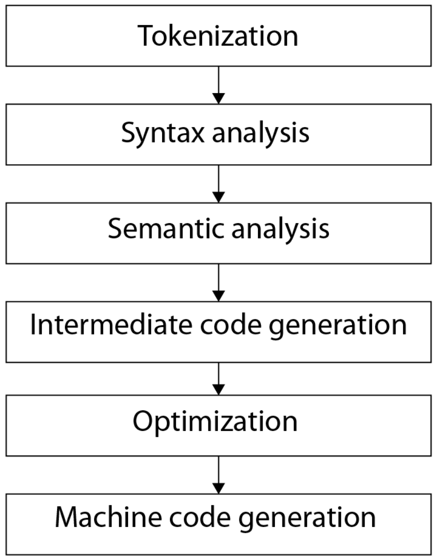

Figure 2.10. *Phases used in compiling step with C++ programming*

Tokenization, or sometimes also referred to lexical analysis, is a step where the compiler divides the source code into smallest meaningful units for the compiler called tokens. Tokens can be either some symbols or words, for instance "=" sign or "string" can be a token. Next two steps, syntax analysis and semantic analysis, then checks the list of tokens gathered from the program and analyses them syntactically and semantically to create a what is called a structure of the program. For instance, C++ compiler throws an syntax error if the line of the source code doesn’t end with the semicolon and in the compilation face to some unknown type error if it doesn’t find any meaning for some token. [18]

Next three steps are somewhat self explanatory. Intermediate code means a light, simplified version of the code where it is easier for compiler to optimize the code, for instance to transfer "a=1+2" into "a=3" etc. And lastly to create machine code out of the source code. [18]

The machine coded executable file is the end result of this whole sequence described in the figure 2.9. This file is then used in the process of generating the snap application. When building a snap with Snapcraft, the application is described in the .yaml file as has been stated before in the section 2.2.1. The parts that are used in the application are also defined in said file. The previously generated executable file and its location is defined into the snapcraft.yaml file as "parts" so the snapcraft command knows where to find correct dependencies to include into the snap. See an example on how parts are defined in the figure 2.6.

Development of snap packages with C++ programming can be roughly categorized into 4 main steps; Creation of the source code files, configuring the CMakeLists.txt file and building the executable file, configuring the snapcraft.yaml file so the correct parts and other components or requirements are found for the application and then finally building the installable snap file.

Figure 2.11. *Abbreviated C++ snap development flow chart*

As stated before, Python is interpreted programming language and the execution of the program is done via the interpreter. Interpreter is the intermediate that translates the source code files to the machine code which the computer can then understand. This conversion is done line by line as the program is being executed. Because the interpreter reads each line of the source code separately and executes the corresponding task related to that line, the execution of the program is delayed every time the code is used. This is the reason why interpreted languages are not efficient compared to compiled languages where additional translation is not required because it has already been done during the compilation. [22]

There are 4 main steps taken by the interpreter when it is translating the code into machine code. First three are almost the same that were used with compilation phase of the compiled languages, see figure 2.10. After syntax and semantic analysis, the interpreter takes the abstract representation, or the structure, of the program and evaluates it against the given input data and then outputs the result directly. [22]

To package applications with Python programming language into a single deployable snap, there are some additional assisting tools required or at least preferred to be used. One being tool called Setuptools that is a collection of enhancements for the Python language to allow developers to more easily build and distribute packages, especially ones that have dependencies on other packages. This tool collects all the requirements, scripts and packages into one file where it is easier to change when necessary in large projects. [29]

Also very popular and used tool to help in application development with Python is Virtualenv. This is a tool to create isolated Python environment so that all the dependencies and versions are correctly installed for the application. For instance, if there are some other application that require other libraries and versions in Python to be used, this doesn't break the first project with newer or older libraries required for the second etc. [33]

Creation steps needed to take for the creation of packaged application is shown in the figure 2.12. After the creation of the source code, requirements.txt file needs to be configured to have correct dependencies. This file can then be run using Pip, a python package installer, so for instance the ctrlX Data Layer Debian package can be installed correctly for the application. Setup.py file is a Python script that uses functions from the Setuptools library. Here you describe the necessary source files and their location with required installed packages. We have been through the Snapcraft.yaml file already previously, but in short, the functionality stays the same always, just necessary changes is to change the application name under apps, and define the correct parts for the application under parts. Then it is possible to run the snapcraft command with specified architecture on the terminal and build the .snap file which can be exported and installed to systems running said architecture.

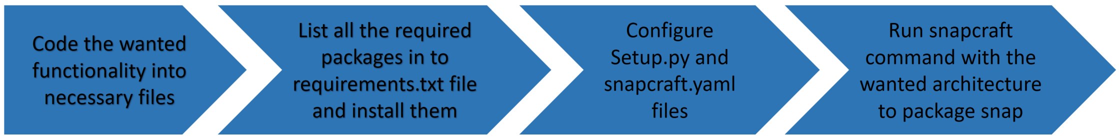

Figure 2.12. *Abbreviated Python snap development flow chart*

2.2.5 Version control systems and testing

Version Control System (VCS) in overall can be described as a system that holds the records about the changes in specific files over time. The files located in the system can basically be anything from text files to images etc. Because the system saves the files over time, every change can be seen and if necessary reverted back to the previous older state or even compare the newer and older files between each other to see changes. System also tracks its users in the project, so if someone makes some changes to the project files, it can be seen who it was and what was made. [5]

Most used choice in every day version control is to just copy the files in to some other directory that probably everyone has sometimes used in their everyday life with computers. Developers have then been introducing different types Version Control Systems, more specifically three types of VCS's. Every version is an improved version of the previous one to tackle some problems or downsides that the developers stumbled upon previously. Local Version Control System is running on the local computer just keeping the files under version control database 2.13. This was first improvement to just copying and pasting it locally on one computer. [5]

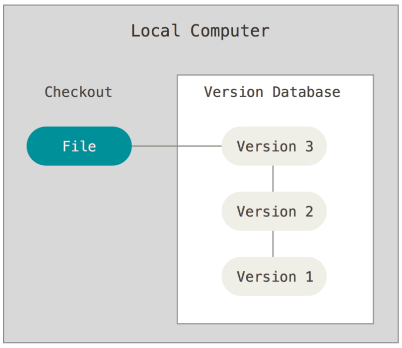

Figure 2.13. *Local Version Control System [5]*

Obvious downside of the local version control system is that it only works locally on one computer and the developers can't easily work together on the same project. Second improvement on version control then happened with centralized VCS (Figure 2.14). This type of version control was the basis in the developer community of any kind for quite some time. Centralized VCS answered the needs of developers that needed to collaborate in large projects with other developers.[5]

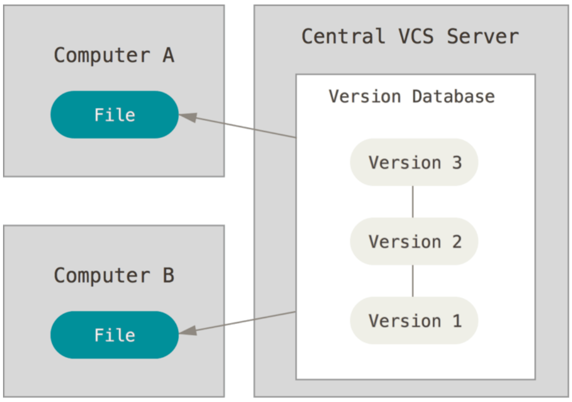

Figure 2.14. *Centralized Version Control System [5]*

Centralized VCS has also pretty obvious downside to it. Image if the central server for some reason crashed or the hard drives become corrupted, all the data will be lost permanently. This was then tackled with the last type of VCS, the distributed VCS 2.15. This is the most common version control system nowadays, one DVCS for instance is popular Git. With systems like these, there is the server where the original repository is held and then with every user accessing the data or working the project, the client computer mirrors the whole repository from the server. This method removes the chance where the whole repository and data could be permanently removed or be inaccessible in critical moments because every client has the whole history and version of the project in their local computer, which can then be used to restore the original server if necessary. [5]

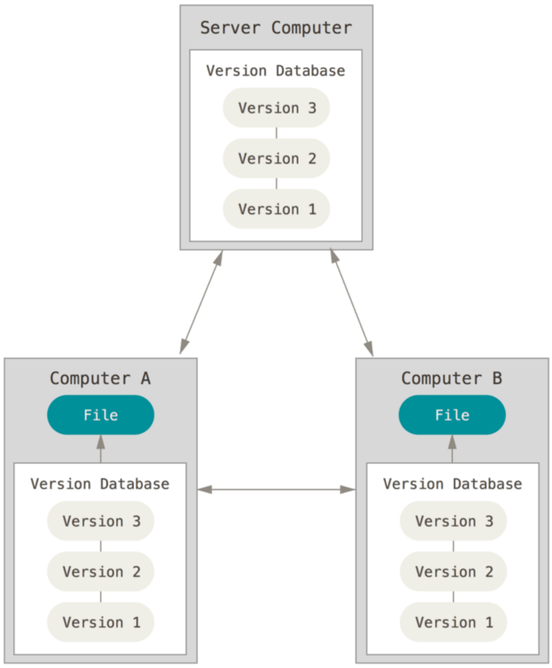

Figure 2.15. *Distributed Version Control System [5]*

Modern DevOps often is aligned with terms continuous integration and delivery (CI/CD) which are considered a set of practices that partly automate and help in the software development. These practices are trying to remove the often seen bottlenecks in software development with testing and delivery of the software. One of the first steps in software development planning phase should be the organization of the source code to some version control system. For most industrial cases and other organizations, the best tool for this is a distributed version control system like Git and some cloud hosting service like GitHub or Bitbucket etc. behind it. After the source code maintaining environment has been decided and some sort of version control system is setup, it's time to consider the testing of the software. [17]

As Emily Freeman mentions in her book [17], the testing part of the software in development life cycle should also include the following parts:

1. Developing test cases
2. Writing automated tests
3. Running manual tests
4. Reflecting on the delivery
5. Making adjustments

Automated tests should consider also different aspects and functionality of the program. There are different types of tests for different purposes and different parts of the programs. Most simple and general purpose test is the unit test. Unit test tests the functionality of the method and the logic it is trying to achieve in the program. Unit tests are simple, fast and cheap because they only test one piece of code. [17]

When the application has been configured to access required databases and file systems, the next test is recommended to be integration tests. Integration tests are for multiple methods simultaneously to test that they produce and achieve the wanted outcome. Performance tests are also verifying the applications stability and responsiveness. These tests can also include some security and load tests, where the system is put under a lot of traffic etc. There are also few other tests practices for user interfaces and other parts, but theses three are considered the most important. [17]

If the automated testing steps are configured correctly, then the continuous integration and delivery can be achieved. For instance, if the source code is committed to the Git, the automated testing suite runs the configured tests like unit and integration automatically to check whether the program still works as intended or not. This is very beneficial with large complex systems where there are different environments and just too many aspect for a human to confirm that everything works.[17]

3. PROPOSED DEVELOPMENT PRACTICES

This chapter explains the proposed implementation of how the application interface to the ctrlX CORE could look like and how it can be used as a default module to implement different functionalities to different applications for the ctrlX CORE. This chapter also grasps the proposed development pipeline of applications for ctrlX based on the modern software development practises and available documentation about the Software Development Kit (SDK) by Bosch Rexroth.

The sections in this chapter are divided into three categories; first we will talk about the ideas behind needs for this thesis and describe the initial implementation ideas that were brainstormed for the application. Second section is focusing more on the idea of universal module that can be used as a template for creating applications for the ctrlX CORE. Finally, the proposed methods suggested for creation of application for devices running Linux kernel.

3.1 Planning the implementation

As have been explained earlier in this thesis, Cimcorp is implementing the new ctrlX AUTOMATION family devices and software's into their device portfolio. This change of low level hardware and new software created an opening of third party software implementation to the proprietary applications like the gantry robot developed by Cimcorp. In industrial cases the possibilities of different applications is almost infinite and there is a lot of useful cases that can be done with additional applications running in the background of the ctrlX CORE.

Industrial devices have massive amount of sensors installed to them to either monitor some critical values or movements etc. One particularly interesting monitoring value in devices that are using electric motors and is moving continuously at high speeds is temperature value of the parts that are put under a lot of pressure. Temperatures are also critical in electric cabinets where dozens of electric components are pulling a lot of power and generating heat which needs to be ventilated out so that the cabinet and devices in side of it won't over heat.

The fans and ventilation units in warm warehouses or tire factories and heating systems in cold environments are often controller with the PLC because the sensors are usually directly connected to it and its pretty easy to do. Now with ctrlX AUTOMATION the PLC, third party applications and external systems can be connected easily together with Data Layer and exchange data via it or OPC UA (see figure 3.1). One popular format of data that is easy to read by humans, easy to parse by computers and is very intuitive is called JSON format. JSON stands for JavaScript Object Notation and examples of how one can read and write JSON formatted data with C++ and Python is showcased later in this thesis [21].

An idea sparked in the technology development department in Cimcorp to create an application that can be used as an intermediate to monitor the temperature values of different objects in industrial cases. Lets consider a situation where a robot solution is using ctrlX drives and the ctrlX PLC application is running and controlling the whole robot system movements etc. ctrlX CORE is therefore also installed either directly to the drives or just as a separate installation in the drive cabinet and ctrlX I/O is monitoring the sensors from the field. The data from the sensors can then be gathered with the ctrlX PLC into structures or just as normal variables and be sent forward to the Data Layer. Data Layer then works as a broker of data and some external entities can therefore use the data directly from the sensors in field level.

This idea then contributed forward and generated an idea that what if this application can be used as sort of a database or temporary database to host the gathered data from the PLC. This data can be used in all sorts of analytics or in alarm handling or for instance to detect before hand the wear and tear of physical parts of the system, possibilities are endless. Figure 3.1 illustrates the preliminary idea behind the application.

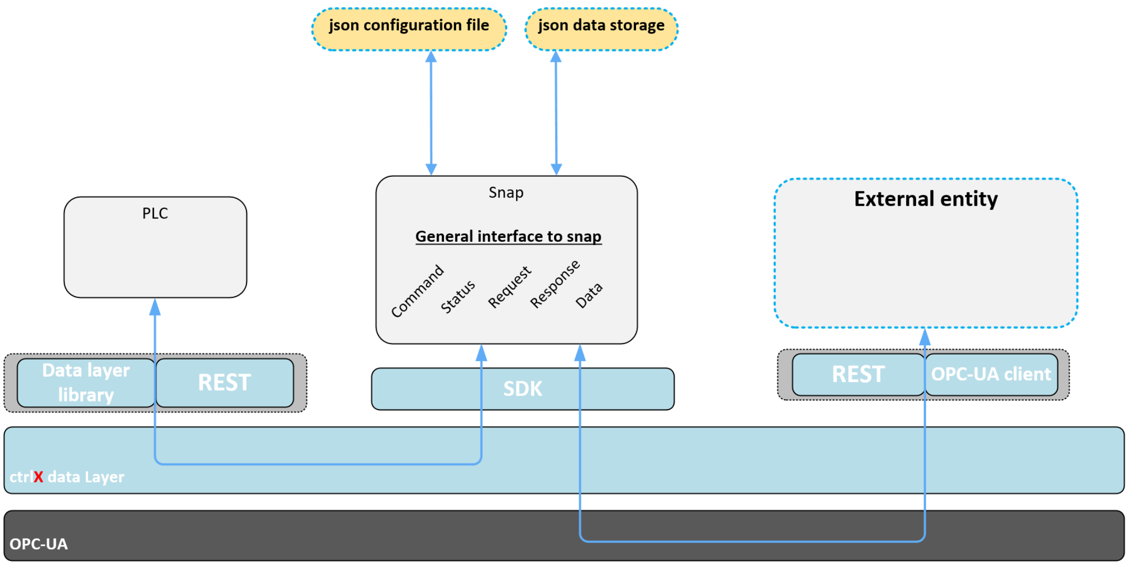

Figure 3.1. *Concept idea of how different parts are connected to each other*

Let's go through the figure 3.1 more thoroughly next so one can understand more precisely of how these application communicate. As stated in the section 2.1.2, the Data Layer is a data broker and it does not store any kind of data and does not have any constructed size. It only knows where the data is located and in what format it is so it can convey it forward so the speak. Conceptually the Data Layer is just like a tree of addresses with hierarchical paths to the data structures, which are called nodes. [13]

In figure 3.2 it is showcased how, for instance a ctrlX PLC application that has three symbolic variables created in the PLC program, shows the variables in the Data Layer as nodes. ctrlX PLC also has access to the Data Layer and if the PLC program has variables created as global variables, they can be added to the Data Layer. For instance, highlighted temperature node has been defined in PLC program as a global string variable, but of course that could be changed to any variable type that the ctrlX environment supports.

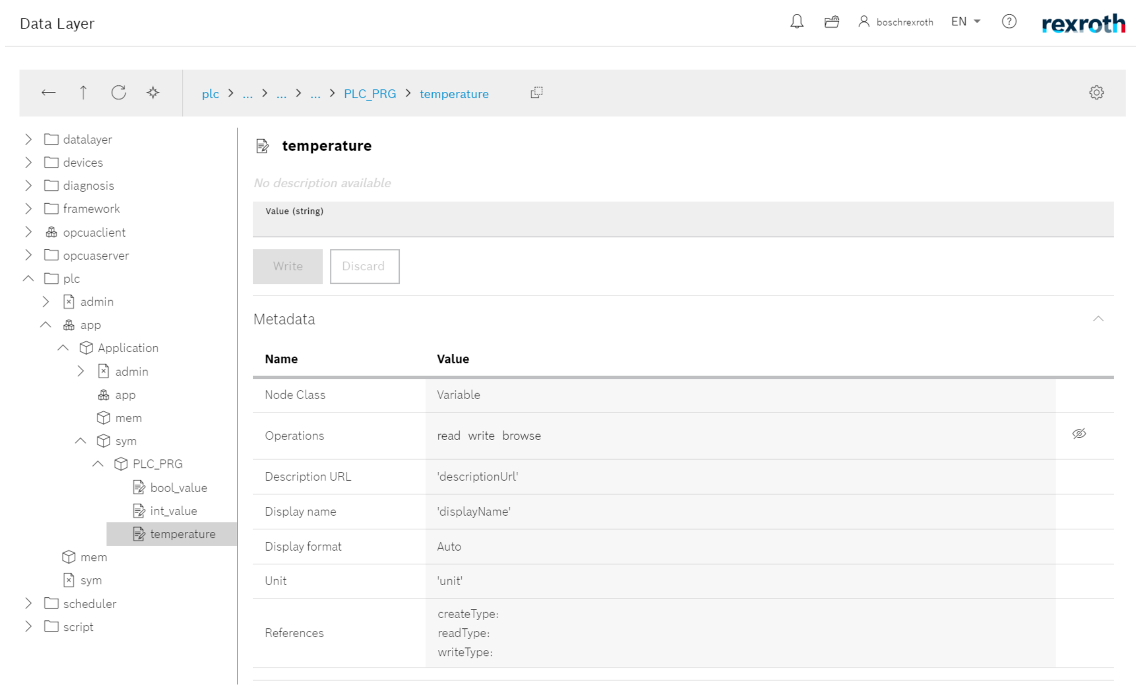

Figure 3.2. *Data Layer tree structure example*

Data Layer basically hosts an address for that specific data, in this case the path to the string variable data would look something like: /plc/app/Application/sym/PLC_PRG/temperature. From there either the PLC or some other party can read or write data, as string variable, to specifically that node and use it how ever they like.

JSON format is very versatile format of data and it can be also considered and parsed as a string variable. There are extensive libraries in both C++ and Python that can take string variable that is in JSON format and parse it in to the format that the programming language then understands and can efficiently use.

Different implementations that are possible are so that the PLC provides, or creates, the node that the third party application then reads for string variable with JSON formatted data or it can be implemented so that the application creates the node and then the PLC writes data directly to that. For the sake of this thesis, we decided to go with latter and provide the node in the wanted application and then have PLC hypothetically write some data to it.

The preliminary idea behind the application also contained ideas of a configuration file to configure the interface to and from the application and also an database setup to save the collected data. Configuration file could for instance hold the names of the nodes in JSON formatted structure that the application then reads and provides the nodes for the Data Layer. Data storage was also originally thought to be created in the application, but as it turned out from discussions with Bosch Rexroth, the SD card in the ctrlX CORE might get full quickly if the data amount is large. Because of the limitations in disk size, at least at the time of this thesis, an idea came to just save the data temporarily into a JSON file in slightly smaller chunks and then send that file to more permanent storage. This thesis will only focus on the temporary file creation and not the permanent solutions.

Figure 3.1 also contains some aspects that have not yet been discussed in this thesis. Next section 3.1.1 discusses about different implementation and methods of how one can access the Data Layer outside of the ctrlX environment.

3.1.1 Communication outside of ctrlX CORE

So now we know the basic principles of how Data Layer works with applications directly installed to the ctrlX CORE and we have decided how and in what format we want to access and use the data in the application that we want to create. How can one then get access to the nodes outside of ctrlX environment? This can be solved with either REST API or OPC UA. Next section 3.2 discusses more about the importance of API while this section concentrates more on different methods on how the connections work to the ctrlX CORE.

OPC UA stands for Open Platform Communications Unified Architecture. Shortly described, OPC UA is data exchange standard architecture for industrial automation spaces. This architecture standard is a series of specifications developed by industry vendors, end-users and software developers to define the interface between Clients and Servers. It is widely used because it is secure and reliable data exchange between different vendors. [23]

OPC UA Server application can be installed to the ctrlX CORE and therefore work as an intermediary between CORE applications and outside environments that have OPC UA Client capabilities. This also means that the Clients have access the nodes in the Data Layer. Figure 3.3 illustrates the connections between CORE and outside environments.

There is also an application for OPC UA Client for ctrlX CORE so that the CORE can access the provided data in some external OPC UA Server.

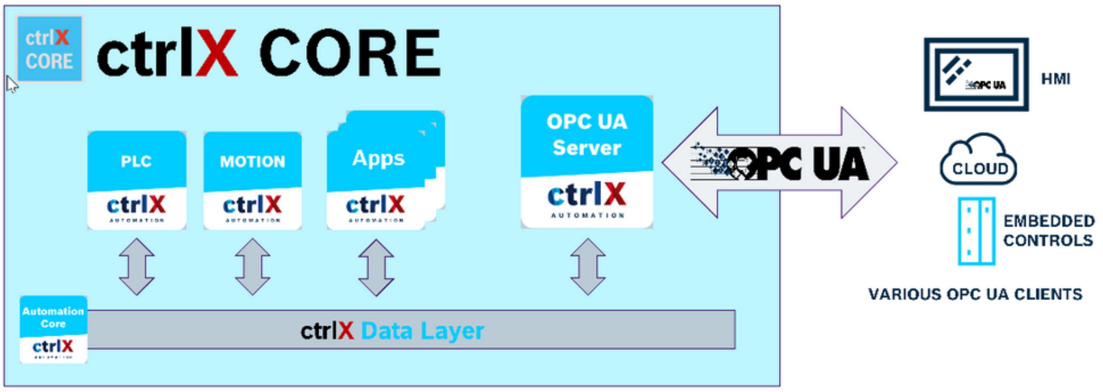

Figure 3.3. *OPC UA communication to and from ctrlX CORE [12]*

REST API stands for Representational State Transfer Application Programming Interface. REST API is architectural style for distributed hypermedia systems which means that is uses same principles to access and use data than the commonly known HTTP. In the HTTP protocol, there are different types of service request methods for data. Commonly the data can be accessed with GET, PUT, POST and DELETE commands which are used to interact with the back-end API through a URL (Uniform Resource Locator). These commands are used to perform different actions for the same set of data. [26]

Typical REST API works so that the client side requests the data that is associated with defined resource methods, for instance GET URI (Uniform Resource Identifier) and then the server responses with the header, status code of the message and then the response body. Example of this is shown in the figure 3.4. [4]

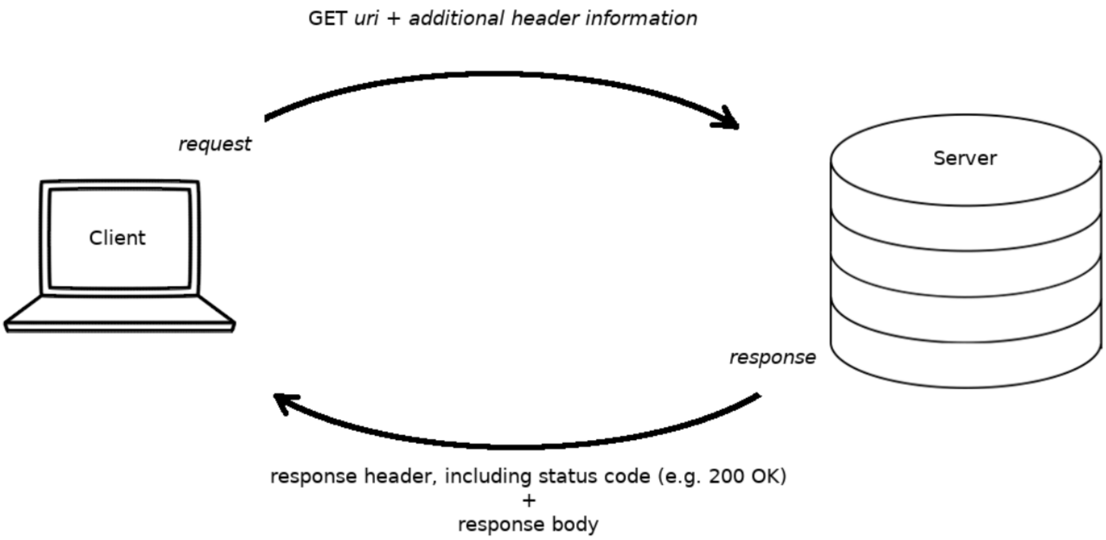

Figure 3.4. *REST API workflow [4]*

CtrlX CORE has implemented a REST API into the CORE. This means that the qualified users can access e.g. the Data Layer nodes via REST Client. With different URL:s it is possible to access different parts of the ctrlX CORE.[4]

Let's consider the figure 3.2 for a moment. For instance, if some external application want's to read the temperature node via the REST API, they'll have to send a GET URI to the correct location and then the server, ctrlX CORE, responses with the data written to that node. Full address path to the temperature node would look like this: https://192.168.1.1/automation/api/v2/plc/app/Application/sym/PLC_PRG/temperature

Figure 3.5 shows an example on how the actual responses would look like with this communication. On top is the Data Layer interface with some string value written to the node. Below is a REST client application called Swagger, this can be any client one chooses for instance YARC is a popular extension to Google Chrome etc. CtrlX works as a server like in the figure 3.4 and the client has requested from the said URL, a GET URI and this figure 3.5 shows what does the response look like actually. In the response body, it is clearly seen that it has the same string value as the node data in the upper picture so therefore the response was successful with status code of 200.

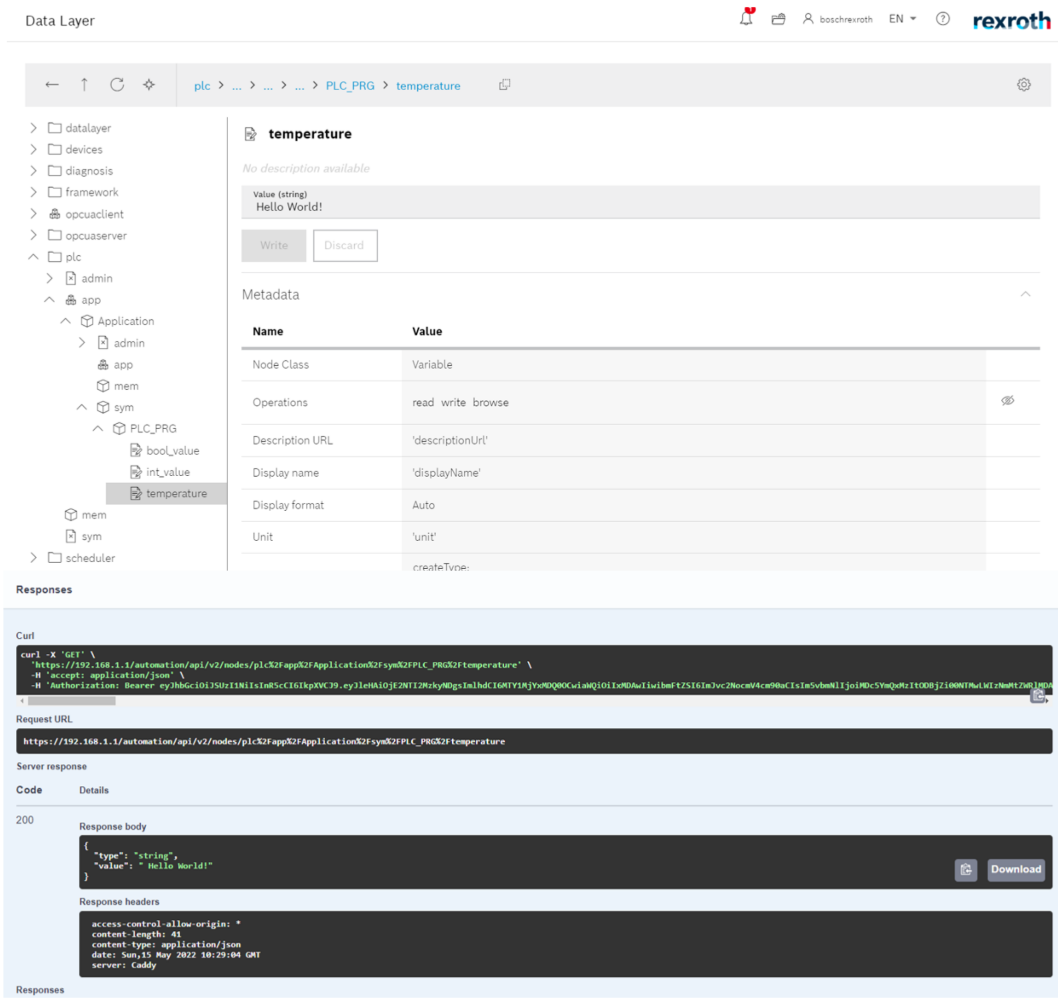

Figure 3.5. *REST API connection to Data Layer node data*

3.2 Concept of the default interface module

Application Programming interface or API for short is an interface for websites to communicate with the back-end logic. Some everyday examples of APIs are for instance some travel booking website. Let's consider Skyscanner for example; when user goes to their website, all they have to do is to enter the destination country or airport and then the wanted dates when the trip is going to happen. API handles the rest, as in finding the most suitable flight, usually the cheapest, for the user. [6]

Some form of API is used by basically every website that one enters. Google and Facebook are prime examples of large corporations that use APIs. APIs are used as the intermediary between two applications, front-end and back-end. This helps us to encapsulate information because this way one company can only show the required information through APIs and still hold on to the internal or personal information about the users that are not to be revealed. [6]

In industrial settings the use APIs is also very common. ctlX Data Layer for instance has different APIs for different programming languages. This way the developers can utilize simple pre-made functions on their applications without needing to know everything that goes behind the scenes and still benefit from these functions. One example could be reading different variables out of the Data Layer nodes with functions that are already provided by the API so there is no need for one to create these functionalities again.

One idea behind the creation of own application for Cimcorp was to create the application so that it can be used as sort of an API that engineers could utilize simple data gathering functionalities for data gathered from the field level devices. Also the idea behind the own application was to use it diversely and the interface, the Data Layer nodes, could be easily configurable. It is important that the application is versatile and can be modified easily to different purposes, as a module for different projects.

The application basically should contain two parts, the default part and the configurable part. This means that the applications, no matter what the use case is, should always use same commands to start the application, stop the application and maybe with slight modifications to request the data. If this is kept in mind while developing any program, it makes the life of other people making contributions the application more easier because then they don't have to do huge modifications to basic functions.

Initial thought was to have similar interface to the snap like the REST API has, so to use commands like GET, PUT, POST and DELETE when accessing the applications data. But as it turns out later in the thesis, this has changed radically and has much more different interface.

Basically the plan was to create the application so that we can use one node as a command node to start, stop and reset the application. This node could also take additional parameters so we could start it with additional JSON file. For instance if there is a need to get some data fast out of one particular data file, it would simple to just start it with that file running and then retrieve the data. Second of all the next default part should be the status node, which only would work as a node to represent the current status of the application. For instance, it could show if the application is in running state or it has stopped etc. Also a node for showing the answers from the application could be default but of course the data entered there would differ based on the application use case.

The configurable part would be the nodes that are used to request some functionalities from the application and of course the node that is receiving the data from other applications. The nodes basically are all the same with same data type written or read from it, but the data inside the data type is the defining part that differentiates the nodes from each other.

Default interface module should be like a template that one can just download and make the wanted functionalities with small modifications and then publish it with ease of use. Ideally it would be so that the template is just copied to new project so every basic functionality doesn't have to be rewritten in the new application. Figure 3.6 illustrates the idea behind this so it would be as simple as possible to start the development of new application.

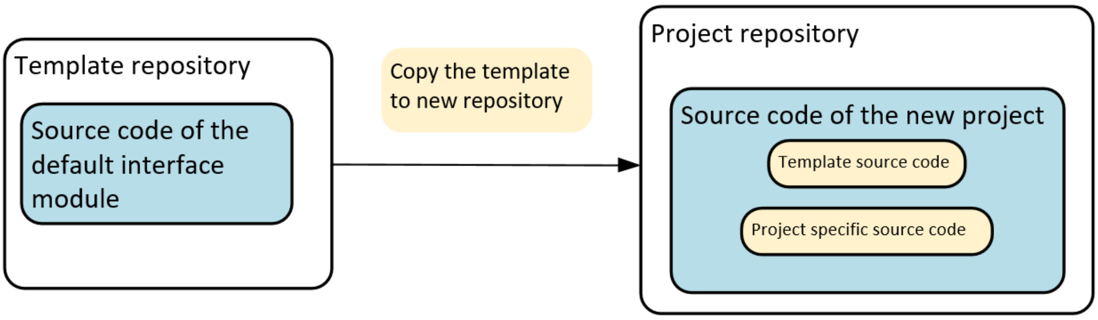

Figure 3.6. *Concept of how default interface module would be used*

3.3 Proposed development pipeline

Based on the documentation about the Software Development Kit and other documentation about the ctrlX family it has been stated quite clearly how the development of applications to the ctrlX CORE should go. Suggested environment is to use QEMU environment which will be discussed in more detail in the chapter 4. How ever this does not effect the software pipeline that much and therefore this section will focus more precisely on how everyday coding of applications with SDK could go.

After everything is set correctly in the environment the next step should be the writing of the source code. Some key principles behind the good practices in software development is to divide the source code into smaller pieces and not cram everything into one file. It would be good to use classes and other functionalities of the code depending on the programming language. This helps in the future where someone else might take a look at the code and do some modifications to it so it would be more clearer to understand.

Next up is the building phase of the snap. This step could also be done after the changes has been pushed into the repository, but this depends solely on the team that is creating the code. It is possible to configure the building of the snap to be done automatically after the changes has been merged to the main branch of the respective repository. This is done by linking the repository with application called Launchpad which is a online building service. In simple cases like the one done in this thesis, building and then pushing should be functioning with out any hiccups, because the environment and IDE's are set so that it is easy to build it with push of a button and merging updated files to the repository is also quite straight forward. [30]

The steps taken in installing the snap to the core is pretty common if one has encountered Rexroth devices before. The created application can be installed to the ctrlX CORE$^{\text{Virtual}}$ via the web browser and the steps taken is pretty straight forward. Figure 3.7 shows the two workflows that could be possible for developing applications for ctrlX CORE. The bottom graph illustrates the possible method where the created application is built inside of the repository with the Launchpad application and on the upper one is the more simple way. The upper graphs pipeline is more simple because creating and initializing the repository to build the snap after changes is merged isn't that straight forward or easy to setup.

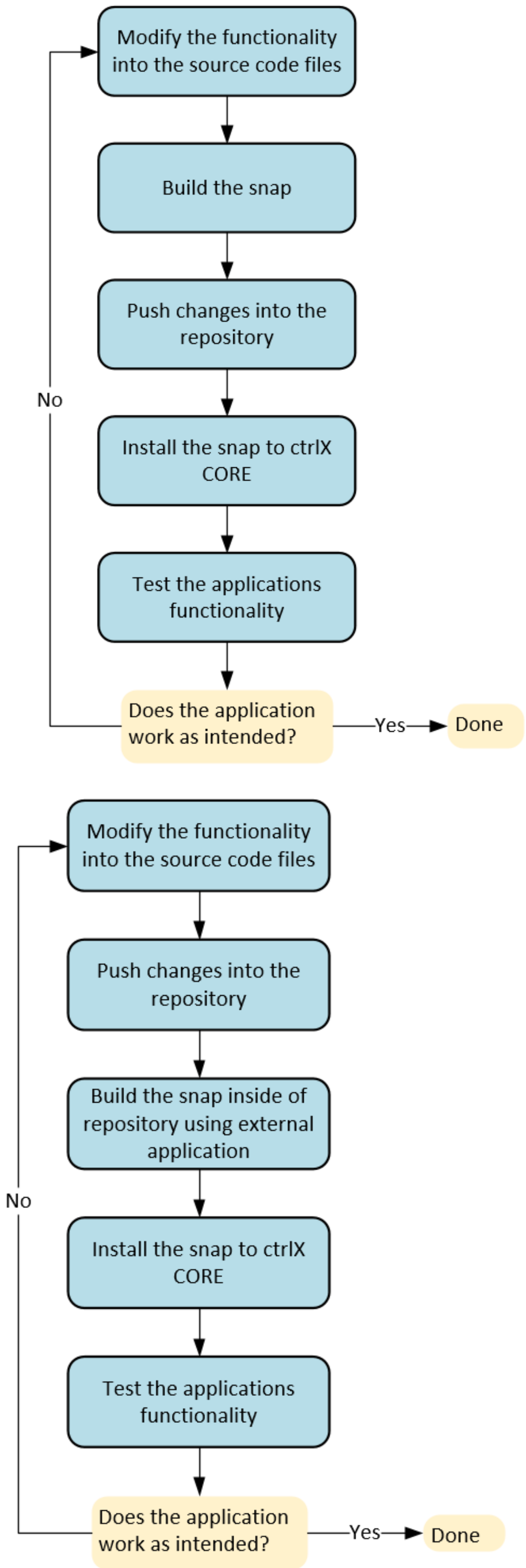

Figure 3.7. *Two proposed development pipelines of application to ctrlX CORE in large projects*

4. IMPLEMENTATION

The created application that will be described in this chapter has gone through many changes along the way of the development work and is much different than the initial thought of the application. Many of the changes are because development work with SDK has gone forward and Bosch has implemented different functions and overall updates to the SDK. This thesis started with the version 1.12.1 of SDK and the current one at the end of development work of applications is 1.14.3. This has created some improvements and also some difficulties in the development work and might be the reason for some of the "bugs" or inefficiencies in the program.

This chapter also tries to go through in detail of how the environment for creating these applications is actually set and what needs to be taken into account because it is helpful for personnel in Cimcorp to just read this document and have some sort of understanding of how things work with ctrlX applications. Side note for this is that because of development work by Bosch, some things or steps are already outdated and not necessary to do because there have been so much improvements for the new platform. It is still good to know of how things work because the underlying principles of implemented improvements are still the same.

4.1 Environment setup

It was mentioned in the section 2.2.2 that there are different ways of creating the environment where the applications for ctrlX CORE could be developed. This section will highlight the tested methods and discuss which proved to be the best way and what not so efficient. There are good instructions found from the SDK's GitHub page so it is also good the check that out if some aspects might be unanswered.

Native Ubuntu Desktop or Server operating system setup to develop ctrlX applications is not that useful at least in corporate world where the IT side of computers is very strict. Usually the corporate IT determines the working environments and often this means Windows OS. If native installation is the wanted environment to develop snaps then comes the problems with virtual testing of the software because the ctrlX WORKS, at least at the time of this thesis, was only available on a Windows OS. This means that the emulation of actual ctrlX CORE with ctrlX CORE$^{\text{virtual}}$ is not possible on native Linux distribution. So it would be needed to have two separate computers to test applications virtually and that is not very ideal. One thing that might be possible is to install a Ubuntu Server into a remote PC that could be used purely as compilation environment, but this thesis won't be discussing this option.

First researched development environment was Ubuntu Desktop running on VMware Workstation virtual machine. This was the easiest of the two researched environments to setup and did not require any tips or tricks in the setup phase. Figure 4.1 illustrates shortly on what are the necessary steps taken in virtual machine environment to start developing the applications for ctrlX CORE. These steps are same with Oracle VirtualBox virtual machine, but the creation of new virtual machine is slightly different because the interface to these machines are different. Though it is pretty straight forward with both virtual machine environments.

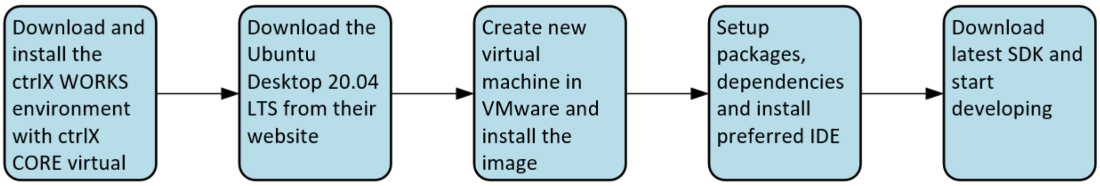

Figure 4.1. *VMware environment setup steps*

The steps above in figure 4.1 doesn't show the steps in detail because these steps can be easily found on the GitHub page of SDK or they are straight forward. Installing the ctrlX WORKS with ctrlX CORE$^{\text{virtual}}$, downloading the latest Ubuntu image and then creating the virtual machine are done in Windows OS. ctrlX WORKS is possible to be downloaded from the Bosch Rexroth's developer community, but it requires registration and purchase of the application. With ctrlX WORKS it is then possible to create an virtual emulation the actual ctrlX CORE. Setting up the packages and downloading the latest SDK is done in the virtual Linux environment because the environment is freshly created and does not have any additional applications installed. Packages that can be installed with Advanced Packaging Tool (APT) are for instance Cmake, Snapcraft and Git. Snapcraft is for instance one of the packages that is required to be installed if snap applications are created. [11]

Dependencies are installed also using the same principles as packages, but for virtual machine to have multiple architectures, some additional settings needs to be configured. For instance, if it is wanted attribute to have both, ARM64 and AMD64, architectures on the same environment to develop them side-by-side there is a need to add them both into APT's source list. This is because ctrlX CORE and ctrlX CORE$^{\text{Virtual}}$ have different architectures as mentioned in the section 2.2.2. [11]

The application is compiled in the virtual environment and therefore the created snap file will also be stored in the virtual environments folder structure. This file then needs to be downloaded to the host system where the $CORE^{virtual}$ is running and install the application to it. This downloading can happen in many ways, but probably the best solution if considering large projects with lots of contributors would be to create an repository to one of the most used Distributed Version Control Systems like Git and connect the used IDE to the repository so that the source code files can be committed directly there. From the repository it is simple to download via web browser the wanted snap and install it to the ctrlX $CORE^{virtual}$. Due to the fact that there was only person developing the application, it was easier in the testing phase of environments to just use Microsoft's OneDrive to save the files and then upload the snap from there to the $CORE^{virtual}$.

Virtual machine is not a bad option when considering the development of snaps. If one is more familiar with Linux environments and does like to use it's user interface more than the Windows one, it is a solid option. It's good to remind that virtual machines that have graphical UI are more resource hungry and with laptop that has limited memory and CPU power it might be slightly slower environment.

Second researched development environment was QEMU virtual machine. This was also the setup that was used in the development phase of the example application for this thesis. It is also good to mention that the QEMU environment improved at the later part of the development work so the original setup was a lot more tedious and required more steps to take. It will be showcased also on how the QEMU environment improved and how does it look like at the ctrlX WORKS 1.14.

The mentioned original QEMU environment was with the SDK 1.12 version and WORKS 1.12 version. Figure 4.2 shows the specific steps that was required to take in setup phase of the environment. As it can been seen from the figure, it was quite complicated and required a lot of tinkering with the settings in Windows and with different software's. Windows has massive amount of different settings and it would be very challenging to change the correct ones to get the environment working if it wasn't a good instructional videos about these more complicated steps that were found on Youtube videos created by the engineers of Bosch Rexroth.

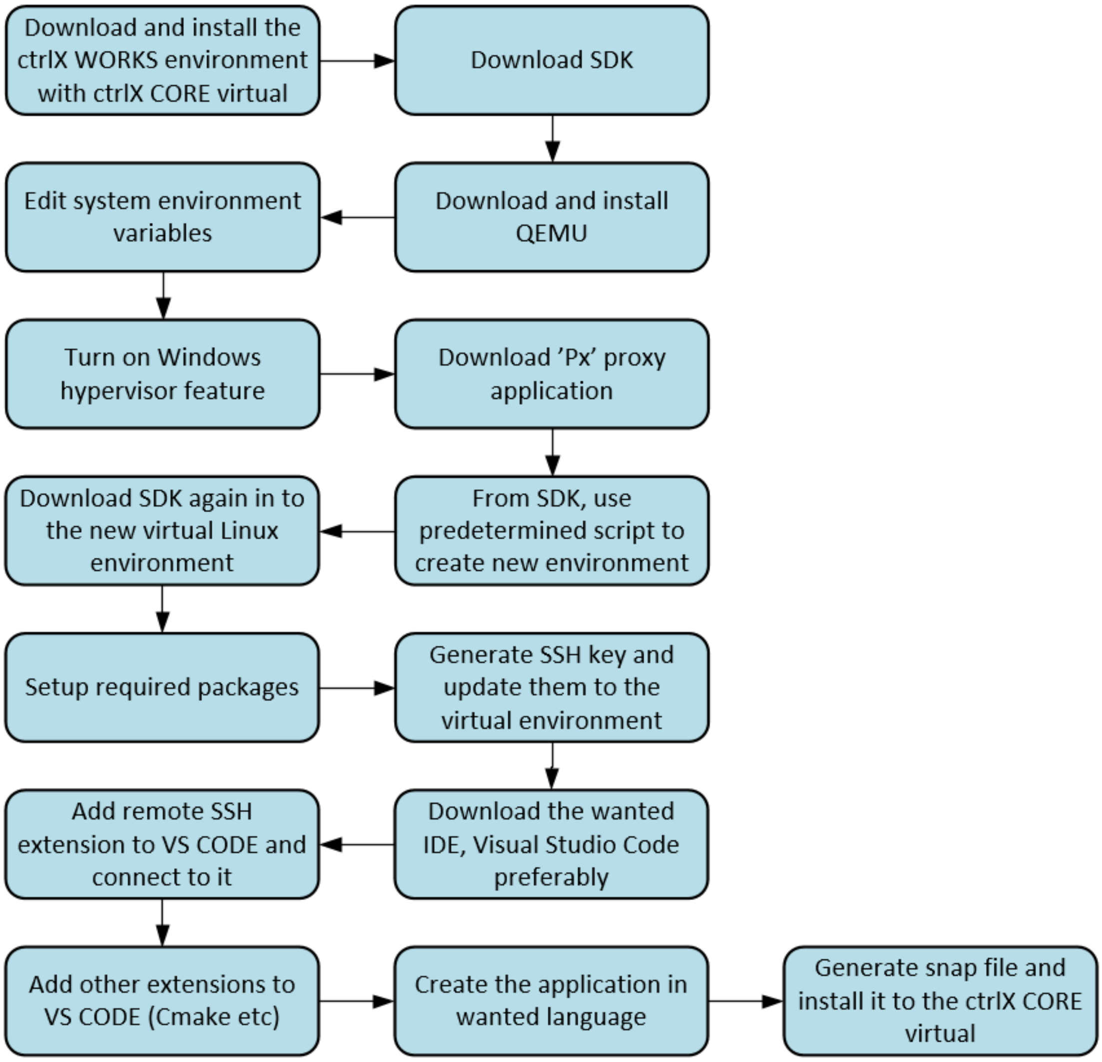

Figure 4.2. *QEMU environment setup steps*

Let's go through the figure 4.2 in more detail and focus on some of the steps that have more importance in general understanding of how the environment works underneath. QEMU machine emulator software is used to emulate the whole machine system virtually which means that everything that is needed to run a computer is emulated to provide a virtual model of the computer [25]. QEMU is the basis for everything in this workflow.

Next important aspect is the Px software. Px is open source software that can be downloaded from their GitHub page. Px software is a HTTP proxy server designed to use NTLM or Kerberos proxy servers to allow applications authenticate in corporate deployments [24]. Essentially a proxy server is a computer systems that works as an intermediary between the client computers and the web servers. When the client computer requests some data from the target server, the proxy server hijacks the communication between the parties and represents itself as the client to the target server. Proxy can be implemented in many ways and to many systems, but usually in corporate world it is a matter of security so it is designed to block the malware's and viruses or just to monitor users internet usage etc. [28]

In our case the Px proxy is used to gain access to the internet from the QEMU virtual environment. For instance, the connections to the Snap store and the use of Snapcraft command cannot be achieved if the Px is not running in the background because they require access to the internet. Figure 4.3 illustrates the general idea behind the proxy servers.

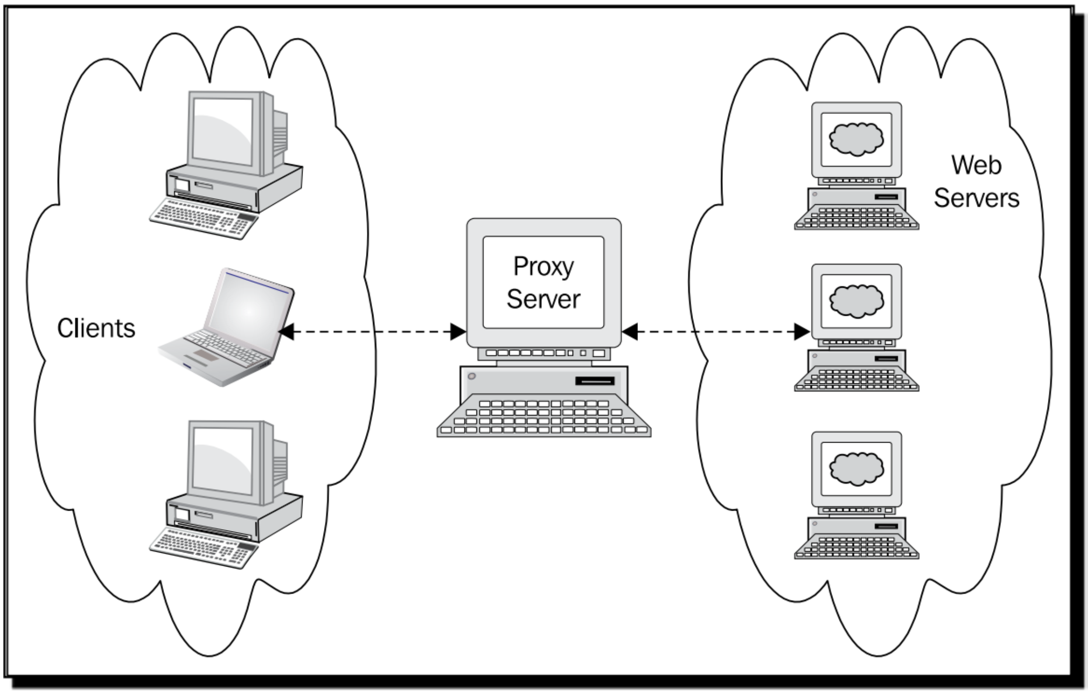

Figure 4.3. *How proxy servers work [28]*

SSH is the next important aspect of the whole environment to keep in mind. SSH stands for Secure Shell which is a software-based network protocol for network safety between clients and servers. SSH basically just encrypts data that is sent from one computer to another and then decrypts it in the receiving end when the message arrives. This client and server architecture handles authentication, encryption, and the integrity of data in the network and it is considered very secure because of strong encryption and the needs for different keys in authentication. Figure 4.4 describes the overall architecture of SSH.[3]

In the environment setup with QEMU virtual machine, there are two different SSH connections that can be established between the different parts. First one is the connection between the host and the QEMU virtual machine which is achieved with port forwarding. Secure Shell uses the commonly known TCP/IP internet protocol as its transport mechanism to provide a secure "tunnel" between the two connections over the internet.[3]

After the new environment is created and launched from the given images in the SDK

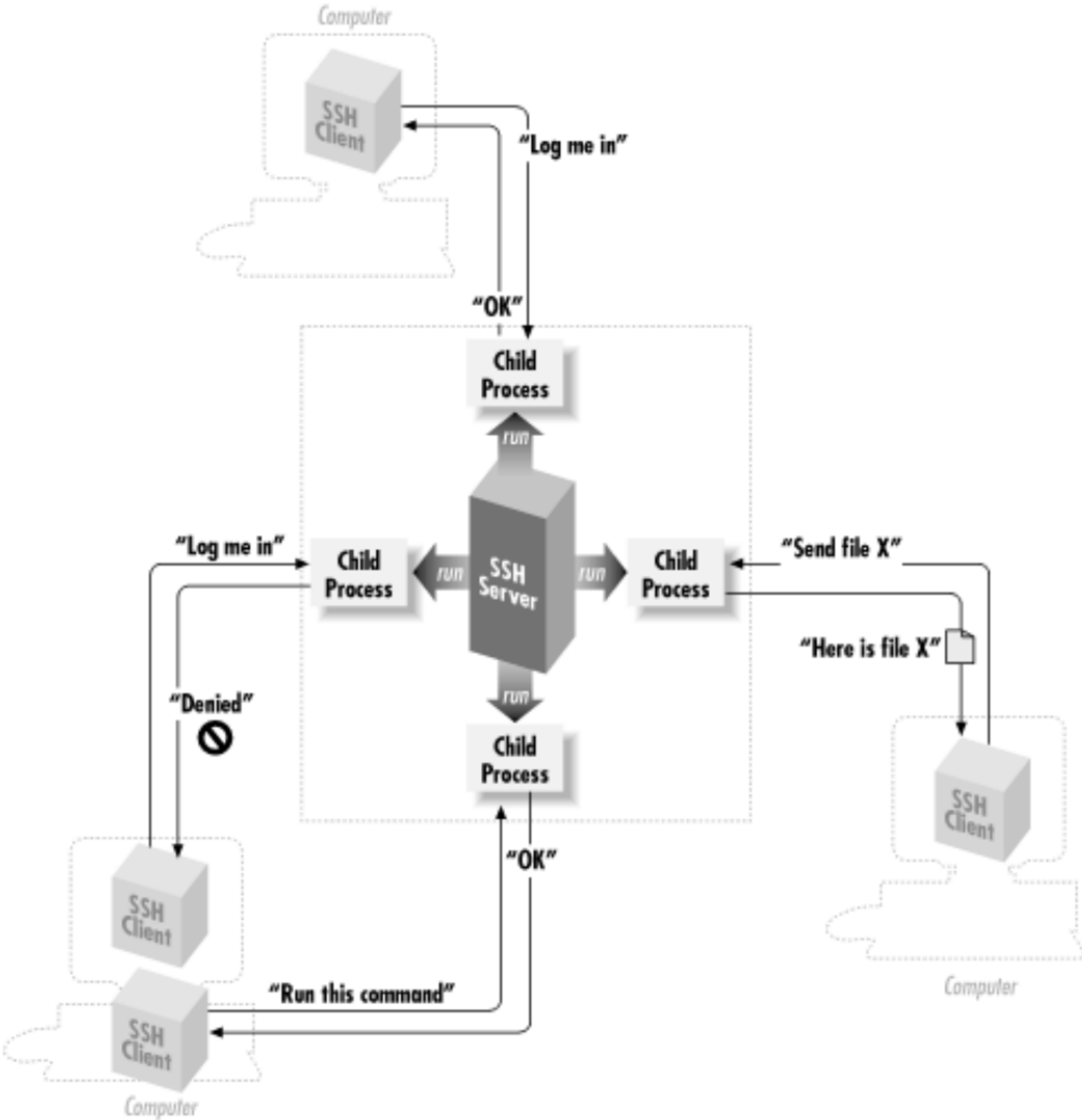

Figure 4.4. *Secure shell protocol [3]*

scripts, access to it can be achieved with new Command Prompt window because the window that was used to launch the virtual machine is not the most stable. For instance, if the virtual machine was created with AMD64 architecture Command Prompt line to gain access would be: "ssh -p 10022 boschrexroth@localhost". This line basically means to get access with SSH protocol to IP-address 10.0.2.2 as local host. Figure 4.5 shows an example of to connect and what is the folder structure like. [11]

The access from the IDE in to the virtual environment folder structure requires the setting of the SSH keys in the environment. Keys can be described as digital identity which are commonly considered as pairs. The pair consist of the public key and the private key. SSH client uses the private key to prove to the SSH server that it is really you that is trying to connect to it. Public key is rather self explanatory, it is a public key that is placed into the server machine with corresponding account and during the authentication procedure

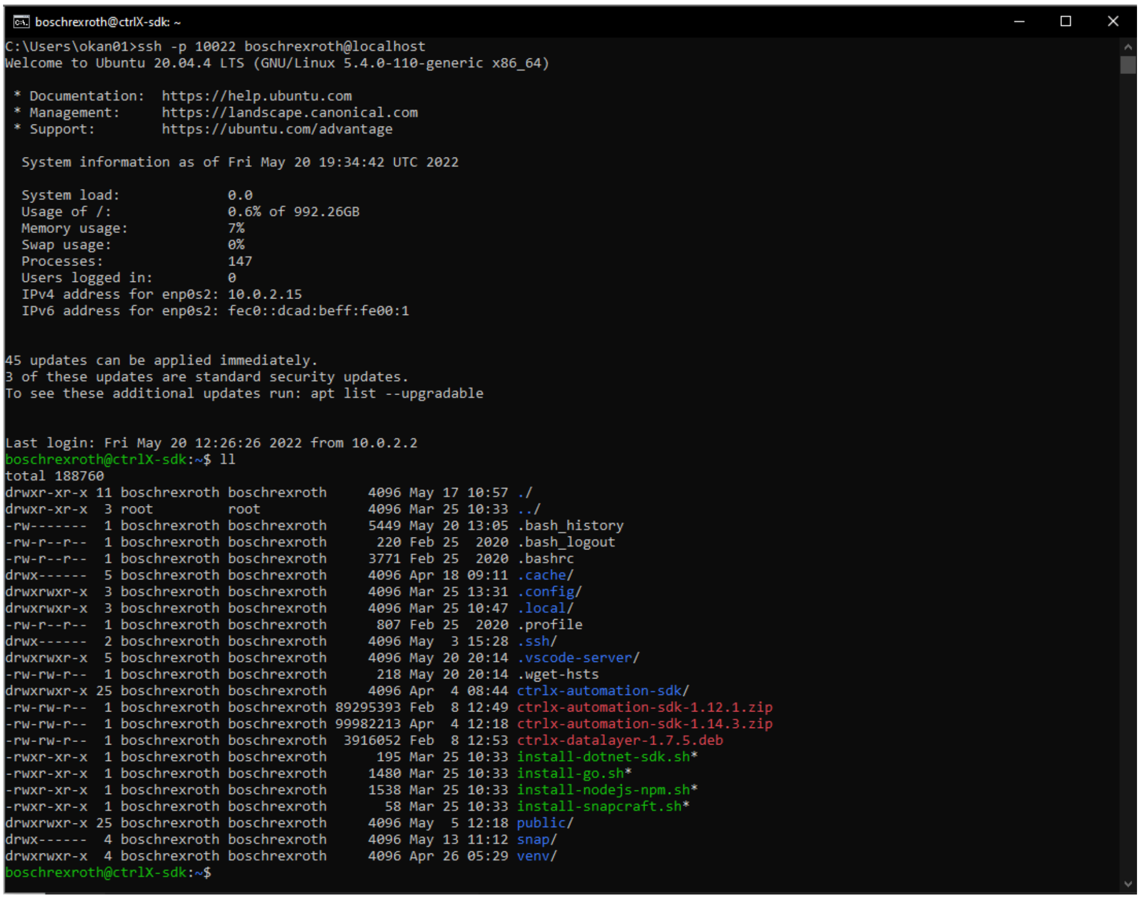

Figure 4.5. *Secure shell connection to the QEMU virtual machine environment*

the server and the client are checking between each other that the cryptographic test between the two machine matches. [3]

Setting the Visual Studio Code to gain access to the virtual machine via SSH protocol is showed in the figure 4.6. By clicking the connection area in 1. and then connecting to the local host (2.) is pretty straight forward. This then provides the access to the same folder structure than in the figure 4.5 as shown in the left side bar (3.) of the figure 4.6. These require that the SSH configuration file in the host system has the correct private key and that the QEMU environment has the correct public key set in the corresponding locations in the systems.

The second connection with SSH is the host's connection to the ctrlX controls. This connection can be done to the CORE itself or with port forwarding or with network adapter to the CORE$^{\text{Virtual}}$. In this thesis the connection was done with network adapter connection to the CORE$^{\text{Virtual}}$. This means that the during the setup process, in the first step in figure 4.2, when creating the ctrlX CORE$^{\text{Virtual}}$ in the ctrlX WORKS it can be decided with what method you want to establish the connection. If network adapter is selected, WORKS creates an new so called TAP Windows Adapter v9 in to the windows network connections and uses it to connect the control with host machine. The method to connect can also be

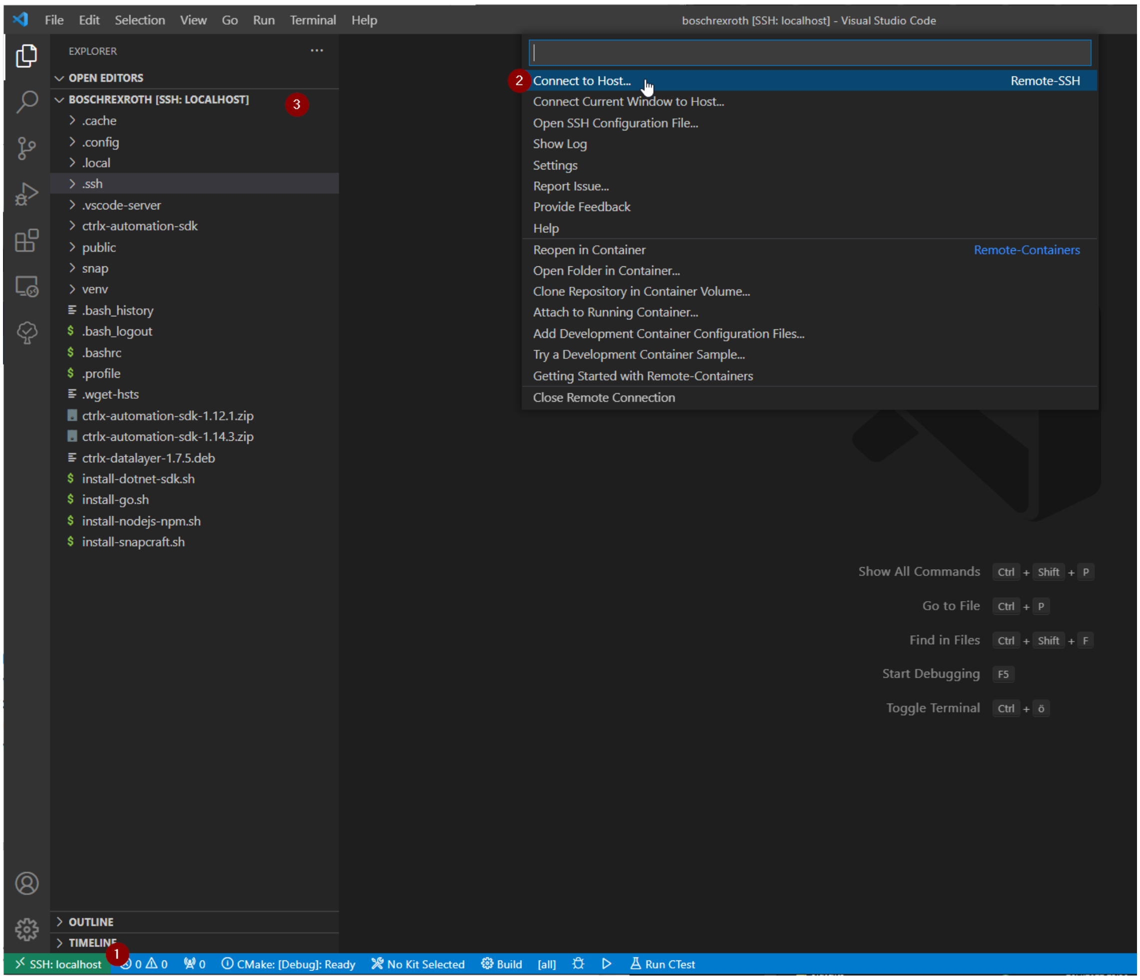

Figure 4.6. *Visual Studio Code with Secure shell connection to the QEMU virtual machine environment in 4.5*

changed later on. [11]

Now that both of the connections are connected to the host machine, it can used as to connect the virtual machine and CORE$^{\text{Virtual}}$ together also. Figure 4.7 illustrates this connection. This connection can help us with the testing phase of the application. Application can be implemented with a debugger that connects itself with a specific port in the network adapter connection to the CORE$^{\text{Virtual}}$ and then it is easy to debug the application directly from the IDE, but more on this in the following chapters (4.3).

Now that we have gone through some of the important steps in the setup phase, let's go through how did all this change during the thesis with the latest 1.14 version of ctrlX WORKS and SDK. Figure 4.8 shows the steps that are now taken when setting up the environment and creating the application for ctrlX.

Bosch Rexroth has develop this App Build Environment that can be added to the ctrlX WORKS as an additional package in the installation phase. This takes care of all the small tinkering steps that were previously required to take if one wanted this environment

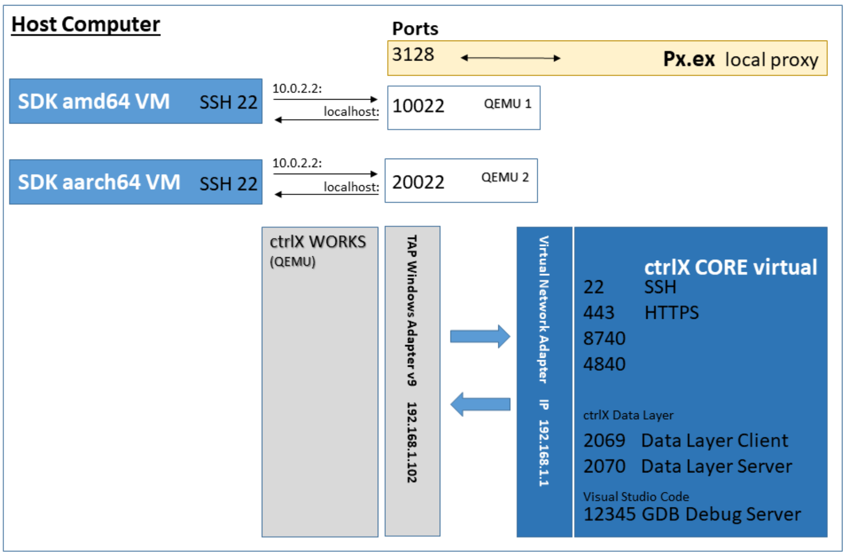

Figure 4.7. *CtrlX CORE^{Virtual} connection to the host and QEMU virtual machine with Network adapter [11]*

as the main development environment. Some steps are still the same in the setup phase, but if this is compared to the previous methods in the figure 4.2, it is much more user friendly and straight forward to do.

Proxy software Px is still required if the host machine is a corporate machine with proxies and of course the integrated development environment is needed to edit the source code. First boot takes some time because it sets up the environment, but after that it is easy to just press play on the App Build Environment to start it up and by clicking the SSH hyperlink to gain access to the virtual machine. No more need to open new Command Prompt windows and launch the .bat files to boot up the environment etc.

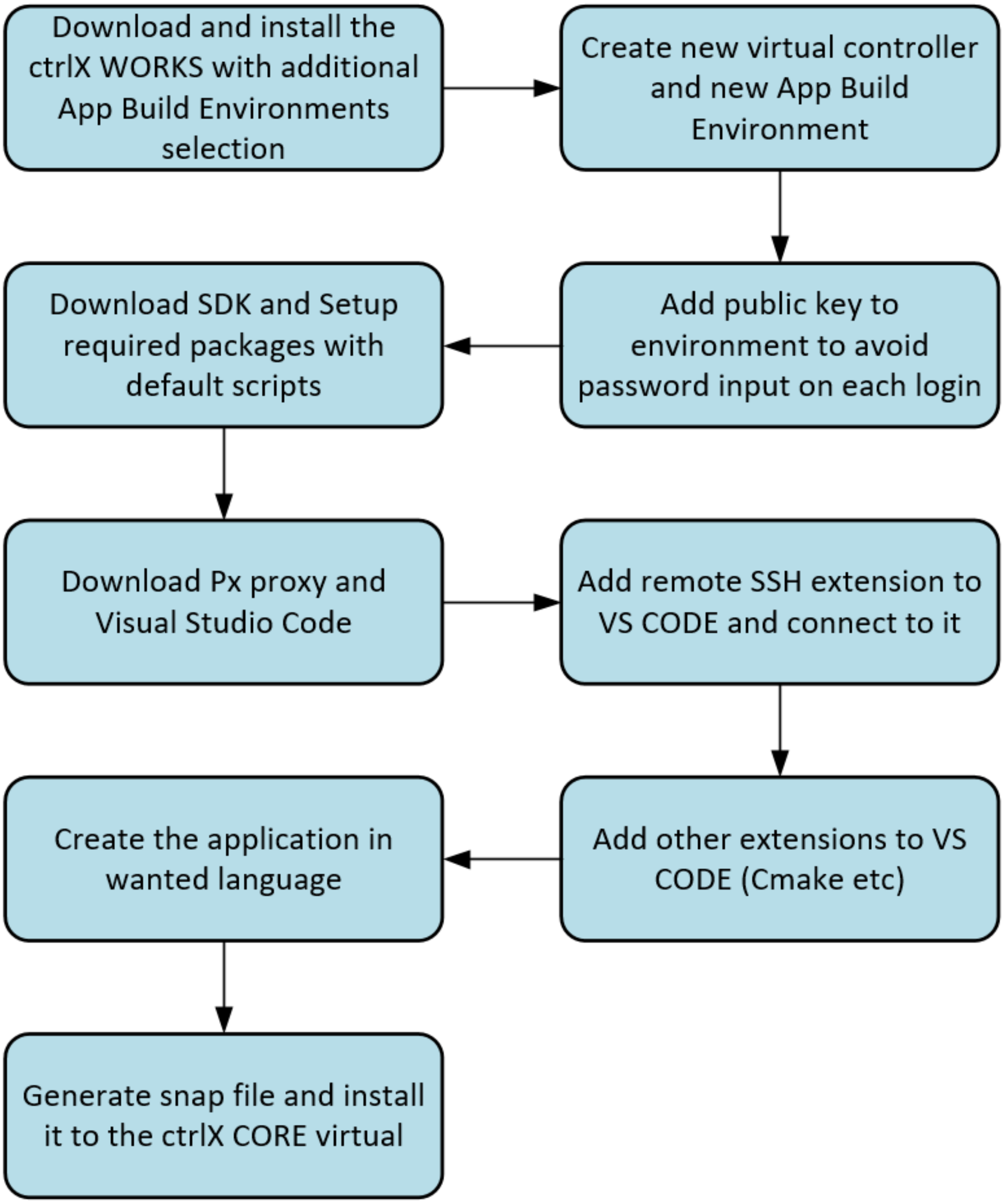

Figure 4.8. *New QEMU environment setup steps*

4.2 Overall description of the application

The example application for this thesis was created with C++ and Python as was mentioned earlier in the thesis. However the functionality of the program stayed mostly the same even though the Python version had updates added to it. These updates consist of mostly making the program structure so that it is more versatile and can be more easily adjust for different use cases, like a standardized application interface. Figure 4.9 describes the applications functionality from the beginning to the running state and some functions around that state.

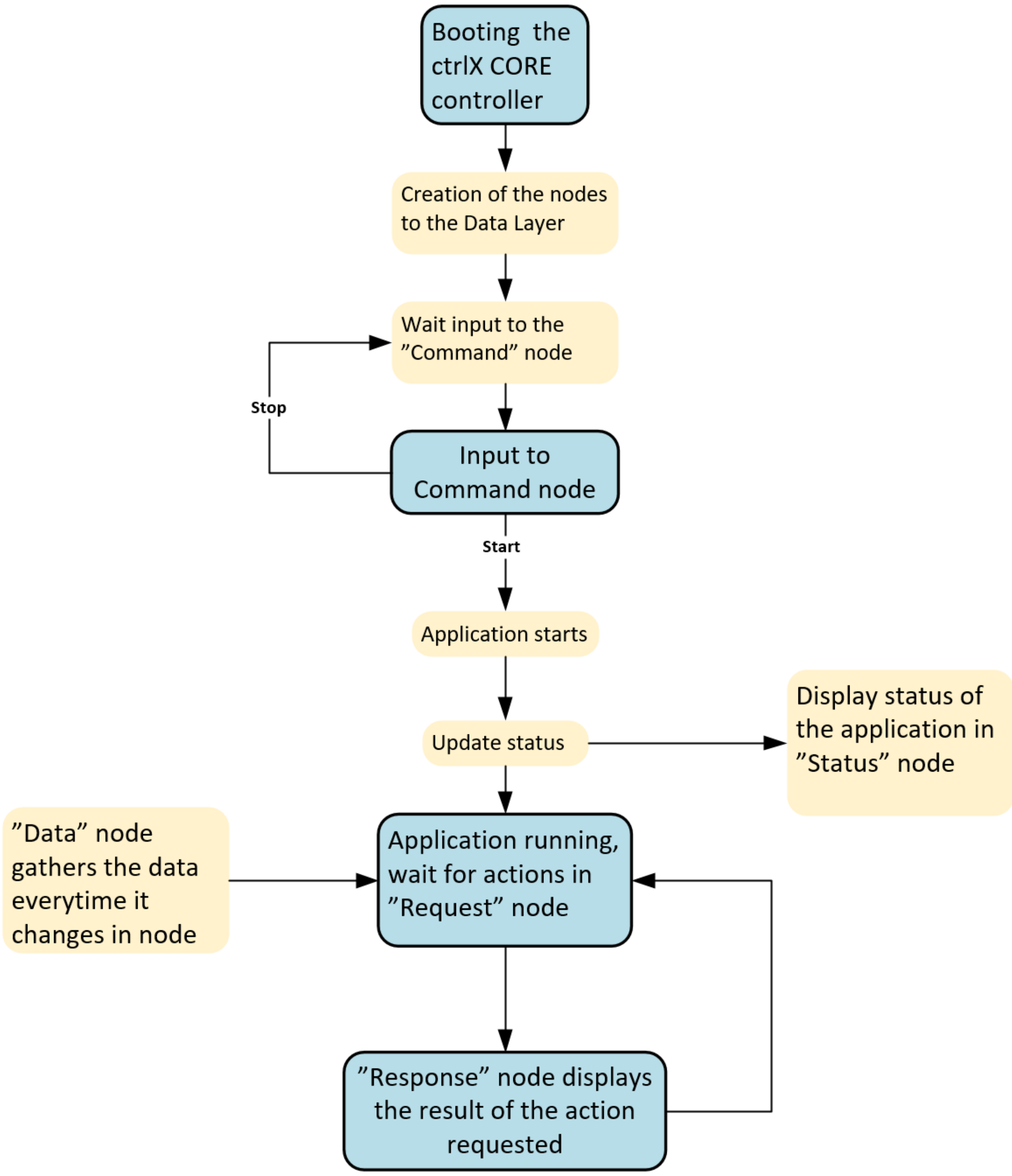

Figure 4.9. *Overall description of the created application*

Let’s focus on some of the steps in the programs flow and provide more insight to some of the functions. Before that we need to freshen our memory from the earlier parts of this thesis, the figure 4.10 reminds how it was intended to use outside interfaces to the snap application. This was illustrated in the proposal phase of the thesis and it was also then later on implemented as a functionality in to the snap application.

When the CORE is booted, the application reads a JSON (JavaScript Object Notation) format configuration file that is placed in the folder structure of the ctrlX CORE. This file holds the general interface structure, such as the node names that are to be created in to the Data Layer. This way it is easy to add or remove nodes from the Data Layer.

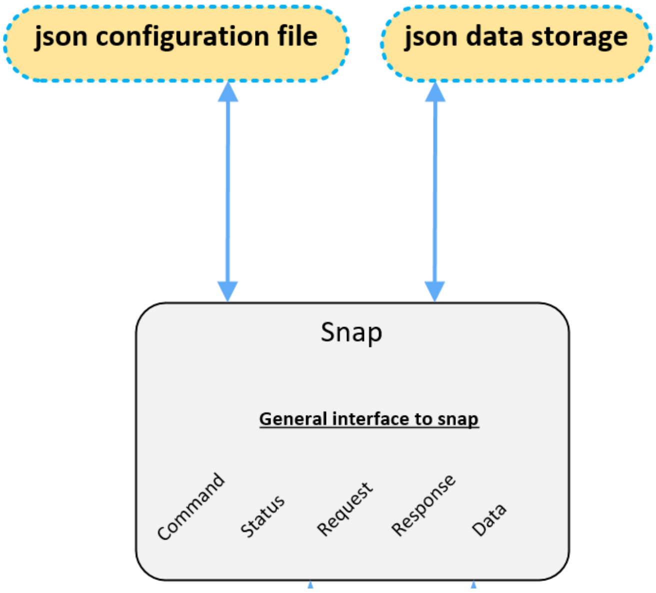

Figure 4.10. *"Outside" connections to the created applications*

Access to this configuration file can be achieved with either from the web interface of the controller or with third party file transport applications that support protocol called WebDAV. One such popular application is WinSCP that was also used in this thesis. WinSCP stands for Windows Secure Copy and it is considered very safe because it also uses SSH protocol to encrypt and decrypt data. [34]

Figure 4.11 shows the folder structure inside of ctrlX CORE and the same structure with WinSCP connection. It is only possible to modify files that are under a folder that you have created. For instance, folder called "Oma" was created for this thesis and this folder holds the configuration JSON etc. It is simple and easy to open and modify files with this WinSCP connection because it uses the same login information that the CORE uses, so if the controller is running, it's possible to make changes to files inside of it.

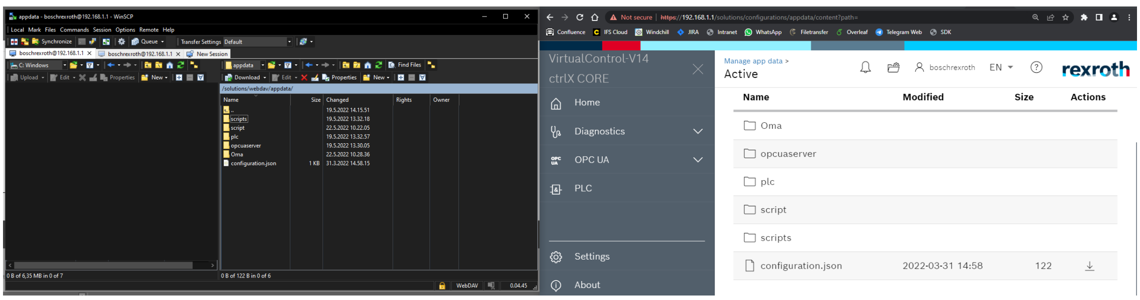

Figure 4.11. *Folder structure in the ctrlX controller (right) and WinSCP connection the same folder (left)*

This same folder is also used as a temporary location to hold stored data. It can also be considered as a JSON data storage (Figure 4.10) because it works as a storage for the data files that created by the snap. Why is it then only considered as temporary if it saves the data to the controllers own storage? Short answer to that is because at the time of this thesis, the CORE controllers would only be capable of having either SD card or USB-stick as a form of storage. This creates an issue where the SD card for instance would fill up quite quickly. If it would be possible to attach a SSD or HDD drives to the controller, then the situation would be different.

All the nodes in the application are defined as string values. This means that a string value is written to the nodes that interact with the program and then the program parses it and makes the corresponding actions. Strings that are entered to the nodes are constructed so that it contains a "key" so the application knows what action is wanted and then after the first key word, the string contains some additional information related to that first string value. For instance, if a string "Start<separator>datafile.json" is inputted to the "Command" node, application searches the folder "Oma" in the ctrlX folder structure and if the "datafile.json" is found from there, it will start the application and take this file as an active file from where it is then possible to search some data.

Same principles are implemented with the "Request" node. String contains the action and then the additional resource related to that. So the strings to interactive nodes are always like: "<action><separator><additional information>". "Data" node is only used as to receive data into the applications run time memory. The application uses separate JSON parserer that parses the string formatted JSON data into the corresponding containers related to the specific programming language.

The nodes that were created were in string format, but also node values as flatbuffers were researched briefly at the later part of the development work. Flatbuffer is a cross platform serialization library that also supports C++ and Python programming languages. Flatbuffers have few benefits compared to the traditional JSON format; the data that is represented is in hierarchical flat binary buffer so that is can be accessed directly without parsing/unpacking unlike JSON and it was originally designed for games so it is fast and memory efficient also. There are also other benefits of implementing a flatbuffer node to application, but it also has some costs to it. It is more time consuming to create and it requires slightly more advanced coding to get the most out of it. [16]

Flatbuffers were researched in this thesis only with Python language. To create a flatbuffer node in Python, an external tool is required. The binary buffer file is created with "flatc" compiler which generates two different files that are required to create the node. A binary file out of previously defined schema file and then a helper class that is used to access and construct serialized data. After this "flatc" command has been run and the files are created, it is possible to construct a flat binary buffer with the help of flatbuffer builder inside of the applications code and use Data Layer provider functions to register the flatbuffer node to the Data Layer. [16]

This would be a good alternative for the "data" node in the application that is created for this thesis, because also the ctrlX PLC side can construct flatbuffers so it might be faster and more efficient to use these instead of string values. Python API of ctrlX also supports flatbuffers and there are good examples of how one can implement flatbuffers to ctrlX applications. Flatbuffers were implemented also for this thesis and figure 4.12 shows how it could look like for our case. But unfortunately as it turns out in chapter 5, this setup had some challenges and will require more development in the future if this method is wanted.

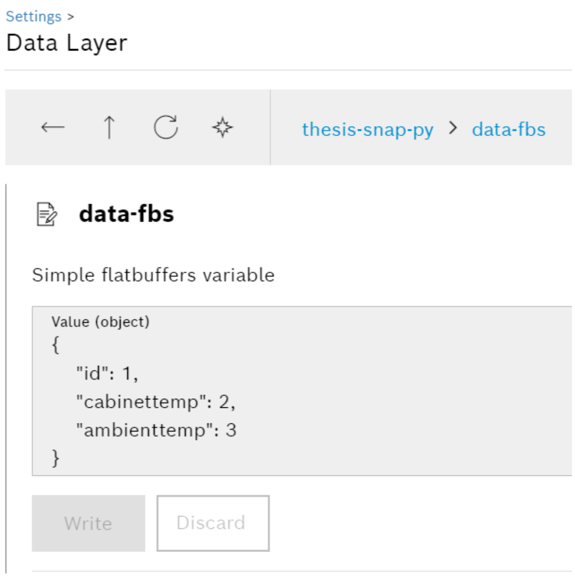

Figure 4.12. *Example of how flatbuffer node could look like in the Data Layer*

4.3 Comparison of implementations with different programming languages

Differences between the two programming languages and their implementation is quite significant. Both API's, Python and C++, of the ctrlX have developed a lot and now contains lots of useful functions that could be used in different implementations. It will be showcased that how the functionalities differentiate between the languages and also how are some functions improved with different versions of SDK.

Both implementations have been constructed using the sample programs found on the SDK package. In the samples there are showed how one could implement application that does Data Layer client operations, like reading and writing from the nodes. Also there is an provider sample which demonstrates how Data Layer connections is established. [11]

The implementation that was done for this thesis was a combination of both, provider and client samples. Provider to create the nodes to the Data Layer and the client for running the basic functionality of the wanted program. Table 4.1 lists all the files that were used in the implementation and shortly a description of what is the function of that specific file.

C++ files	Description	Python files	Description
main.cpp	Creates nodes using provider.cpp and then calls the functions in client.cpp in an endless loop, also hosts containers that would be destroyed if used in client.cpp	main.py	Creates nodes using my_provider_node.py and then calls the functions in call_datalayer_client.py in an endless loop
provider.cpp	Defining a class which creates the Data Layer nodes	my_provider_node.py	Defining a class which creates the Data Layer nodes
provider.h	Defining a class which creates the Data Layer nodes	call_datalayer_client.py	Defining a class which manages the Data Layer client access and runs required functions for application to work
client.cpp	Defining a class which manages the Data Layer client access and runs required functions for application to work	ctrlx_datalayer_helper.py	Auxiliary methods to create ctrlX Datalayer client and provider connections to ctrlX CORE devices
client.h	Defining a class which manages the Data Layer client access	snap_helper_func.py	Contains "project specific" functions used in the program
CMakeLists.txt	Description of source files and executables for compiler	setup.py	Define required packages and files for Python interpreter
Snapcraft.yaml	Define dependencies for snap	Snapcraft.yaml	Define dependencies for snap

Table 4.1. *Description of main files that are used in both implementations*

As said earlier the implementation was done slightly differently in Python than with C++. This was due to the fact that the C++ was an initial version and received feedback to which was then changed for the Python version. Also the C++ version was done with SDK 1.12 and the Python code was done with SDK 1.14 version so it got slightly different approach from the sample side. Let's compare the pseudo codes of the two main functions from C++ and Python implementations. This will showcase how these two implementation differ from each other.

```
// C++ main.cpp pseudo code
Create new provider datalayerProvider
CALL provider class function datalayerProvider->RegisterNodes()
Initialize containers to hold data

WHILE Provider is connected
        Construct new client instance datalayerclient

        // Run takes map as an input and also returns map from "Data" node
        CALL class function dataLayerClient->Run(std::map) return std::map

        IF data in "Data" node is changed
                save values to std::map which is given to Run() function
        ELSEIF remove data
                empty std:maps

        Stop and Delete client instance

        IF provider connection failed
                print error and try reconnecting

        Sleep for 1 second
END WHILE
Stop Provider and delete nodes
```

```
//Python main.py pseudo code

create Data Layer Provider instance
create Data Layer Client instance

IF Client OR Provider is disconnected
        stop system and try reconnecting

Start Provider
Read node names from json file
CALL provide_string(node names) function to register nodes

IF Provider is connected
        initialize class DL Client with DL client instance

        WHILE DL Client is connected
                CALL Client class function Run()
                Sleep for 1 second
        END WHILE
Unregister nodes and stop Provider
```

As illustrated above, the C++ implementation uses the main function to hold the containers for data that needs to be saved or hold without destroying it. This is because with C++ the client instance is created again in every loop, it erases the memory held by the client every iteration so if one needs some data that is needed to be used again later, it is required to be saved somewhere else. The case is very different with Python because the instance is not created again after the program has entered the endless while loop. It can hold the memory that the client instance allocated and therefore the main function does not need any additional containers.

These changes that were done to the Python version were intended to make the program more versatile and easily modifiable. With C++ it is required to specify the construct of the std::map container. Also because it was given to the function, it will require a lot of modifications to many functions to change the data structure for example. Python dictionary is versatile and replicates also the JSON like format in its structure so the data is therefore easy to modify and because there is no need to save it to the main function, it mitigates the changes to files if there is a need to change the functionality.

These changes in a way that the applications main program operates also effects the functionality of the "Run" function in the Data Layer client file. The "Run" function utilizes different functions that are created to "monitor" changes in the created nodes. Because it is called in a so-called endless loop, it will go through the same steps in every iteration and based on the given values to nodes, do some different functions.

```
//C++ Data Layer Client Run function

Create Data Layer client connection
Ping the connection

Read configuration data into C++ containers from config.json

Initialize temporary std::map
Read data from "data" node to std::map

Read "Command" node and execute functions

IF status is executing
        read "Request" node and execute functions

IF delete data
        return std::map with init values //main function clears std::map
ELSE
        return temporary std::map
```

```
//Python Data Layer Client Run function

Authenticate token
Ping the client connection

Read data from "data" node to dictionary
Check if data is changed

IF data changed
        append data into main dictionary
ELSEIF main dictionary size > threshold value
        save data into json file
        clear data from main dictionary

Read "Command" node and execute functions

IF status is executing
        read "Request" node and execute functions
```

Above pseudo code examples illustrate the "Run" functions functionality in the developed applications with C++ and Python. Overall idea is the same in both but the handling of the main container for data is handled in the client side functions in call_datalayer_client.py file where as in C++ these are handled in the main.cpp file. Also with Python there is an added feature to monitor the actual size of the dictionary, so that its size does not get too large for either the ctrlX CORE itself or for the temporary storage that is holding the JSON files inside of it.

Data Layer also provides default nodes that are implemented by Bosch Rexroth so that it is possible to monitor certain aspects of the ctrlX CORE. For instance the RAM (Random-access memory) or the CPU usage are showed in the DL from where it is possible for applications to read that information and use it some how. If application that was created for this thesis is considered, it would also be relevant to use them to monitor some aspects of the CORE, but this was not implemented yet. That would be good future improvements task for instance.

Python version had some beneficial default functions added to the API of ctrlX. For instance, reading a string value from a node in Data Layer is implemented as a default. There is possibility to get other variables also, but for this thesis the string value was especially beneficial. With Python, reading a string value from node is done in two steps; Calling a Data Layer client function called read_sync(address), which returns an variant type data from given address and then using a function get_string() for that variant which generates a string variable that can be used in the application. There was no such function found for C++ in the API, so after the readSync function, the variant type must be converted to string using combination of two functions, getData and getSize.

Debugging the created application is also very crucial in software development. Debugging is step in testing phase of the software where all potential errors and bugs are tried to be removed so that the application does not crash in any situations. Due to the fact that the programming languages are executed differently, compiled and interpreted languages, the debugging is also done differently in snap environment.

C++ debugging is done with additional tools called, gdb server and gdb debugger. Fortunately the snapd, the background service that manages snaps, has a built-in tool that provides a gdb server and is therefore running in the remote target. The gdb debugger then needs to be configured into the files of the application so the snap can package them correctly. This is quite complicated setup, but there are good examples and instructions on how this is done in the GitHub page of the Software Development Kit. [11]

Python debugging is done with an application called debugpy. The debugger is also implemented into the configuration files so that snap can correctly package everything together. Both of these implementations of debuggers are utilizing a remote debugging functionalities of the Visual Studio Code. After the application is installed to the ctrlX CORE^{Virtual}, application is stopped and started in a debugging mode so that it connects to a different port which then can communicate with VS Code and the debugging can be started in the IDE. [11]

5. RESULTS

This chapter describes the created applications functionality overall in the testing environment and answers whether or not the the proposed development methods described in the chapter 3 were successful in real-world development cases. This chapter is divided into two sections; first section describes the tests done and the functionality of the created application. Second section discusses whether or not previously proposed methods are good fit for industrial settings to develop applications.

5.1 Testing results and functionality of the created application

The features that were intended to be implemented into the application were almost all done. Some ideas changed along the way and are now slightly differently implemented. There are some errors that appear occasionally, but if the commands and data that are entered to the nodes for the application are defined with correct format, the application will execute properly. Figure 5.1 shows the created applications in the ctrlX environment.

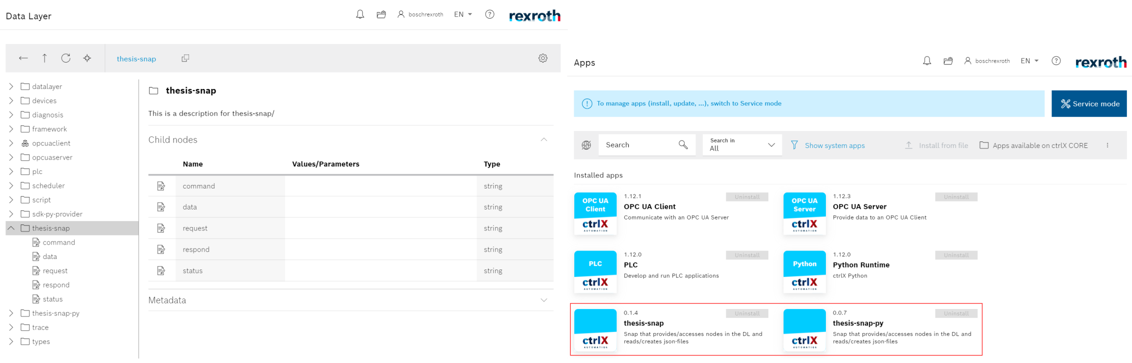

Figure 5.1. *Apps installed in the ctrlX CORE^{Virtual} and the Data Layer interface to created application*

As an example for this thesis, the data that were entered to the "data" node in the Data Layer sustained the following JSON form all through out the testing:

```
{"ID":"1","cabinetTemp":"29","ambientTemp":"22"}
```

Of course the values could be changed, but the key names stayed the same. This was intended to simulate a situation where some application (PLC) writes gathered temperature values from field devices into the Data Layer node which then can be used later on in monitoring the system etc. One of the commands that was created was a save function to saves the data gathered from the "data" node into a single file. Application is meant to be used as a temporary database to hold the data before it can be saved into more permanent storage if that is necessary. Created functionality replaces the "ID" key-value pair in the structure and saves the current date and time to the file so the file would look something like this for instance:


```
{
    "data": [
        {
            "cabinetTemp": 29,
            "ambientTemp": 22,
            "date": "2022-05-29T16:14:04"
        },
        {
            "cabinetTemp": 30,
            "ambientTemp": 22,
            "date": "2022-05-29T16:14:12"
        },
        {
            "cabinetTemp": 28,
            "ambientTemp": 22,
            "date": "2022-05-29T16:14:19"
        }
    ]
}
```


Also other functions were created to retrieve the data from the file or from the container that was hosting the data entered to the "data" node in side of the application. These functions included finding a minimum, maximum and average values of certain data set and also to retrieve longer JSON format data based on search terms that were added to the command in the "request" node as additional command. Table 5.1 describes these functionalities and also others that were created. The "Command" node that basically 4 types of arguments, three for starting the application and one for stopping it. "Request" node can also be used to save data and delete it as mentioned previously.

Possible Command node commands	Additional parameters
Start	*File name that is wanted to be set as running *Command to load the last used file as running *Command to create new file and set it as running
Stop	No additional parameters
Possible Request node commands	
GET	*Command to either get minimum, maximum or average values from data in "Data" node *Search term that is wanted to be found on active data file that is running
SAVE	*file name where the data from "Data" node is saved
DELETE	No additional parameters

Table 5.1. *Description of commands that can be entered to different nodes*

One aspect that was not successful in this thesis was the flatbuffer implementation that came to surface at the later part of the development work. Problem with flatbuffer as a node was that it was not possible to write any values to it. For instance, if values in figure 4.12 in chapter 4 were to change, it resulted with following error: "Error 400: DL_TYPE_MISMATCH - Can't read type information". This error also happened with the default samples of ctrlX Python so this was reported forward in a meeting with Bosch Rexroth. Maybe in the near future this is fixed, but unfortunately it will not be at the time of this thesis.

At least with the Python version of the application, some form of default interface application was created and is now easier to implement similar functionalities into ones own application for ctrlX CORE. It will require more development to fully create default template application. Time is also required to see what functions are working in long term testing and what are not, this therefore effects also the template module and its structure.

Application has not gone through long term testing, for instance continuous running in industrial environment, so this might provide additional errors that were not accounted for in the simple tests that were done to it. Application might need to go through some modifications if it will be used in continuous testing in actual robot solution, but for the sake of this thesis, the results that were wanted in the first place were achieved.

Overall the application is stable and works as intended. Some quality of life changes to the snap could be done, but for demonstration purposes that these kind of application are useful in the new ctrlX automation platform, this application suites its purpose well.

5.2 Use of proprietary applications in Cimcorp

Thesis was intended to be a starting point for third party applications in Cimcorp. All the different possible environments were studied and found a good understanding on what is the desirable development environment. Table 5.2 describes the tested environments and also shows shortly what are some of the pros and cons about these environments.

Environment	Pros and cons
App Build Environment	*Suggested method to develop applications *No GUI *Easy to setup *Good for debugging *Takes a lot of storage space
QEMU virtual machine	*Tedious to install because lots of settings *Good for debugging *Takes a lot of storage space *No GUI
VMware virtual machine	*GUI *Can be configured to take less storage space than QEMU *Not good for debugging *Simple to setup *Supports multiple achitectures

Table 5.2. *Short comparison of tested development environments*

Sample application was also created and demonstrated for personnel in Cimcorp and Bosch Rexroth as a introduction that what might be our implementation cases and also possibilities that it might give in the future if these snap applications were further developed.

Understanding of snaps and their development environments should be now well enough documented so that it is easier for someone new to start familiarization of applications to ctrlX CORE by themselves and not spend additional time in the setup phases etc. The source code will not be published in any public Git repositories, but it will be saved into internal documentation of Cimcorp.

The environment setup depends on the developers preferences, but now that the App Build Environment is implemented directly to the ctrlX WORKS it is suggested that it will be main environment to develop third party applications for ctrlX CORE. The proposed development pipeline in the chapter 3 is still valid if the project is large and there is a remote builder implemented into the repository or some other automated testing included. Only if the project is rather small then it would be proposed that the development pipeline would be like described in the figure 5.2. This way it is more simpler to just download the snap from the remote machine via SSH connection on the VS Code and install it directly to the ctrlX CORE. Only after everything is checked that they are working, it would be wise to save the source code into another location. This method was also used for the sample applications created for this thesis.

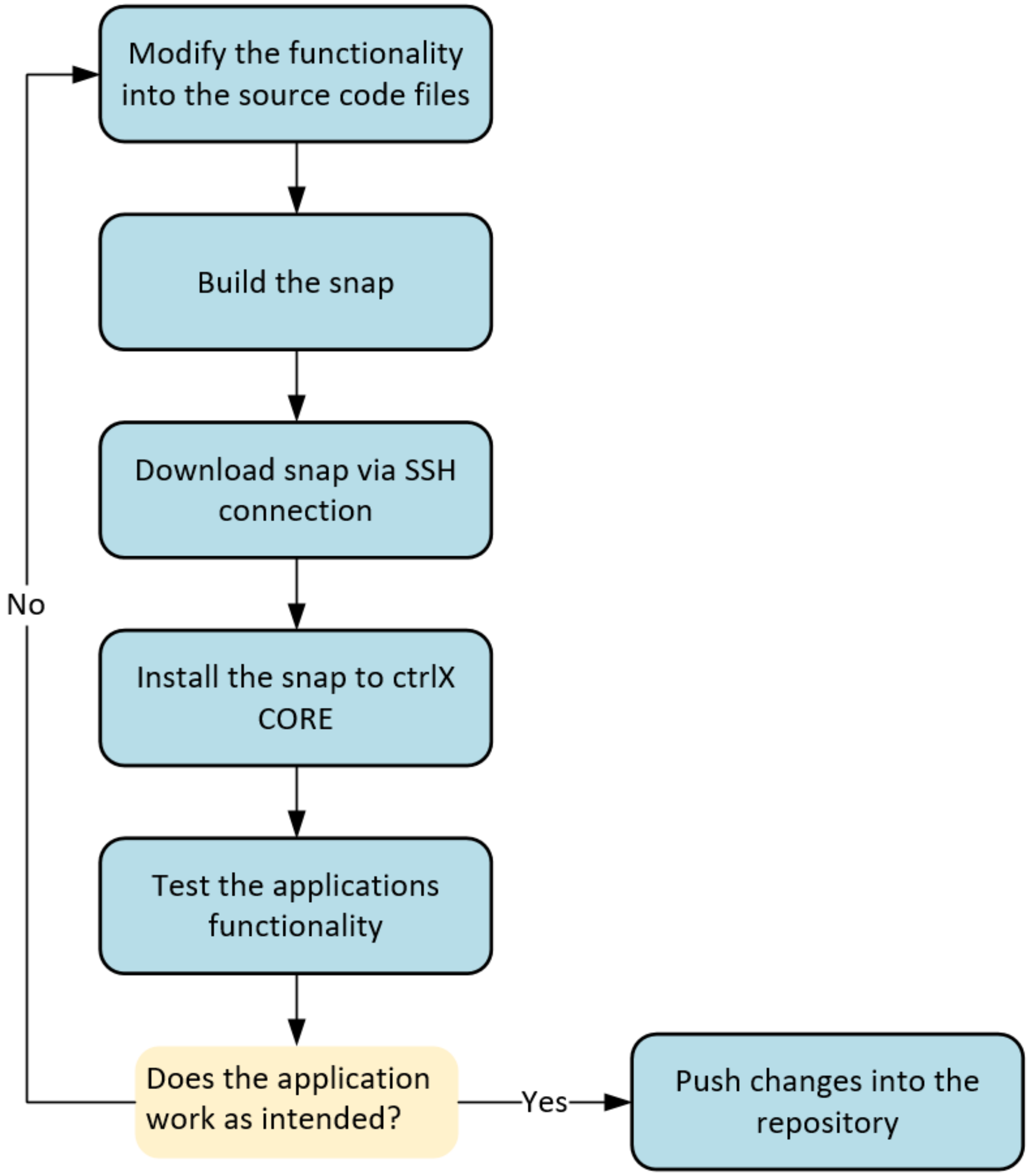

Figure 5.2. *Suggested development pipeline of applications in small projects*

6. CONCLUSIONS

The conclusions chapter provides the final results of the research work. This chapter summarizes the work done in this thesis and discusses the results of work with future in mind. Also the research question assigned in the chapter 1 will be answered and the thesis will be concluded.

6.1 Conclusions

Let's remind ourselves on what were the original research questions at beginning of this thesis in chapter 1 and try to provide an answer to these questions.

- What would be the standardized application interface when considering the usage in industrial automation domain?

The standardized application would be like the template model of default application that then could be utilized in new projects where there are new applications developed. Figure 3.6 in the chapter 3 describes this function quite well. The Python version of the sample application had features that could be used as a default module. The interface to the application should be somewhat similar depending on the use case of course. Application should be able to be started, stopped and resetted with similar commands every time. Data request should have similar commands to it also, but with different additional parameters depending on added features. Status of the application should be showed in the same manner every time etc. These features were implemented to the sample application, but it must be emphasized that it is hard to create such application that would bend into many forms because there are so many possible use cases that require different features to be added so it might need some modifications along the way.

- How to use database functionality with proprietary snap application and how well does it perform in CtrlX environment?

In the short term testing of the application, the folder structure of ctrlX CORE$^{\text{Virtual}}$ was sufficient as a database function. It worked well, did not have any slowness issues and did not fill up completely. Only with long term testing when there are hundreds or thousands of data point gathered and saved to the JSON files in the folders of ctrlX CORE, it will show us how does it really perform. It will probably fill up the SD card of the CORE and then it will show some slowness issues etc. An attempt were made to prevent this filling up by creating a functionality in to the application so that every time there is over 10 data files in the folder, it will overwrite automatically the first one if 11th data file is created. Further development would bring us some additional feature that we could retrieve these data files from the folders of ctrlX CORE and save them into another permanent place. Or it could bring us large SSD-cards that are attachable to the ctrlX CORE and be used as a database.

- What is the most convenient method for building a production version of a proprietary snap application?

Whole thesis hovers around this question, but in the section 5.2 of chapter 5, this question was answered in more detail. Basically the convenience comes from using the suggested software's and environments so that everything works properly together as they have been designed by Bosch Rexroth. With additional following of modern development practises, like the use of Git, it is very convenient to build application based on snap technology for ctrlX CORE.

6.2 Future

Seemingly in the near future, the Gantry robots and so-called standard devices that Cimcorp is producing for intralogistics applications are to be transferred to the ctrlX family completely by hardware and software side if everything goes according to the plans. Therefore there may be an opening to test out the application created in thesis in actual ctrlX CORE. The near future will probably hold continuous testing of the application with actual ctrlX devices inside the technology development department of Cimcorp.

Testing results with actual robots will be effecting the long term use of the created application. If everything goes as expected, the application example created for this thesis might even be used in some projects that are delivered to customers around the world. The use case could be for additional analytics or some data monitoring with slight modifications based on the end use case of course.

Flatbuffer data type node that was mentioned earlier as not successful implementation might get tested more later on. Also some additional features should be implemented into the application, like the hardware monitoring of the ctrlX CORE. As it was stated previously, it is possible to gain access of CPU and RAM values of the actual controller via the Data Layer to build self-monitoring capabilities. so all this is a pretty good future development target to implement some monitoring features in a snap application.

REFERENCES

[1] Magnus Åkerman. *Implementing Shop Floor IT for Industry 4.0*. June 2018, p. 2. URL: https://www.researchgate.net/figure/The-automation-pyramid-according-to-the-ISA-95-model-The-five-levels-0-5-are-defined_fig2_326224890.

[2] *ARM website*. 2022. URL: https://www.arm.com/ (visited on 04/2022).

[3] Daniel J Barrett et al. *SSH, the Secure Shell: the definitive guide*. " O'Reilly Media, Inc.", 2001.

[4] Carl Bostrom. *ctrlX CORE: REST API*. 2022. URL: https://developer.community.boschrexroth.com/varuj77995/attachments/varuj77995/dcdev_community-dcae-core/88/1/REST%20API%20ctrlX%20CORE.pdf (visited on 05/2022).

[5] Scott Chacon and Straub Ben. *Pro Git, Second Edition*. eng. 2nd edition. Apress, 2022. URL: https://git-scm.com/book/en/v2.

[6] Jack Chan, Ray Chung, and Jack Huang. *Python API Development Fundamentals*. eng. Packt Publishing, 2019. ISBN: 9781838983994.

[7] *CIMCORP optimizes gantry robots for intralogistics with ctrlX AUTOMATION*. 2021. URL: https://apps.boschrexroth.com/microsites/ctrlx-automation/en/news-stories/story/cimcorp-optimizes-gantry-robots-for-intralogistics-with-ctrlx-automation/.

[8] *CMake Documentation*. 2022. URL: https://cmake.org/cmake/help/latest/ (visited on 04/2022).

[9] *ctrlX AUTOMATION*. 2021. URL: https://www.boschrexroth.com/en/xc/products/product-groups/electric-drives-and-controls/ctrlx-automation (visited on 03/2022).

[10] *CtrlX Automation - Two step ahead*. Dec. 2019. URL: https://www.boschrexroth.com/documents/12605/24070539/ctrlX+AUTOMATION+-+Brochure/318ca8a7-0995-8e5e-cfba-664ca9a6cd36.

[11] *ctrlX AUTOMATION Software Development Kit*. 2022. URL: https://boschrexroth.github.io/ctrlx-automation-sdk/overview.html (visited on 05/2022).

[12] *CtrlX CORE - OPC UA Server APP*. 2022. URL: https://developer.community.boschrexroth.com/t5/Store-and-How-to/ctrlX-CORE-OPC-UA-Server-App/ba-p/13271 (visited on 05/2022).

[13] *ctrlX Data Layer*. 2021. URL: https://developer.community.boschrexroth.com/t5/Store-and-How-to/FAQ-for-ctrlX-Data-Layer/ba-p/21236 (visited on 03/2022).

[14] *ctrlX MOTION*. 2022. URL: https://apps.boschrexroth.com/microsites/ctrlx-automation/en/portfolio/ctrlx-motion/ (visited on 03/2022).

[15] *ctrlX WORKS*. 2022. URL: https://apps.boschrexroth.com/microsites/ctrlx-automation/en/portfolio/ctrlx-works/ (visited on 03/2022).

[16] *Flatbuffer GitHub online documentation*. 2022. URL: https://google.github.io/flatbuffers/ (visited on 05/2022).

[17] Emily Freeman. *DevOps*. eng. 1st edition. For dummies. Hoboken, New Jersey: For Dummies, 2019. ISBN: 1-119-55224-9.

[18] Vardan Grigoryan and Shunguang Wu. *Expert C++: Become a Proficient Programmer by Learning Coding Best Practices with C++17 and C++20's Latest Features*. eng. Birmingham: Packt Publishing, Limited, 2020. ISBN: 1838552650.

[19] Aaron Hand. *Bosch Rexroth Reinvents Its Automation Approach*. 2020. URL: https://www.automationworld.com/home/article/21136247/bosch-rexroth-reinvents-automation.

[20] *Introduction to x64 Assembly*. URL: https://www.intel.com/content/dam/develop/external/us/en/documents/introduction-to-x64-assembly-181178.pdf (visited on 04/2022).

[21] *JSON online documentation*. 2022. URL: https://www.json.org/json-en.html (visited on 05/2022).

[22] Ampomah Ernest Kwame, Ezekiel Mensah Martey, and Abilimi Gilbert Chris. "Qualitative assessment of compiled, interpreted and hybrid programming languages". In: *Communications* 7.7 (2017), pp. 8–13.

[23] *OPC UA online documentation*. 2022. URL: https://opcfoundation.org/ (visited on 05/2022).

[24] *Px Github online documentation*. 2022. URL: https://github.com/genotrance/px (visited on 05/2022).

[25] *QEMU online documentation*. 2022. URL: https://www.qemu.org/docs/master/about/index.html (visited on 05/2022).

[26] *REST API Tutorial online documentation*. 2022. URL: https://restfulapi.net/ (visited on 05/2022).

[27] Jérémy Robert et al. "Minimum cycle time analysis of Ethernet-based real-time protocols". In: *International Journal of Computers, Communications and Control* 7.4 (Aug. 2012), pp. 743–757. URL: https://hal.archives-ouvertes.fr/hal-00714560.

[28] Kulbir Saini. *Squid Proxy Server 3.1: beginner's guide*. Packt Publishing Ltd, 2011.

[29] *Setuptools documentation*. 2022. URL: https://setuptools.pypa.io/en/latest/setuptools.html (visited on 05/2022).

[30] *Snapcraft online documentation*. Apr. 2022. URL: https://snapcraft.io/docs (visited on 04/2022).

[31] *Snapcraft website*. Apr. 2022. URL: https://snapcraft.io/about (visited on 04/2022).

[32] Mark. Summerfield. *Programming in Python 3 : a complete introduction to the Python language*. eng. 1st edition. Developer's library. Upper Saddle River, NJ: Addison-Wesley, 2009. ISBN: 1-282-64719-9.

[33] *Virtualenv website*. 2022. URL: https://virtualenv.pypa.io/en/latest/# (visited on 05/2022).

[34] *WinSCP online documentation*. 2022. URL: https://winscp.net/eng/docs/start (visited on 05/2022).