Date of acceptance Grade

Instructor

Open source tools for Linux distribution development and maintenance in corporate environment

Niko Kortström

Helsinki 22.05.2017

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta – Fakultet – Faculty	Laitos – Institution – Department
Faculty of Science	Department of Computer Science

Tekijä – Författare – Author

Niko Kortström

Työn nimi – Arbetets titel – Title

Open source tools for Linux distribution development and maintenance in corporate environment

Oppiaine – Läroämne – Subject

Computer Science

Työn laji – Arbetets art – Level	Aika – Datum – Month and year	Sivumäärä – Sidoantal – Number of pages
Master's thesis	22.5.2017	70 pages + 4 appendices

Tiivistelmä – Referat – Abstract

In this thesis, we will look into utilizing open source software build tools in an enterprise environment. We will aim at providing a complete set of tools starting from developer support and leading to software delivery. We will discuss the different tools that we will use to offer more reliable, easy to use and efficient process of composing software products. Many open source projects will be utilized and we will examine the required steps to be able to successfully operate them in a closed environment. We will also look into providing a completely new base image for in-house cloud platform building. The process of internally composing the operating system from open source components will be discussed in depth.

ACM Computing Classification System (CCS):
Software and its engineering
  Software notations and tools
    Development frameworks and environments

Avainsanat – Nyckelord – Keywords

Cloud computing, DevOps, Software build systems, Developer tools, OS development

Säilytyspaikka – Förvaringställe – Where deposited

Muita tietoja – Övriga uppgifter – Additional information

Contents

1 Introduction

The backbone of any computer program is the operating system (OS) it is executed on. The tools you use for developing and executing your applications depends on what is supported by the operating system. The successful execution of applications is also greatly dependent on the successful execution of operating system software. Errors in the underlying OS can lead directly into errors in the running applications.

For some time now, I have been working on a cloud platform project. Simply put, we work to offer operating system and basic services for Nokia applications running on cloud. We are currently using a Linux [Lin17a] distribution called NetLinux as the OS. While the exact content of NetLinux is not publicly available, most of its packages are included from open source releases without modifications. Many internally developed services that the applications can utilize for effective operation are installed on top of the OS.

We deliver our platform as a disk image. This image is used to launch virtual machines that the cloud applications can be installed on. To clarify, there are three clear layers used: operating system (NetLinux), platform services offered to applications and the running applications.

In the cloud platform project, we have our own fork of NetLinux and many of us work directly on the operating system components instead of the platform services. However, the fork is basically just direct clone of the NetLinux main repository, with regular merges executed after new releases. The reason for using this repository fork is being able to deliver critical changes to applications without having to rely on operating system release schedule.

Having worked with NetLinux before, we have experience on using the open source components and the current development workflow is familiar to us. These skills will become useful when tackling the problems presented in this paper. We are in luck not having to start from scratch in this matter. We trust that we are able to acquire better results in a lesser amount of time when working on a subject that we already have basic knowledge of.

NetLinux distribution is based on Linux From Scratch [Bee16] project. Composing our operating system like this offers some benefits over the most widely used distributions such as compactness, flexibility and security [Wha17].

Building the operating system from source code ourselves is important to us for many reasons:

01 Code changes should be reviewed by one of our developers before including them in an official release.

02 We want to have the possibility of implementing and releasing our own changes quickly when necessary.

03 The build process must be completely in our control to make sure that all the specifications are

met.

04 We must be able to control the release schedule of all the software.

Obviously, there is no perfect approach. Many issues have been identified with the current distribution and the way our platform is built. The operating system is built as a monolithic entity from sources and offered as a binary image to applications that operate on it. This results in a number of issues which application developers have called for improvement on.

The NetLinux distribution is used by many different applications, many different components are required. Application developers would obviously prefer for the operating system to only include components that are required by their application, not the ones used by some other software that they do not have any affiliation with. This is one reason why different applications would prefer different kinds of images to use.

It would be extremely costly for applications to maintain a modified version of the operating system because of:

01 Separate build system needs to be set up that would build modified images for the application in question

02 If one part of the operating system is modified, the whole image has to be rebuilt from the sources

In order to set up a private build system this the application developers would have to fork the distribution, implement the code changes and handle building a version of the entire OS themselves. Constant work is required for keeping the operating system up to date with upstream while making sure that features that they do not want to use, are excluded.

Rebuilding the platform from scratch and delivering it to application developers would require tremendous amounts of effort. Compiling an entire operating system is very time consuming. It is unreasonable to spend time on building all the components that have not been modified.

Another major issue with NetLinux is its development process. Implementing code changes, testing the modified software or debugging current software requires a lot of unnecessary effort. This is what we will be trying to improve.

Software development is by no means a simple task. Finding the points of failure and figuring out the appropriate corrections can be even more demanding. Complicated tasks should never be made more difficult than necessary. Any software development project should aim at using modern tools and processes that are efficient, effective and easy to use.

Our application developers have been pointing out that when a fault occurs in the operating system, they have very limited chances to debug the issue. Even finding the correct point of failure can sometimes prove difficult. If the developers are able to identify the malfunctioning component, they have had major problems with accessing the source code of the component in question. Having the program

code available is obviously essential in being able to track down the bugs and prepare corrections for them.

The inability to access the sources can be caused by not having the access rights or simply not knowing where to look. What we should do, is to provide a simple way for any relevant developer to access the sources of all packages running in our system. After accessing the code and making changes to a package, the developers should also be able to easily install the new version. Adding debug prints for examining the issues more closely or preparing a fix and testing it should become trivial tasks for any developer to perform.

Currently, it can be very cumbersome to test code changes locally even for operating system developers. Compilation has to be carried out manually and the correct configuration and installation paths can be challenging to track down.

These are the kinds of issues that no organization in any field should deal with for a long time. It is essential for customer satisfaction that broken products are fixed as soon as possible. The services that are offered must be reliable and responsive. However, if the staff does not have the proper tools to carry out their tasks, how can they be expected to meet tight deadlines given to them?

We want to divide our software into clear packages to help solve the mentioned challenges. There is already ongoing research on building the cloud platform from RPM packages. While we want to continue assisting with the current progress, we also want to look into offering our developers with proper packaging tools. When implementing changes to the way software products are formed, we want to closely examine the whole process from development to delivery. We will have to find tools that allow the development cycle to be simple and efficient.

Obviously building our products from RPM packages instead of source code requires us to build the packages first. Making this process simple and elegant is essential. Improving on building the OS image should not reflect negatively on our ability to develop new software. This could easily be the case if we left the developers on their own.

To battle this effect, we want to provide package maintenance software that hides the more complicated parts of software packaging. This tool should be available for all the developers working on products that deal with our cloud platform.

New tools must not attempt to replace existing functionality unnecessarily. Learning new things takes time and should not be demanded without grounds. Therefore, tools that we will be developing should offer new functionality that can easily be used alongside the existing tools. Finding out the needed features and implementing them in a way that integrates well into current development process, are things that we must pay special attention to.

In short, we see three possible avenues to choose from when starting to implement the developer tool set:

01 Implement a completely new package maintenance tool specifically designed for our operating environment.

02 Look into the packaging tools used in the open source community and take in use the most suitable one.

03 Partially utilize open source products and adapt these to our specific needs

Our research takes place at a time of change. The development of next major version of our platform has begun, and the operating system is yet to be decided. We do not expect our research to complete in time to be used with the current iteration. Thus, we do not want to make decisions that are affected by our current operating system.

We will make our decisions generally on what we see as the best solution for interacting with RPM packaging in large software projects such as ours. Some characteristics to keep in mind are:

- large and diverse code base
- supporting different storage methods
- integrating the tools to operate with existing systems

We will also explore the future possibilities of our tools. One technology that is known to be gaining more popularity rapidly is containerization. We want to ensure that the tools that we develop will stay useful in the everchanging context that we operate in. From our point of view this means investigating packaging software into Docker images in addition to RPMs.

In addition to streamlining the development process, we should also improve the development platform. Currently the development is conducted either on the NetLinux virtual machine that we want to modify or some external computer from which the code is then transferred to the NetLinux deployment for compilation.

The former approach suffers from the lack of development tools. NetLinux does not contain all the standard software that developers use on their mainstream distributions to develop code. On the other hand, the latter way of making software changes is not as efficient as we would like either. Moving the source code between hosts takes time and adds unnecessary complexity to the development process.

We will aim at finding a way to offer both, the simplicity of native development and the tool diversity plus customizability of remote development. Firstly, we can accelerate development by offering more advanced tools. Secondly, it is self-explanatory that enjoyable working environment enhances productivity. We have noticed that this not only applies to physical environment like office interior, but also the virtual environments we deal with. Having to battle with various unrelated issues can cause great harm to morale.

We will attempt to utilize the new development environment that we will be working on, to find ways

to improve the platforms that the developers use for their daily tasks. The tools available in the development environment should be easily customizable by each user. Thus, all developers would have a way to quickly get access to their personalized development environment within the system that is being developed. We want to find a solution that offers a reproducible, customizable and easy-to-use platform for development.

The corporate environment that we operate on enforces its own challenges and many things that work in the open source community, will not work in our context. We will dedicate a chapter for summing up our findings about the requirements enforced on us by our operating environment. Some differences are ownership of programs, strict development practices and smaller developer base when compared to working on an open source operating system.

The open source community is wide and incorporating some of their practices can help us greatly. Still, we must make sure that we conduct our research with Nokia developers as our target audience. To achieve this, the special requirements of the users must be clear. We must find the right solutions for the specific users of our tools and be able to make their working life as easy as possible.

The goal is to incorporate these development tools into the operating system and also offer the chance to install them on personal development machines. My work will include many programming tasks required to implement the tools. However, research must be our main focus as the approaches should be carefully considered before making decisions.

We should aim at conducting both tasks side-by-side. Implementation must be sound and the tools in question should be operational as soon as possible. We want to be able to offer our software to the developers in early stages for their benefit and ours. Starting to introduce new tools and getting feedback at early stages of development are invaluable assets. A lot of working hours can be saved when the features that the users prefer to take becomes clear in the beginning of the development.

We hope to be able to have our tools in wider use. We will keep testing the tools throughout the research process but acquiring different views from different users is often the best way of choosing the next steps to take. This will also greatly benefit us in evaluating our own work and promoting our tools in case they are received well.

1.1 Research questions and structure of this thesis

01 How can we simplify the workflow of developers working on our products that incorporate our cloud platform?

02 What kind of convenience tools can we offer the developers?

03 What special constraints does the corporate context of Nokia and closed development environment in general set for the used tools?

04 Can we prepare the chosen developer tools for the future by exploring venues related to containerization? Is what case our tools would become unusable in the future?

05 How can we improve internal storing of used source code to better support the developers' efficiency?

06 How can we prepare for the possible operating system change of our cloud platform?

07 What should a future development platform be like?

08 What changes do we need for our current development setups?

09 What are the befits we can gain by introducing new a development platform?

We will start by taking a quick glance into the DevOps philosophy and discuss how we could benefit from this. Moving towards DevOps is in our sights and embracing it should be enabled by the tools that we use. The acquired information will work as a basis for the rest of our work.

Next, we will explore tools that we want to offer for our developers. We hope to introduce tools that function properly in our specific programming environment and will make the development process simple and efficient.

We will then move onto the basics of containerization. This subject is a hot topic and there is a lot of ongoing research on the subject. For these reasons, we must not exclude it from our research conducted for this paper.

We will also briefly look into, how we should handle open source code that our tools help developers accessing. Current storage approach will be examined and we will present some benefit we could gain by changing it.

To conclude the development process research, we will present moving towards possible new operating system for future platform versions. While this is quite technical task, the workload can be large so the possible avenues should be examined closely before thinking about starting the transition.

Next, we will go through the new development platform approach. We will also introduce how we can complete the work for the build process. These two subjects will be examined as future steps to fully offer the tools needed for the complete development process.

Finally, we will gather the various corporate requirements we have come across during our work. These finding have been used throughout the research process and should be taken to heart for future use. Then we will briefly evaluate our achievements during the research process and conclude the paper.

1.2 Research setting

We will focus our efforts of incorporating open source development tools to examining Fedora software infrastructure. While NetLinux and Fedora are two very different distributions, we are confident that we will find a suitable developer tools for the future. The differences that will affect our work the most are the fact that no package management system is installed in NetLinux and building the OS from source (NetLinux) versus building using binary packages (Fedora).

Every software project incorporates similar build and development tools at some level. This is true for both corporate and open source setting. Therefore, we have a wide variety of possibilities to choose from. OpenSUSE utilizes Open Build Service [ope17a] for building its packages and Debian developer use aptitude and dh-tools to speed up their work . The decision to focus on incorporating Fedora infrastructure is based on multiple different reasons that will be visited many times in this paper.

First of all, we prefer to utilize tools that have been specifically developed to operate on RPM based environments. Secondly, the Fedora tool supply is very large, as we can expect from such a popular distribution. Thirdly, we are confident that Fedora community will offer us good amount of support when it is required.

Our initial review has shown that Fedora development and build process is highly evolved. It fares really well when compared to the tools of other distributions and there is at least satisfying amount of information available about the systems that we want to utilize. No doubt there will be many cases, when we are forced to learn things by reading code or by trial and error. However, it is common to open source software, and software projects in general that not all information is documented. Thus, this caveat is unavoidable. We are also confident that we will be able to learn required things by ourselves when necessary.

At this stage, all of the systems that we attempt to integrate into Nokia development and build processes, will be run on an internal cloud service. We will be able to start virtual machines of different capacity on demand and connect to them via VNC and SSH. The sparsity of available hardware and enormous flexibility we are able to achieve are the main reasons for this approach.

A positive side effect that we will be facing is the need to implement our systems in easily reproducible fashion. Cloud instances can be unreliable in many cases and we must be able to quickly replicate the systems and start them again when failures occur. We encountered this in practice on many occasions during this research process. Without documented and automated process of system setup, we would have encountered enormous delays as we would have had to redo a lot of the work that we had completed earlier.

In addition to the remote systems that we will be setting up, the development environments that we will be using for test and demo purposes, also reside in the cloud. Similar reasoning can be applied to this choice, perhaps even more strongly than before. Setting up the development tools must be extremely simple. This task will have to be completed by all users, in contrast to the remote build systems

where it is enough that the administrator has set up the service. Therefore, it is highly useful that we will face frequent reinstallation. If the installation process that our tools require is cumbersome, we will want to simplify it as soon as possible to save our own and other users' time.

2 State of the art

At the time of writing, the most sought after way of developing and delivering software is through DevOps principles. This approach is being discussed widely and many organizations are seeking to transition using this practice.

DevOps is a promising new way of providing customers with software products. It covers the software delivery process ranging from planning to development, verification and delivery. In recent years this term has gained wide attention and attempts to adopt DevOps principles have risen immensely over the past few years [Wei16]. There are already architecture modes and platforms being designed with precisely DevOps in mind [DJG16, GCG15].

There seems to still be some ambiguousness present in defining DevOps as stated in [JBP16]. We are going to follow definitions from just a few papers to keep the terminology simple and straightforward. To still acquire a wide view of the research conducted on this development practice, we will want to examine papers from the early days of the topic, but also the research containing the most recent results and views. In short DevOps entails quick and agile development of software and the business processes that enable this [EGH16].

There has been wide interest in the IT industry to adopt the DevOps principles. In telecommunications, being left behind has been a trend for long. This should definitely be improved on and the newest innovations should be adopted when benefits can be gained. In our research, we will attempt to start strongly supporting the possibilities of moving towards DevOps practices.

The benefits of this approach have already been seen in practice as shown by [BFH16, VKK14] and we strongly believe that we could gain a lot by adopting this way of producing software. We want to be able to decrease the amount of time it takes for us to deliver our software products to customers [DJG16]. Benefits of testing and delivering software often have also remarkable. The issues caused by finding bugs can be limited by being able to provide lightning fast fixes.

Delivering features and fixes in a faster pace will allow moving forward with new technology much faster and keeping the software operational at all times. In the telecommunications industry, availability is essential and big defects can even cause life threatening situations in some cases. Minimizing the time from the point is receiving a software development task, to having it included in the customer's copy of the system, benefits us and our customers greatly.

2.1 Defining DevOps

DevOps is about integrating or at least tightening the gap between different stages of software delivery [EGH16]. These stages can be considered to be software development, verification, delivery and operations. These four stages have traditionally been clearly separated from each other. The type of work

has been strictly bounded by the team you work in. To be more accurate, the task assigned for your team has defined the type of tasks you will be working on. Software development personnel have focused on software development and operations personnel have focused on operating the delivered software only.

Working closer together with teams from all stages of software production is essential. Proper communication and co-operation benefits everyone. Also, combining differing skillsets can offer new insights to your own tasks and provide new ways to complete them more efficiently. A developer that has experience in operating the system he is creating can acquire a deeper understanding of what is required of the system or what would be the best way to implement a feature. For a developer, being able to get a glance at what kind of approaches work best in practice can help improve future implementations. Similarly, operators can for example learn new tricks and gain a deeper understanding of the software they are using, if they get to work on the source code too.

It has also been stated, that DevOps can be seen as an extension to agile development [JBP16]. Agile has already conquered the world of software development and is utilized all around the programming world. DevOps is the next step of the evolution towards even more efficient software development processes. We never want to stay in what is familiar to us, but keep improving and trying to be able to find ever better ways to conduct the required tasks.

Another thing to consider, is what moving towards DevOps has required from the first adopters. Agile development practices and cloud computing have been seen as enablers for the emergence of DevOps [JBP16]. It is clear, that moving from a bounded and very formal development process towards unrivaled flexibility requires a lot of changes in the minds of industry professionals. Only after accepting agile ways of working it has become feasible for people to start moving towards even greater elasticity. On the other hand, cloud computing is seen as the technology that allows delivering software to customers the way that DevOps requires [JBP16]. Being able to continuously integrate new code for production would not likely be possible without the benefits provided, like on-demand scalability and availability.

The main goal of DevOps is reducing the time it takes for a software component to transform from an idea in a person's head, through the development cycle, and into a value producing part of the final product [ABD16]. This can clearly be applied to everything we are trying to accomplish while working on this project. While reaching the use of true DevOps principles will most likely elude us, we will certainly take steps towards this goal.

2.2 What Composes DevOps?

Some software development practices that are already used are considered vital for a successful adoption of DevOps. For example, without continuous integration it would not be possible to offer the

continuous and healthy product deliveries that are considered to be an important part of DevOps. If the software cannot be integrated and verified often enough, there is no chance to deliver it to the customer in required intervals. Delivering software faster and more often at the cost of quality is not a viable option.

There are many practical things that implementing DevOps contains and we are not going to list all of them in this paper. However, we consider offering some examples to be useful in better understanding the thought process behind DevOps. One example listed in [EGH16] is considering infrastructure to be handled as source code similarly to the product application. Used infrastructure is essential in proper functioning of the product and should be handled accordingly. This includes full testing, logging and monitoring services. The infrastructure should also be configured in a consistent manner. The goal of this practice is to make it possible for the viability of the product to be verified in all levels.

An interesting way of describing DevOps practices is that teams working at different stages of software delivery should adopt a using common Definition of Done [PuB16]. Often a task is considered done by developers for example when it has been verified and accepted to be merged into master branch. This is what DevOps wants to change. The different teams should collaborate closely in not only getting the feature pushed to the master branch, but all the way to the customer.

In our case, the reasoning behind all this can be stated as aiming for better ways of answering to customers' needs [RPA16]. Communication must be efficient on all levels and new requirements must be answered to swiftly.

2.3 Steps Towards DevOps

Changing processes and practices used for guiding the creation of value-offering software products is always a great challenge. This is especially true for large corporations that often rely heavily on these processes. Internal structures of an organization like this can be quite complex. Using well defined processes to deliver products helps understanding and controlling the product pipeline that can contain a very large amount of different steps. Sometimes the processes can be cumbersome, but in order to control enormous enterprise entities, some amount of them are needed.

In the telecommunications industry, we also face other challenges that can make the transition more difficult. One reason is that implementing DevOps is considered harder for embedded systems in comparison to cloud services [EGH16]. Taking continuous delivery in use while handling loads of legacy code is challenging. This is exactly what we are going to face. Being able to integrate the diverse code base that our products contain, into the delivery pipeline completely, is challenging.

Once again, it is also stated by Ebert et al. that one of the most essential parts of the transition is increasing collaboration of development and operations teams. This can prove especially demanding for us, as a telecommunications vendor. In most cases the division of tasks is quite clear. The distance

between different teams can be huge as they are from different organizations. Development is usually conducted by the vendor whereas deployment and operation falls under the responsibility of the service providers [JMN17].

Cross-organizational collaboration is almost always more demanding in comparison to combining your efforts amongst internal actors. Even simple aspects such as communication can be cumbersome. Different organizations can have conflicting guidelines and their goals can take different directions. Clear standard practices would most likely have to be defined, in order for the collaboration to be effective.

Another telco-specific challenge in using the DevOps practices is that the infrastructure is geographically much more divided than in case of traditional IT services [JMN17]. While we are moving towards cloud utilization in telco as well, the locational challenges are something that we should definitely be kept in mind when planning our transition.

Practical examples of conforming to DevOps practices are also covered in [EGH16]. One great advice given is that build, development and production environments should always be configured similarly. Getting to verify the whole software stack on all stages can yield great benefits. Interested parties are referred to the original text for more details.

The effect of DevOps transformation is not limited to molding the workflow of development, deployment and verification personnel. It is important that for example software architects and product owners also take the new way of delivering software into account [DJG16]. A clear indication of this, is that DevOps approach contains moving towards employment of smaller components in the used software architecture [EGH16]. Therefore, the transition offers challenges on higher levels as well. According to earlier research, designing software architecture that truly supports using DevOps practices is quite hard [DJG16]. Providing employees with related training well ahead of actually using the approach is essential.

2.4 Software development in a DevOps environment

While trying to take the DevOps mind set to the heart in general, we want to keep our focus in trying to improve the development practices. We want to stick to our chosen topic to keep the research quality as high as possible and offer answers to the questions that we have defined.

So how can we make our software development process more efficient by following the DevOps suggestions? Tools used by the developers must be made as simple as possible to operate, while offering all the necessary functionality [EGH16]. Developers need to be able to target their attention towards problems related to the program being developed. For example, building and installing the software under work should be made simple. Fast delivery cannot be achieved if even software development is not efficient.

According to [RaS16], in addition to sharing knowledge between different teams, tools and infrastructure should also be shared. This will help everyone at all levels to understand the work of others better which in turn can reduce communication difficulties. The software that is constantly running on the same infrastructure will not suffer from surprising problems caused by changing operating environment. Propagation of information and communication in general will be easier as people use the same tools and thus have access to exactly the same information.

We must aim to assist developers and operators alike by automating many of the steps of software delivery [RPA16]. When the manual effort needed, for example, to test or install software is minimized, employees can really focus on their actual tasks.

The tools used should also support automated mechanisms of collecting information [CYC14]. From the developer perspective that we are looking, this could mean implementing processes for installing modified software. Gathering information by quickly adding debug prints for example, is a common method of finding malfunctioning components. Spending a small amount of extra time on daily tasks translates into a lot of time wasted over time.

When talking about DevOps, continuous delivery often comes up as a definitive term [KVL16]. In order to achieve this the automation must range from development to deployment. As mentioned by Kärpänoja et al., packaging and testing are important steps here. In our research, we will also look into implementing effective packaging processes. Our aim is to make testing and delivery of new packages simple for the developers. As suggested in [RaG03], what we are trying to achieve is in a way a combination of our current agile development methods with component based development practices. This falls naturally into place with the other goals of our research.

3 Accelerating the development process

The topic of this section, is looking into the tools that we can use for simplifying the development process. Allowing developers to focus their efforts on programming their respective subsystems is what we want to achieve. In this chapter, we will focus on smoothening the workflow from acquiring the sources, to publishing the code changes.

We want the developers to be able to work with software packages similarly to many open source projects. Developing and updating singular packages eases the process of getting to test your changes and makes publishing process faster. When making code changes and creating a completely new image from scratch, a lot of time is lost. If we can utilize software packaging to easily develop, test and debug the programs, our work will become much more efficient.

When developing a platform for applications to operate on, we offer different services that the applications can utilize [Vau12]. This is why, it is important to make our development process simple. The application developers must have better chances at participating on the platform development as well. This allows developers from other layers of the software stack to better understand the platform.

As application developers' understanding of the platform layer deepens, they can better utilize the different services it provides. It will be much easier for them to figure out what services they actually need to use and how they should be used effectively.

Communication with platform development teams is important. So are the possibilities to reduce time to fix bugs for example [Vau12]. If a certain application needs platform patching to function properly, they can allocate their developers the help platform developers, in order to speed up the correction process.

The improved development tools and practices can lead to great mutual benefit. By adopting similar practices and deepening the collaboration of different software projects, we will be able to create better software in a lesser amount of time.

All that we have discussed in in this chapter, supports our ambitions of producing software systems the DevOps way. Developing and building software more efficiently can reduce the software delivery intervals greatly. There are of course other things that affect this. While releases to customers are conducted on certain intervals, no development process can change it. However, our new tools help us in preparing to answer to stricter delivery schedules, when it becomes necessary.

3.1 Software packaging

Packaging software includes compiling the source code based on specific configuration and packaging

the chosen result files. These files can then be easily installed to a system that wants to use the packaged software in question. Being able to skip compiling the code all together yields great benefits. It saves the time of an end-user and helps people with less technical expertise to be able to use the applications they want. New disk images can also be built much more efficiently. Only the changed software components have to be recompiled. The rest of the system already exists as packages and can be installed directly.

There are multitude of package managers in use all over the worldwide computer infrastructure. Some of the most well known ones are apt-get [vaus17] and yum [Fus17a]. These tools use the pre-compiled software packages in different kinds of repositories to allow users to have easy access for large amount of programs with minimal effort. Finding out and installing the required dependencies, before installing the desired component itself, are their main tasks.

It is stated in [Spi12] that the key gain in using package management systems is that the installation and maintenance of software components can be accomplished in a simpler way. The authors state that the reason for this, is that package management software organize and standardize the software production and consumption processes. These words present a clear reasoning for us to pursue packaging as a way of improving our software development process. Standardizing the installation process is needed for developers to be able to install packages of their choice easily. Organization of software components makes them more available and accessible, which is exactly what we are after.

As stated, we are currently delivering our software to customers as disk images. This practice is working well, and at this time is something that we do not seek to improve on. Allowing users to upgrade their software components in a package based way would not come without a price. We would have to allow access to package repositories from all sites where our products are used. Starting to use package based upgrade could also result in complex installation problems due to package conflicts. Even the most commonly used package managers often struggle with this kind of problems[ADT11] [TSJ07]. Even when finding the out the solutions, it is quite an expensive computing task to achieve [DiV11]. We want to provide our customers with simple installations and upgrades. We can avoid “dependency hell” situations that continuously upgrading singular packages can lead to. Thus, in this case we will continue providing pristine installation images for our users.

What we want to achieve, is to offer our developers easier way of installing software. This way we can achieve many of the benefits we have discussed in this thesis.

Easily installing new packages and adding debug information effectively is something that needs to be a quick process as it is a daily task for many developers. Finding out the best ways of working by using the best possible tools can offer great gains for all parties. Our developers can achieve more in less amount of time, and also keep their motivation to work higher. When there is no standardized way of installing different components, often a great deal of effort has to be placed on finding out how this is done. Solving these problems that are not directly related to the tasks at hand can be very frustrating.

We also want to have all software in our system to be fully traceable. When running our services, it should be easy to find out which components and what versions of them are being used. Starting to troubleshoot often starts with finding out what version of software was being used. Which release should be downloaded for examination? Is the problem caused by something that is already fixed on a newer version?

Obviously, software packaging does not come without its own challenges. Additional work is required from the developers, as they will be forced to write configuration files for packaging their software. Maintaining the software dependencies, that play an essential role in common software packaging approaches, can cause problems. Also, as our product does not currently support software packaging at all, this technology leap can prove especially challenging.

The tools that we use to achieve the simplicity of development, play an essential role in our success. We need to be able to support our developers and allow them to work to the best of their ability. Finding the right solutions for our specific working conditions must be the main goal.

3.2 Chosen tools

We have opted to attempt moving to packaging our subsystems as RPM (RPM Package Manager) packages. This decision has been made internally in a project I am involved in, based on previous research. The following sections will focus on finding out ways to help the developers interacting with the new software distribution approach.

We will start by looking at an existing RPM package maintenance tool fedpkg [GBS16a]. We want to have similar functionality available for our developers. RedHat places a good amount of work on the Fedora project and to our advantage, on its release engineering team. Their open source software development tools are sophisticated and have been shown to deliver results in large scale on the Fedora project.

The Fedora release engineering project has produced a large development, build and deployment infrastructure that is largely in production use. This will offer us many exciting possibilities if we wish to extend our tool base. While Fedora tool integration can cause issues, when not the whole tool set is not adopted, it can also yield great benefits when multiple different tools are needed.

Finding out what parts of the functionality of fedpkg we need and how are we able to implement them to operate in our environment. Some of the guides on package maintenance workflow where we started researching the subject are [Mat17, RoA16]. These documents offer a fine view of the tools used for building the most widely utilized Linux software packages. Key parts include the required interaction to fetch and build software. The commands described in these documents are abundant, and we do not see a need for such a wide variety of commands at this point. With the most essential components, we can already start to utilize the tool set in practice while continuing to fine tune the functionality.

We will implement only the essential features and keep adding new ones when there is need for them. User evaluation will surely be essential in determining our success in choosing the right features to offer.

We started by looking into directly making use of the wide utilities available in fedpkg and trying to configure it to fit in our development environment. First, we sought out existing attempts at achieving similar goals. Integrating fedpkg has been attempted by developers working on a Fedora-based operating system for Raspberry Pi, Pidora [Mah13]. This development seems to have stopped a long ago. However, examining their efforts indicated to us that using fedpkg out of the box requires a lot of additional components. Conforming to fedpkg needs instead of integrating it into the already existing systems seemed necessary. There has also been an RPM build system in use at CERN (European Organization for Nuclear Research) [Tra09]. This seems like a more valiant effort and has resemblance to what we are trying to achieve. These slides can offer a good overview of the basics of RPM build process. They have taken in use a full, Fedora-like development and build process. We cannot use similar approach as we must respect our existing processes. Still, this shows us that the Fedora approach can indeed be used to empower software development.

As our examinations quickly showed that full functionality of fedpkg is tightly integrated into Fedora development environment. We would have more trouble disconnecting the ties into Fedora, than implementing our own changes. The most viable option would have been to commit ourselves to other Fedora tools, such as Fedora package database [Chi14], like they have done at CERN. They have incorporated the full Fedora release engineering toolset in order to compose their software products.

For us, this approach is out of the question, as we need to combine software components from many different providers. Tying ourselves to singular ways of storing our software cannot be accomplished. Different components need different handling, as we will learn later on.

A Koji [Ata17] server has recently been set up for our research purposes. It is an RPM package and image build system used by Fedora. Koji offers a centralized place to monitor the ongoing builds and download the results to debug possible problems or take the components in use. Koji can be scaled by connecting more builder machines to the hub to answer needs of large software projects. We will be able to support multiple different architectures if necessary. Koji can be accessed through a web-based user interface and a command line client.

Koji is a well established tool that can be relied on and is also proved to be an efficient hub for building RPMs. We can scale it’s building capabilities by deploying additional builder nodes and offer support for different package releases with tagging functionality. In addition to this, it is already well integrated into other Fedora release engineering software that we will be adopting.

These tools have been designed to co-operate from the beginning. This enables us to achieve the best results in wide development use. Difficulties with communication among different tools will be minimal. The upstream will also provide support for the combination of these tools. This way we do not

have worry about our issues being declined due to unsupported tools being used as part of the chain.

In our work, we will utilize the command line clients building and configuring capabilities. The web UI will only be used for monitoring ongoing tasks and fetching results when needed. We will also be performing some maintenance tasks directly on the Koji server. Examples of such tasks are configuring Koji itself, debugging Koji malfunctions and in a few cases, even directly modifying the postgresql database used by Koji.

Koji was also included in the work conducted at CERN. We were in luck to be able to incorporate this functionality into our work without the effort of setting it up ourselves. We must however, start examining the possibilities it offers. Only a little examination has been conducted on the system before our involvement.

RPMs have two type of dependencies: runtime and build. Koji server enables resolving the package dependencies in a chroot build environment using mock [Suc17]. With this functionality, developers can easily build RPMs for local testing and debugging without having to install all the build dependencies for each package.

We can also use Koji in our chain of building and publishing the RPM packages. When preparing an official build of a package to be released, the build will be handled on Koji server and the resulting RPMs will be made available for download from the server.

Koji is a powerful tool and we will want to see how we could get the most out of it. Taking fedpkg into use sets too many boundaries on us. However, we want to be able to utilize the build functionality available to us. Creating tools, that integrate well with both, the Fedora software ecosystem, and that of ours.

Fedpkg is a python subclass of a tool called rpkg [GBS16b]. Rpkg offers most of the functionality, while fedpkg integrates it into the Fedora systems. This is a perfect opportunity for us to adapt these tools for our own needs.

We have decided to create our own rpkg subclass called rcppkg. This way we can offer the set of commands that we actually require to be able to optimize our developers' effectiveness. We want our tool to offer some functionality that fedpkg does not contain and likely would not accept to be merged. We can leave those parts of the development process that we are comfortable with intact. We can also avoid having to use all the systems that have been chosen by Fedora.

Those that examine the rpkg and fedpkg code base, can quickly see that functionality that fedpkg implements on top of rpkg is quite small and it is strictly related to specific Fedora needs. It would make no sense for us to fork fedpkg, then stripping most of its functionality to implement our own. Rcppkg will be examined more closely in the following chapters.

3.3 *Developing rcppkg*

We approached developing our tool in the order of the natural workflow of a developer. The initial implementation offers four simple commands for the user:

Command	Purpose
$ rcppkg clone	Clone the spec file repository into rcppkg build direcory. Currently this is configured to ~/rcppkgbuild. Then based on the information acquired from spec file repo, clone the sources to current directory.
$ rcppkg local	Locally build an RPM from sources in current directory and the files in matching spec file repository under the rcppkg build directory.
$ rcppkg mockbuild	Locally build an RPM in a mock chroot from sources in current directory and the files in matching spec file repository under the rcppkg build directory.
$ rcppkg build	Build an RPM in a mock chroot on the Koji server from sources fetched from remote source file storage or git based on the information acquired from latest spec file repository version.
$ rcppkg search	Search through the packages in git spec file group and print all matches.

These commands are enough for developers to be able to build and maintain RPMs effectively. Builds can be carried out both locally and remotely. The most essential command is ‘rcppkg build’ as this allows us to connect to Koji and build RPMs for relase. The possibility for local builds is also essential because we do not want to clutter the Koji server with test compilations. We have also wanted to incorporate the search command to quickly be able to locate the needed packages from the command line.

3.3.1 rcppkg clone

First, we looked into assisting developers in accessing the source code of different packages. For this purpose, we have implemented support for “clone” command. Using git for source code cloning requires knowing the location of the repository you want to access. In addition to this, much of the source code is not even stored in git. This leads to the previously mentioned problem of spending unnecessary time on locating the right sources.

We want all developers working on different levels of our software stack to be able to access all of the source code running in the systems developed. This can often be challenging due to source code being stored in multiple places. This way we are also conforming to the DevOps philosophy of standardizing

the tools in use. Operating system developers often deal with source tarballs copied from upstream, whereas application developers use mostly version control systems. Having a single tool that can be used by everyone is our first step towards DevOps in this paper.

By using rcppkg, the developers avoid having to find out anything related to where the source code is actually stored. Developer just has to know the name of the package and with a single command, "rcppkg clone <package name>", the tool will be able to fetch the latest packaged source code for the developer.

When building RPMs, specific configuration files called spec files [BMS00] are needed. When fetching the sources via "rcppkg clone" to the current directory, spec files are also fetched into a special "RCPPKG_spec_file"-directory. This way the developer will instantly be ready to build the package in question.

3.3.2 rcppkg local and rcppkg mockbuild

After making the needed changes, the next step for a developer would be to compile the source code and test it. We offer two different commands for enabling local testing. Commands "rcppkg local" and "rcppkg mockbuild" both compile the sources in the developer's working directory and create an RPM package. The difference is that "rcppkg mockbuild" uses the Koji server to set up a chroot environment and resolve the package dependencies by utilizing mock. When using "rcppkg local" the compilation and RPM package creation will be done locally and the developer will need to resolve possible missing build dependencies manually. Both have their advantages, but we suspect that mockbuild will be the more popular one, as it allows developers to keep their own machines clean. Local builds on the other hand take less time, once the build dependencies have been resolved.

3.3.3 rcppkg build

Lastly, the command "rcppkg build" works in a similar fashion as "rcppkg mockbuild". The chroot environment created by mock will be utilized and the actions required from the developer are kept to a minimum. Here however, the results will be stored to the Koji server and be made for everyone. Developers should always first create and test local builds before using "rcppkg build" to publish their results as official builds to the Koji server.

3.3.4 rcppkg search

In addition to these practical, development related commands, we implemented the command "rcppkg search". We feel that this kind of utility will be highly used and serves to reduce "development overhead", the time spent on other activities than the actual task. Being able to use keywords to search for available packages without leaving your terminal window is effective. The time spent of switching tabs and finding correct URLs can start building up over time.

A major difference to the fedpkg way of doing things is that our implementation handles the source and spec files simultaneously. Fedpkg has separate commands for fetching the sources and spec files

while we wanted to provide simplicity by cloning everything that is needed to be present for issuing a build. Probably the biggest functional change that we have made is that we offer integration for building from git sources. In Fedora builds, the sources are usually packages as release tarballs. Many included packages are constantly being developed and we do not want to force the developers to issue source releases each time they want to create a new build.

3.4 Setting up rcppkg

We have attempted to minimize the manual work required to start packaging the software component you are working on. To use rcppkg, you will only have to create spec files for the packages you want to manage and setup git repositories where these are stored. Spec files are configuration files guiding rpmbuild in compiling the software and creating an RPM package out of the matching source files. In Appendix 1, we have attached an example base for a spec file for one of internal packages that need to be built from git sources. The most important parts of the spec files are listing the dependencies of the package, compiling instructions and listing files to store.

There are two kinds of dependencies to be listed for a package: build dependencies and runtime dependencies. We would argue that listing finding out the dependencies of a package is the most time-consuming part of creating a spec file. Build time dependencies are easily found out as you proceed to test the correctness of your spec file. However, the runtime dependencies can be quite elusive if they have not been listed correctly in some documentation beforehand. This is an example where a good test coverage can come in handy. Running most of the functionality of a package should unveil external packages needed for it to operate correctly. Otherwise, some feature of the software component may be unusable due to missing a needed library for example.

Overall, the spec files are quite simple to create. Often programs already follow configure – make – make install [Wil03] installation process. This makes it easy to compile them and therefore also easy to specify their installation procedure into the spec file. Sometimes there are additional steps like copying additional configuration files to target locations. More of often than not this is also quite simple to handle.

Listing the result files is also a fast task especially if you have previous experience with the package in question. By convention, the files are built into a specific build directory where they will be collected from as specified. You can even store all the result files in the RPM just see the full list. You can then choose the ones you actually need to have included and then modify the spec file to match the decisions.

The spec files should be stored under a same git group in their respective repositories. When following our guidelines, the group is already configured into rcppkg. This way, component developers do no need to touch the rcppkg configuration. Only if one wants to take rcppkg in use on a different project,

and replace our spec files, will rcppkg require additional configuration.

To take rcppkg use locally, you only need to:

1. install the dependencies
2. install rcppkg
3. set up certificates for Koji authentication

You are then set to use any of the commands that we have provided.

We have now provided a way to easily and efficiently develop, build and install RPM packages. Software developers will be able to enjoy the benefits of software packaging, without having to focus on the complexities involved.

3.5 Implementation challenges

As we have phrased it before, adding a new package to be maintained with rcppkg should only include minimal effort. We want it to be limited to creating a spec file for the package. This step cannot be avoided, but can be assisted by offering guidelines. However, even this might not as simple as it sounds. There are hundreds of packages that need to be added if we want to be able to manage the complete software products that we are working on. Manually creating spec files for all of these packages would obviously take a lot of time and effort.

One possible approach we have looked into, is reusing the spec files from Fedora. This way we should have the dependencies and files to be included already defined. Some tweaks might still be needed in order to make the spec files compatible with rcppkg. We would possibly also need to do some other changes, such as removing Fedora-specific patches that we do not want to use. Obviously, this approach is viable only for the open source packages that we use.

When it comes to our own subsystems, we need to create spec files from scratch as currently these do not exists at all. We have already created example spec files for our internal packages in order to start testing developer workflow with rcppkg properly. We must be able to test the whole scale of functionality we have promised. Both open source and internal packages should be supported.

When starting to use rcppkg in a larger scale, we should definitely look into possible ways to automate creating minimal spec files for the packages. A lot of resources could be saved this way. Then again, when dividing the work between all the subsystem developers, the task seems a lot less demanding.

We also faced some challenges due to most of our internal source code not being public. Because of this, we need to be able to set up ssh keys [YCB99] for git access. It sounds simple enough, but due to many different tools being used, the correct configuration becomes much harder.

As Koji uses mock to build the packages in a chroot environment, we need this chroot environment to

contain the keys. The challenge here is that Koji is creating configuration used by mock automatically and we are yet to find out the proper way to change this. One approach would be to directly change Koji source code. Offering better chances to modify the created build roots would become handy in general. For now, we have settled with adding ssh keys to be included alongside the spec files to get on with the testing.

Dependencies on our internal packages prove to offer additional challenges as well. When depending on open source packages, the dependencies are easily installed in the chroot environment using dnf [Fus17b]. When it comes to internal packages, currently we have to manually add the packages to a dnf repository after building them. This is the only way for the dependency resolver to find them. This is another caveat and we should find a way to overcome it by having our build system automatically include them in the used repositories.

We considered four different solutions to solve the problem of retrieving source code with restricted access:

- Configure mock to bind mount .ssh directory from root home to chroot home
- Include source code in the spec file repositories
- Change Koji to read entire confguration
- Modify mock source code to copy ssh configuration from root home to chroot home

The first option seemed the most natural and that is what we started investigating first. However, this turned out harder than expected to achieve due to different layers in play. After further studies, we found out that the same problem had been encountered on other projects [Hea12] as well. Through this, we found out that chroot initialization will clear the .ssh directory, rendering this approach invalid for us to pursue further.

Including spec files with the source code was something that we had tried to avoid from the start. Our aim was to be able to keep the source code completely unaffected when taking these new tools into use. This approach would also require us to make further changes to rcppkg that would render its code much more complicated. For example, we already have different behavior for cloning open source packages and internal packages. If we were to include the spec files in the source repositories for our internal packages, we would also have to change the code to find the correct way to find the spec file for each package. We consider this to be close to a last lifeline to be used and decided to find another way to complete this task.

Modifying Koji was definitely an option that we considered strongly. Having even wider possibilities to configure our chroot environment in the could be useful. However, testing our system showed that we had no need to any other configuration changes. Having to provide configuration files was an extra step that we did not want to take needlessly.

Finally, we decided to modify source code of mock. We now have a version of mock that copies the

ssh private key and the known_hosts file from root home to chroot home [Kor17]. This way we can avoid modifying our git repository structure, while still keeping our continuously required efforts minimal.

We first were afraid that this approach would hurt the setup simplicity of our tool. If users would have to install this modified version of mock on each machine they want to use rcppkg on, the required setup effort would increase a lot. This would surely make many users hesitant about the benefits of our tools. Luckily, this is not the case.

The reason that modified mock only needs to be installed on the Koji server is that there is no need for builds happening on the user's machine to clone restricted repositories. When attempting a local mockbuild, the source code will be copied from the user's working directory. Only when executing a release build, will the source code be fetched from a remote repository. Thus, a standard version of mock on user's own machine is enough.

The biggest caveat in this approach is that we cannot support different user accounts. If rcppkg is taken into wider use, we would most likely be willing to monitor who has executed the builds. This would require us to again make source code changes to mock.

4 Containerization

Linux containerization is an important modern virtualization technology. This makes the subject highly relevant for our work as well. When taking new tools into use, we already want to be prepared for possible changes that face us in the future.

Using containers can roughly be divided to two different categories: packaging complete operating systems and packaging single applications [Ris17]. Different technologies are designed for these use cases. Operating system containers such as Linux containers [Can17] are used for the former and application containers such as Docker for the latter.

We will follow the existing research and focus on Docker containers. It is a project started in 2013 [Mer14] that aims at packaging applications by utilizing Linux containers. The aim of the project is to avoid the "dependency hell" situations between packages by placing the required dependencies alongside the application in the container. This container can then be used to run the application on any platform that supports Docker containers [Mer14].

Containers are a much more lightweight approach to virtualization than virtual machines [VMw06]. It has also been shown that computational performance on them exceeds that of virtual machines [Joy15]. Containers run directly on the host operating system. Virtual machines on the other hand are used to execute the entire software stack, starting from the operating system. This causes their overhead to be small and allows them to shut down and boot up quickly [Mer14]. On the downside, some security issues might arise as they are not as strongly isolated from the host as actual virtual machines are.

Containers have been utilized to great lengths to provide some of the most stable services we have ever seen. Their use seems to be widening all the time. New users are adopting the usage of containers and new use cases are being invented. For example, there even is an operating system that runs Docker as the initialization service (pid 1) [Ran17]. All applications run on top of it are applications started by Docker.

4.1 Microservices

Microservices is a prominent architectural approach that has been utilized by many renown enterprises such as Google [KBB16] and Netflix [SZY16]. This approach has been developed to at some point entirely replace previous architectures such as client-server model. The qualities required from high-end IT services are constantly changing. As the user base keeps growing, answering to increasing demand of attributes such as scalability, reliability and efficient use of resources [SZY16] is what microservices architecture attempts to offer.

There is one main reason to the strength of this architecture. It is the inherent division of the system.

Software can be seen to gain in robustness as there is no single point of failure and scaling is more efficient as it can be conducted only on the services that demand it [SHD16]. Having separate, simple and lightweight services to deploy keeps the operating environment very flexible. This is needed as the cases when operational context of the application stays exactly the same, are rare [OEC16]. The new requirements set by customers must be met rapidly. However, developing of these separate services is more complicated [LZX16].

The challenges in design and implementation of these services is what we must battle against in our research. If the creating the applications that compose our system is going to become even more challenging, we should make sure that the development processes are made easier in return. Adopting lighter processes that restrict developer work is essential. We should also be able to automate as much of the software delivery process as possible. Luckily flexibility of containers helps us in achieving this as well.

As the name implies, separate microservices are small, independent services [New15]. The variety of microservices in a system can be large. They are not tied to using the same techniques. It has been shown in [New15] that microservices operating as a part of the same application can use different programming languages and can also answer to different performance requirements.

The approach can also be seen to enable adopting DevOps [PZA17]. As stated in [SZY16], microservices help in being able to update running software frequently without disruptions as the updates can be completed in parts. Currently rolling updates can be offered by booting up instances side by side, but this approach allows this to be achieved within a single running system. It could also be stated that containers enable microservices approach. According to [KBB16], in this approach, applications are composed from simple parts that are each deployed within a separate container. Linking these separate parts together then offers the full functionality of the application.

Applying microservices architecture to existing software is no simple task. Monolithic legacy applications that we have in abundance in the telecommunications industry are often far from this single responsibility principle embracing architecture. Two major challenges are identified in [ECA16]:

- How to handle dividing the system into smaller parts?
- How to define the correct dependencies while retaining the previous functionality?

In [SZY16] the progress of dominant software architectures is examined. They start from client-server architecture evolving all the way through to microservices. Their goal is to identify the biggest caveats that implementing microservices causes.

- Coordination between different development teams is difficult as microservices should be developed independently [SHD16]
- Need to ensure that no coupled microservices are created
- Amount of communication through networks is required

- May cause requirement to increase network security
- Data storages need to be split out for different services to access them
- Monitoring will be more expensive as processes are highly distributed
- Figuring out the optimal size and number of services is very challenging

They have also acknowledged the requirement to use lightweight communication methods as the amount of traffic is high. As we previously mentioned, the security of containers can be seen as somewhat of an issue. However, as stated in [Fet16] the underlying systems will never be completely secure. That is why an application should, running inside a container or not, take care of its own security.

There are modern ways to achieve this such as allowing applications to keep their state stored in encrypted memories [Fet16]. This way not even privileged users will be able to access its runtime without proper means. It is noted in [ECK16] that in addition of not relying on the underlying system, applications must also question the integrity of their peer services. Any other part of the system should be seen as possibly malicious.

4.2 Containers in rcppkg

In addition to previously mentioned utilities of rcppkg we are also looking into implementing "container-build"-command for it. The adoption of containers is increasing rapidly and we do not want our tools to be left obsolete once the transition truly starts. The basic functionality is already present in the underlying python class, rpkg. When developing application to be ran inside containers, you often want to follow single responsibility principles [Mer14]. This is possible as containers are computationally so cheap to use.

There is two venues that the container-build could be used for:

1. Packaging software into containers instead of RPMs
2. Installing software into containers instead of virtual machines that we currently use

There seems to be strong opinions in favor of using Docker container packaging when delivering applications [Moc15]. With true microservices, the used components could solely be minimal containers that are based on some minimal base image such as Alpine [Alp17]. Before this will be realized in practice, the containers can be used as a replacement for VMs when possible. With this division there would be two steps:

1. Move software to run on containers
2. Break the software in the container to multiple containers

After this, the second step would be repeated until there is only a single service present left running in

each container.

When setting up the container-related Fedora release engineering tools, it will be important for us to make sure that both of these cases are supported. Step one will be simple as often developers can just define a package manager to install software into the container that they want to build. In step two, when we want to strip down containers from everything that is not needed, additional challenges may arise.

Unfortunately setting up full support for building Docker images in rcppkg requires more than just some minor code tuning in our python subclass. We need to set up a new plugin [pba15] into our Koji deployment. This seems like a relatively easy task. According to the documentation, with some minor configuration changes we should be able to integrate this plugin with our current Koji build system without disturbing the current functionality.

If previously mentioned steps were enough, container building would be in use shortly, however there is additional component that will have to be taken into use. Koji plugin koji-containerbuild uses a build system called OpenShift [Red17a] to create the container images. There is an open source version [ope17b] available, that we can start taking into use without higher level management decisions. RedHat offers a supported version of this software that is one possibility to look at if it proves useful [Ris17]. Getting to know this tool and actually deploing it successfully is what constitutes the biggest challenge for us before being able to integrate Docker container building into rcppkg.

Similarly to building RPMs, developers will be able to commit their changes to version control and then build a new release version of the package with a single command. Dockerfiles are simple to create and modify so teaching the package developers to make changes in them instead of spec files should not be too much of an issue. Still, the adoption of containers must start from the architecture side, only then could this functionality of our developer tool truly be useful.

4.3 Docker build configuration

Previously, when building RPM packages, we used spec files to guide the build process. Now that we are moving onto implementing container build, we must use Dockerfiles [Doc17a] to specify how the container should be built.

Dockerfiles consist of instructions and arguments. Instructions are pre-defined commands to notify the build process of what its arguments should be used for. Instructions are places in the start of the lines and they are usually written in upper case letters to be able to easily separate them from the arguments. Here are some of the most commonly used instructions that we have come across while getting to know Dockerfiles:

- FROM

- ENV
- RUN
- LABEL
- USER
- WORKDIR
- ADD

FROM- instruction is used to define the image to be used as a basis for your container. The rest of the build process depends a lot on your choice of container platform. This instruction must be present in every Dockerfile. ENV can be used to set environment variables into the container and these can then be used in the Dockerfile to assist in defining the build process as well. With the RUN-instruction you can specify shell commands to be run inside the container. This is an essential part of the build process as we will often be using a package manager to install the needed packages with the help of RUN. LABEL can be used to specify additional metadata to be added into the container image. Form of key-value pairs should be used here. WORKDIR specifies the path to the working directory to be used for running your shell commands and USER determines the user under which the commands should be executed. Finally, ADD-instruction can be used to add local or remote files to be included into the container image.

4.4 Implementing container-build for rcppkg

Again, we were able to adapt the command implementation from rpkg superclass with minimal effort. Due to our endeavors to keep the spec files separate from the source code, we had to adjust finding the correct git version to build. When building on the Koji sever, this is needed to format the checkout URL properly for the server to use in fetching the Dockerfiles.

As mentioned before, our spec files will not be used at all in the process of building Docker images. Dockerfiles are used to configure what should be installed into the container. To explore this build system as a proof of concept, we have defined a minimal Dockerfile with the following instructions: FROM, RUN, LABEL, WORKDIR. We will be using Fedora as a base image and install one internal RPM package with dnf as an experiment to test the functionality of container-build in rcppkg. LABEL-instruction is required to define some information for the OpenShift build system. Without this information, the resulting Docker images cannot be published on Koji.

We have set up a local Docker container to run the OpenShift server, as setting up a proper OpenShift server would likely take a lot more time. Evaluating the functionality of a full OpenShit deployment is also out of the scope of this paper. Thus, we will choose a simple approach to achieve the needed functionality with minimal effort.

In between OpenShift and rcppkg resides Koji, which we have now configured to support building containers. Achieving this was relatively easy, installing the koji-containerbuild [pba15] plugin was achieved simply with python-setuptools [Pyt2017]. After the installation we just had to modify two Koji configuration files to inform it to look for this plugin and enable the it to locate the plugin correctly.

After getting the connection working between our host system, in this case the Koji server, and the OpenShift build system running inside a Docker container, we ran into some additional trouble. OpenShift build system utilizes a container management tool called Kubernetes [Lin17b]. This is a tool that we have seen used in some demos, but no have no hands-on experience with at all.

Configuring Kubernetes to function properly proved quite challenging. Luckily our other thesis workers who are conducting work that revolves around containers have more experience on the subject. With our combined efforts we were finally able to get closer to getting the containerized OpenShift deployment to function.

At the start of the build process, a separate container is deployed to handle the build in question. This container will replace the mock buildroots that were used with RPM builds. To offer the qualities required from buildroot container, it has to include atomic-reactor [pro17]. Atomic-reactor is a python library that assists in building the container image. That is the most important requirement and the buildroot image can be customized if needed.

Creating a Docker image that contained atomic-reactor and fulfilled our other requirements proved quite difficult. There are atomic-reactor images available on the internet to be pulled and used directly, but we could not take this approach. We were forced to implement some code changes into the atomic-reactor library, so we had to build the image ourselves. Often customizing the image can be easily achieved, but as we were forced to again modify another open source component, the task seemed needlessly complicated.

The need for atomic-reactor changes was caused by trouble with submitting result image into Koji. As the build name was always already being used by the process calling the Openshift client, build process was not able to reach completion. Koji will deny processes that are trying to publish results under already existing names. We implemented a small workaround to fix this for now and published the code in case it is needed again [nip17].

Next, we had to make changes to the basic Dockerfile [Doc17b] used for building the atomic-reactor image. We had to copy the Koji certificates and configuration into the image to be able to communicate with Koji. We also had to change /etc/hosts [hos17] file inside the image to allow the ip of our Koji server to be determined properly inside the buildroot container. That was harder than expected as modifying this file isn't allowed by Docker. We had to use a workaround [Wor15] to be able to proceed with our implementation.

After this, there were only a few final modifications we had to make to get this full process working.

We set the line ‘Environment="NSS_STRICT_NOFORK=DISABLED"’ into the systemd [sys16a] service file [sys16b] of Koji daemon, to make it trust our OpenShift server. Then we just had to add atomic-reactor as a content creator into the Koji database and allow the user that we conduct the builds with to use this build method. Now, we were able to execute complete Docker builds and download our created test image from Koji.

As we have seen, the task of taking container build functionality into use proved quite difficult. This was caused by the fact that there are so many different components used in this build process. It also seemed quite hard to find information regarding these components. They all are still actively being developed, so we expect to find out easier ways to achieve what we want in the future when they are used by an even wider audience.

It should also be noted that the system that we have set up should be used strictly for test purposes. Many points of authentication have been disabled for convenience so it is not secure enough to use in production.

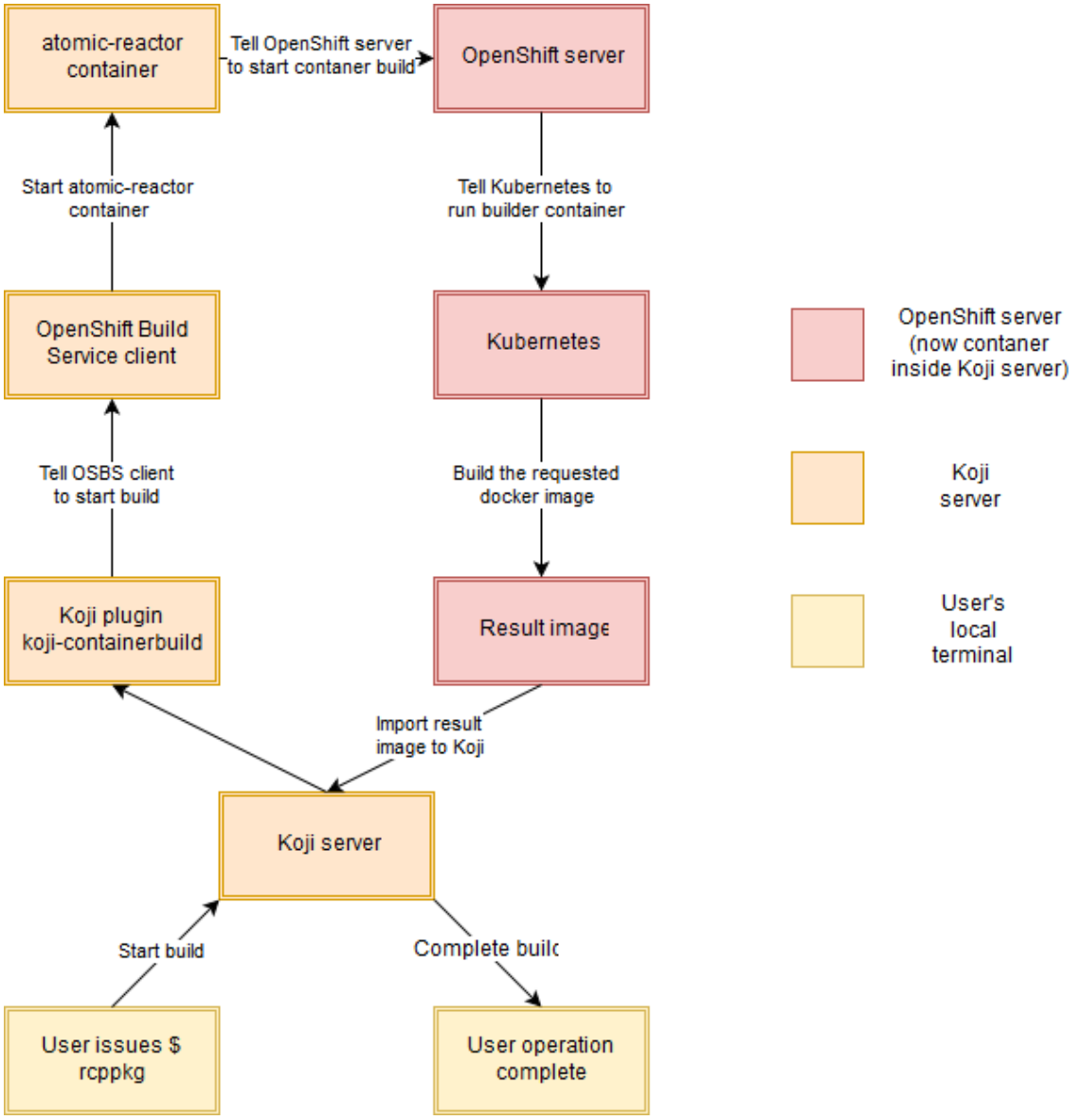

Figure 1: This figure shows the flow of container image build that is initiated via rcppkg and carried out by Koji and OpenShift. The squares represent different states that the build will move through before reaching completion. The arrows represent what action is carried out by the system when moving to the next state.

There are many different open source tools used here and the build process is quite complex. As an overview, Figure 1 illustrates the execution flow of "rcppkg container-build" from start to finish. The colors indicate which part of the build process the components are used in.

4.5 Using container-build

While setting up support for container-build was demanding, the usage itself is as easy as the other rcppkg commands. To clarify, in order to use container-build command, no additional setup is needed

to be completed by the developer using rcppkg. It is enough that the Koji server maintainers make the needed changes to be able to fulfill the container build request.

If rcppkg is already installed on the system, similarly to RPM builds, just one command is enough: “rcppkg container-build”

As with previous commands, this should be issued inside a source directory that is supported by rcppkg. In addition, there should be a Dockerfile present in the rcppkg spec file repository of the component in question. However, in production use we would likely to implement own git group for storing the Dockerfiles and possible other files needed for successful builds.

After finishing the build, user can head to the Koji dashboard on a web browser and check whether his/her build has been successful. In case of success, there is a download link available, where user can get the newly built Docker image. After download this tarball can be loaded into Docker by issuing “$ docker load <tarball name>”.

We have also included a Docker registry to be ran in the Koji server within a container. This registry will offer no persistent storage but will be enough for our testing purposes. All images built will be stored in the Docker registry too. From here, it is even simpler to load them for your local usage. Simple “$ docker pull <koji server URL>:5000/<package name>” will be enough to acquire the image and start using it. Uploading the images automatically to a docker registry would definitely be the image delivery approach that we would use in real use cases.

4.6 Challenges

We have now successfully set up container build functionality for our Koji server. While this deployment is far from being viable for production use, it adequately fulfills our needs for evaluating this build approach. We can start producing container images in larger amounts for the packages we have chosen to test building RPMs with.

In addition to the biggest reason, lack of authentication, our build system has many other weaknesses that would prevent direct cloning of this environment to real use. Some of these may not be issues in practice but they all should still be kept in mind.

Firstly, there are some limitations which images we will be able to build. Currently building an image that does not contain ‘/bin/rpm’-executable will always fail. We have not examined this closely as it is not a issue for our tests, but in production use, the feature could prove annoying. The build system is derives from building RPMs so there must still be Koji processes in place, that use RPM functionality for various tasks.

We will also be unable to use scratch images with this build system. Scratch images are completely

blank Docker images that can be built upon. These can be very useful when attempting to build minimal container images that do not contain anything unnecessary. I can see this very well being the case for us, as we are always trying to ensure that our systems are as compact as possible.

The previously mentioned caveat arises from the fact that the base image is pulled separately before the build in our atomic-reactor buildroot. Scratch images cannot be pulled, they can be only built on. This is why their usage is not possible without modifications.

The problem with initial Koji build failing is also still present. When a user starts a container image build, there will be a Koji build for carrying out the image creation, followed by a separate build for importing and tagging the created image to the Koji server. What we have noticed is that the former of these will always fail. This is due to Koji expecting a build having RPMs in its resulting files. No practical downsides should follow from this but it certainly is not a good practice to fail builds without real reason in production build systems. Many false alarms and needless debugging sessions could very well follow.

Even though these caveats are not too big and must not weight too much in our evaluation, we see these as clear signs of the immaturity of utilizing Koji to build containers. Taking the basic RPM build functionality in use was much more straight forward. This is of course also affected by the fact that there was lesser amount of different systems used.

After testing our container-build utilizing Koji and getting to know OpenShift platform, we are ready to discuss its viability. While the use cases of Koji should still be evaluated from different perspectives, currently the need for this tool in our project seems minimal.

As we are moving towards containerization, we are also attempting to embrace DevOps principles as much as we can. A big part of this is moving towards automated deployments. We are aiming to automate the build process as much as possible and minimize the actions needed by users. In this scenario, there should be no need for a manually issuing official builds. In that sense, using manual build commands does not fit in the picture.

Users will in most cases want to test their code before pushing it, but this needs to be looked into separately in any case. Our command container-build has been designed for releasing builds and not for testing.

While this command is useful, the build process seems overly complex. We strongly think that when if true automated deployment practices would be embraced, there would no longer be need to utilize Koji. OpenShift can clearly be useful, but in this case having Koji server as part of the build process does not offer enough benefits anymore.

It seems to us that using Koji for this purpose this goes against what we are trying to achieve, simplicity. Already setting up this build system was exceedingly complex. No doubt it would take no time from an expert or even from us from now on, as we have now done this before. Still, if we take a look at Figure 1, we can clearly see how many different components we would need to use. Simply the

maintaining the number of components would encumber more effort on us. It is clear that they will have to offer clear value for the maintenance to be worth it.

The biggest reason to use Koji for building containers, would be continuity. Having developers use a new command in a software they have experience on is often more tempting than learning to use new software tools.

We are not certain this would be proper basis for choosing our operating environment. Decisions should be made on based on best overall available tools, not the easiest adopted one. Usability and ease of learning is essential but as we question the need for a manual step that Koji would help us simplify, even this loses value.

When this issue becomes timely, we should look into how can the users best test their code locally, or is this kind of functionality even needed anymore. As the used technologies are becoming more advanced, our supporting processes should also reflect them.

In the end, automated deployments are far in the future for the telco industry. Especially when we are slowly starting to implement software that is run on containers we can likely utilize these features of Koji and rcppkg. Having a few components here and there that are deployed as containers certainly does not offer incentive to move to another build and deployment system.

With closer examination, more implementation by the upstream developers and proper setup, container-build could very well offer useful utility. We must wait to see how the containers will be taken into use and attempt to offer our support in this regard.

5 Storing open source code

Use of external packages has been on the rise for a long time, as cost and time of development process can be cut greatly [Cla92]. We have already existing version control repositories in use for our own packages. Packaging out internal subsystems was tackled quite easily as it is clear how the source code is and will be stored in the future. However, there is a lot of open source software that we use as part of our products.

Most of the open source packages will compiled directly from the upstream source code, with possibly a few internal patches applied to them. Thus, there is no sense creating new git repositories for each of them and having to merge the code every time there is a new release. This would result in a lot of manual labor as there are hundreds of these packages in use.

At the start of this paper, we mentioned that the operating system that we will be using on the next version of our platform, is yet to be decided. This seems like a great opportunity to look into the foundations of the operating system's build process. In this chapter, we are going to briefly examine how the source code is stored right now, and how could we improve on the storage method. In case of transitioning into a new operating system, there will surely be package and version changes that we face.

5.1 NetLinux solution

At the moment, NetLinux stores all sources of its upstream packages in a single directory on an internal server. In other words, all the versions of all the packages are stored in a single remote directory. Build process fetches the packages from the repository and uses md5 hash [Riv92] to validate the downloaded package.

This storage approach has multiple weaknesses, most importantly lack of structure and long term usage.

Storing all of the packages in a single directory is bad idea from the point of view of a human actor. While building the system, a compilation script will have no trouble picking a specified source tarball from the directory. However, when a person wants to examine the stored packages or find a specific one to download manually, problems can arise. Scrolling through different versions of different packages or finding the exact spelling of the filename to be searched for can take time. A clear structure is always useful when storing software components. Currently the number of packages is small in comparison, so the problems are not big. However, this could very well change in the future.

Maintaining the package base can also get quite complicated in the long run. There is no automatic functionality in place to assist in getting new or deleting old packages. This is why, in the long term, the amount of storage resources would most likely keep increasing over time. With no automated

maintenance processes in place, designated personnel would have to carry out screenings of the packages from time to time. Also, the lack of structure would become even bigger problem as the number of packages to scramble through would increase.

One might also wonder why do we want to store the open source packages ourselves. Why do we not just create the spec files and fetch the source packages from the Internet? Even outsourcing software storage has been widely used approach [Man14]. Firstly, we must be able to have control over the source code. For example, we cannot risk any public source code repositories letting in possible attackers that could cause security holes in our systems. Secondly, we must be able to control the availability of our source code. We cannot trust in a third party keeping their servers operational as much as we want. Another possible issue to consider is possible slowness or breakage when connecting to the Internet. We want to be able to guarantee ourselves that the source code will be accessible at all times or as close to that as possible.

5.2 Existing solutions

A huge Debian source code collection has been gathered in [CaZ17] to be available for download and analysis [Zac15]. This collection is far larger than the one we are dealing with, consisting of over 30,000 packages [Zac15] but we might well be able to draw guidelines from their ways of maintaining the source base.

Packages released with current and past Debian releases have been placed in debsources at the time of creation. In addition to this, debsource is able to get live updates of package updates [CaZ14]. This model, kind of a developer – intermediary – user interaction is very common nowadays, according to the authors. In this case Debian works as an intermediary packaging open source software for its users to simplify installation. Users can simply run apt-get commands to install compiled packages from Debian repositories. Similarly, we have an intermediary, currently a single directory, as a place of storage for open source packages. Users do not have to figure out the correct website the upstream releases are posted to, but instead can trust on Debian forwarding the correct sources when needed.

This dataset has been created to enable research on a large set of open source software. Research on macro-level software evolution is scarce, not because of lack of interest, but because of various big challenges involved [CaZ14].

There is multitude of functionality available in debsources. Packages can be sorted by names or by groups, code search can be performed in browser and there even is an automated garbage collection functionality for unused packages [CaZ14].

For Fedora, there is a source package repository collection available under http://pkgs.fedoraproject.org/repo/pkgs/. Arch Linux has listed its packages to be easily downloadable from https://archlinuxarm.org/packages.

5.3 Questions regarding our storage approach

As stated, we do not want to set up git repositories for most of the open source components. Currently, almost all of the open source components used in our project are stored as tarballs with git repositories set up only for only a few of them. These are components that we are making many own modifications for. Therefore, it would be quite messy to handle these modifications by applying large amounts of patches every time.

In our operating systems, we have far fewer packages in use than most flavors of these widely used open source distributions. This, is why we feel like there is no need for sorting mechanisms for accessing the sources. Most likely sorting the packages by their name will be enough in most cases to allow users to easily locate what they are looking for. From the ones covered in the previous section, this can be considered to resemble the Arch Linux approach most closely.

The ideas from more complex source package management systems we would like to utilize are automatic updates and automatic garbage collection.

We do not want to implement a completely automated approach into pulling new sources. This could lead to possible security vulnerabilities and a maintaining source code that we will not be using for a long time. What we want to examine is a possible notification system for new updates on used packages. How much would this decrease the work load of our OS developers? Could a system like this be relied on? Is this even needed or is manual learning of available updates enough in our case?

Garbage collection for unused packages is something that we would like to implement to avoid bloating the package base of our source storage in the long term. The size of stored components keeps increasing rapidly [VoD13]. We do not want to have full source history stored. Nor do we have a real need to archive all the past sources. We only want to store the ones that are still being used. This is to ensure we can access them for development and debugging.

The interesting question here is how do we determine which versions of packages are still needed? Under what conditions should a tarball be deleted to avoid cluttering the storage space?

5.4 New storage approach

What we want to achieve, is help developers themselves in accessing the source code of all components in use. In addition to allowing build systems to access sources, software repositories are maintained exactly for this reason. There has to be a place where different human actors can easily access the software code [SiS14]. Accessing is divided into two categories in [SiS14]: browsing and retrieving. Retrieving is made quite easy with our different tools assisting. Therefore, browsing is what we

mainly want to improve on here.

Changing our storage approach to contain one directory for each package, under which all of its versions reside, has other benefits besides simplifying package browsing. We could easily include in-house documentation of the package with the sources, without cluttering the package directory with all different packages' files.

By taking a look at Fedora source package storage, we can quickly see that many additional files are often stored. These include test packages, some installation scripts and documentation. It is clear that we cannot utilize our source storage like this, while there is no directory structure present at all.

In [Man14], software maintenance projects are considered to be divided into three phases:

- Transition
- Steady-state
- Preventative

The first phase contains moving the application from previous vendor to the new maintainer. This would often contain information exchange and guidance between the parties. This is not the case for us as components have also previously been in our use and are familiar. What we need to figure out is, do we want to automate this procedure. For now, we will not pursue fully automating package source retrieval and updating. This could save little time, but the used approach should be carefully reviewed to make sure that the acquired source code is correct and trustworthy. However, utility tools to make fetching new sources simpler would be beneficial.

In steady-state phase, software maintainer will be resolving different problem situations related to the software. Often this contains answering some tickets on different topics like access to the software or possible bug reports [Man14].

Moving to the final phase happens by maintainer acquiring extensive knowledge about the software package by performing related duties in the previous state. This phase resembles steady-state a lot, but can be considered to dive a lot deeper into the software with lesser effort. This is made possible by the expertise the maintainers have acquired from continuous experience dealing with the packages in question.

These phases are not that clear in our case, but are useful to keep in mind. External software maintainers of software [Man14] can have for example a lot more demanding transition phase. They also often have very strict contracts about ticket answer times and assigned responsibilities on fixing possible bugs in the software.

We could construct a documentation approach that would move us a bit closer to the external software maintenance lifecycle. In each open source package's directory, we could make small notes specific to us about the package. One of the most important things to note would be an internal contact that has better-than average knowledge about the package.

Currently, when encountering problems with open source packages, our developers are often puzzled about the next steps to take. Getting answers from the open source community can take time and they will not have knowledge about the special circumstances that we operate on. Having a few internal specialists assigned to each package would help our developers taking the first steps after encountering problem situations. When this information would be easily accessible side-to-side with the source code, moving forward with troubleshooting could happen a lot faster.

Other useful things that this documentation could be used for:

- System-specific installation guides
- Links for upstream bug reporting
- Information concerning Nokia-specific changes, for example internal installation paths, our systems' use cases

When it comes to automatically removing the old sources that are not used anymore, this is something that we would like to implement. Having our storage usage grow indefinitely is not desirable, as having too many files to complicate users' view is not. We will leave this as a future project to be kept in mind for now.

We have also started storing spec files in a larger scale to an internal VCS. The goal has been to re-use Fedora spec files. This has been possible for many packages (120?), but many more require some manual inspection before being viable.

We have attempted to clone Fedora spec file repositories directly, so we are also using their "sources"-files that list the names of source files to download before starting compilation. This caused the first of the most common difficulties. In our internal storage, where our source file fetching pointed to, there were only source code tarballs available. Many of the "sources"-files listed also some additional files like tests and patches.

When starting this process, we stuck with our current source package storage, as there is much work to be done in any case. This caused many errors, as builds failed due to files not being found. Manual intervention was needed here to correct the "sources"-files to only use source packages, since they are the only ones available.

Second of our major challenges was version difference between Fedora rawhide packages and source packages available in our repositories. There were cases where either one was ahead of the other and in many cases an older package wasn't even available in our repository. These cases need to be examined more closely to find out which version we actually want to use, or do we even have a choice because of possible compatibility issues.

In addition to these previously mentioned obstacles, we also encountered some build failures due to things like being unable to fulfill requirements (gcc version being the dominant one), missing static libraries and test failures.

In the next chapter, we will examine more closely how we adapted our storage approach to our new build process. While experimenting with a new, Fedora-based, operating system, we decided to also attempt moving towards upstream way of storing the source.

6 Reusing Fedora code base

We have started to examine the possibility of reusing open source spec files created for Fedora. The next iteration of our cloud is still at quite early stages, so the ties to its underlying operating system are still quite weak. The platform has already had test builds conducted on top of Fedora cloud image. There should not be massive Fedora specific changes that would cause need for rewriting. For these reasons, we can see moving directly to using Fedora packages to build our operating system to be viable. We can save a terrible amount of work hours when providing an automated way to achieve this task. Writing spec files for possibly hundreds of OS packages would require collaboration of many software teams to accomplish. We aim to be able to build the entire OS from scratch without any outside help.

As we have mentioned in this paper, whatever operating system we decide to go with in the future, we need to be able to control the sources and the build process. In practice, this means we need to build around 200 packages that are needed for the basic cloud platform image that we are going to be testing with. Creating spec files and moving the source packages for such a large number of packages sounds like a marathon. Reusing Fedora spec files as best we could and automating most of the source copying and build processes was the reason that we were able to accomplish our goal with limited resources. The developer tools that we have created during our research, along with new functionality designed to assist in this task, proved very valuable for achieving our goal.

Our current operating system has been attempting to keep its packages up to date with upstream with minimal delay. We have not had a long release cycle where release is frozen over time while carefully choosing the right versions. This is the convention in many open source distributions, and Fedora is not an exception. Instead, we publish regular releases of constantly evolving package base.

Often application developers can be slow to adopt new operating system features even if they are offered. The main reason to attempt to keep the operating system up to date has been that we want to avoid backporting patches. When critical updates are released, NetLinux developers have preferred to update the package instead of backporting patches into a version that has been released in our previous iteration.

Following this approach, we have decided to use the spec files of master branch, rawhide Fedora. Here most of the packages are newer versions than the ones used in NetLinux. However, if we decide to take this newly built operating system into use after evaluation, we may start adopting the Fedora release cycle. Having an operating system upstream where we can pull backported patches would ease the load of backporting patches.

In the following chapters, we will present how we have overcome the challenges of moving huge amounts of source code and rebuilding the packages internally.

6.1 Command for fetching source packages

We have made our own implementation for yet another rpkg command. This command, ‘rcppkg new-sources’, serves two purposes. First, it will greatly assist us in handling this massive source fetch operation. In later stages, the command would become quite useful in maintaining our new operating system. New versions of upstream packages are constantly being released. Thus, we need an efficient way of updating our packages before rebuilding them.

Again, we have attempted to make use of the functionality already provided by the underlying python superclass, rpkg. It offers functionality to parse the ‘sources’ file that lists source file names and their hashes in different formats (<hash> <name> or <hash type> (name) = <hash>). We utilize this function to find out the names of the files that we need to have present for building successfully using the current spec file.

Once we have the list of files that we need, we must find them from Fedora source package repository. There are two types of paths used there: <package base URL>/<file name>/<hash>/<file name> and <package base URL>/<file name>/<hash type>/<hash>/<file name>.

As there are different forms for the paths and the hashes that are part of them are obviously different for each file, we have decided to opt for a recursive approach. This way we can easily track down all the full URLs for the files that we are looking for. For each filename, we start the search from URL <package base URL>/<file name>. In each call of our recursive function, we find the list of files under that URL by getting index.html and parsing it from there. If we find a file that matches the name of the file we are looking for, the full path will be added to an array that will be returned by the function. Otherwise, the search will continue by looping through the found files and appending them one by one to the current URL before calling the function again.

Once we have found all the URLs of our all our source files, we will start fetching them to a temporary directory on local host. We start by forming the same directory structure, that is present on the Fedora repository, under the temporary directory. Then we copy the files to our local host before moving the entire temporary directory to the internal package storage residing on a remote server.

Now we have all the needed source files needed for building this package in our internal storage. With one command, we are able to automatically fetch all needed files. By checking the needed sources from the ‘sources’-file, we also avoid storing unnecessary files. If we were just to fetch all the files present in the Fedora source repository, we would store many old source versions that we certainly would have no use for.

When it comes to package maintenance, updating packages becomes extremely easy. First, the administrator will merge the spec file updates from Fedora repositories to matching internal spec file repositories. Then, ‘rcppkg new-sources’ will be run to fetch the possible new sources for each updated package. Now, new version of the package is ready to be built.

In the next chapter, we will look more closely into how we utilized this command in building Fedora packages internally on a larger scale.

An added benefit here is that we actually get the storage format we were discussing before. We now have own subdirectory for each of the packages' sources. How much additional benefit we can obtain from this in practice, will be seen in the future. For now, we are satisfied that the structure will greatly assist in the source mirroring process. The sources are now also much more easily browser by human actors.

6.2 Process of taking packages into use internally

Utilizing our collection of package maintenance tools, the process of taking an external package, that is present in Fedora repositories, becomes extremely simple. We can clearly separate four different steps that are needed:

10 Mirror Fedora spec file repository of the package into our internal git repository, under the spec file group that we have created. At this stage, you could also for example make changes to the spec files or patches if some Nokia-specific modifications are required.

11 Use 'rcppkg new-sources' to fetch the source packages indicated by the 'sources'-file in matching spec file repository from Fedora storage to our own.

12 Add the package to Koji for the tag that you are building to. This can be handled by issuing a single command to a shell that is authorized to use our Koji server: 'koji add-pkg --owner <user on Koji serve> <tag that we want to add the packge to> <package name>'.

13 Run 'rcppkg build' for the package.

We have formed a list of open source packages that we want to be included in our distribution. Using this list, we have written a script that will import the sources of each of these packages into our own storage and build an RPM for us. The script will simply loop through the package list and for each package, it will perform the four essential steps listed above. Now we can just leave it running for the night and see how we have hundreds of packages available internally in the morning. Doing all this manually would be extremely cumbersome and time consuming. Figure 2 depicts this full process of building listed Fedora packages internally.

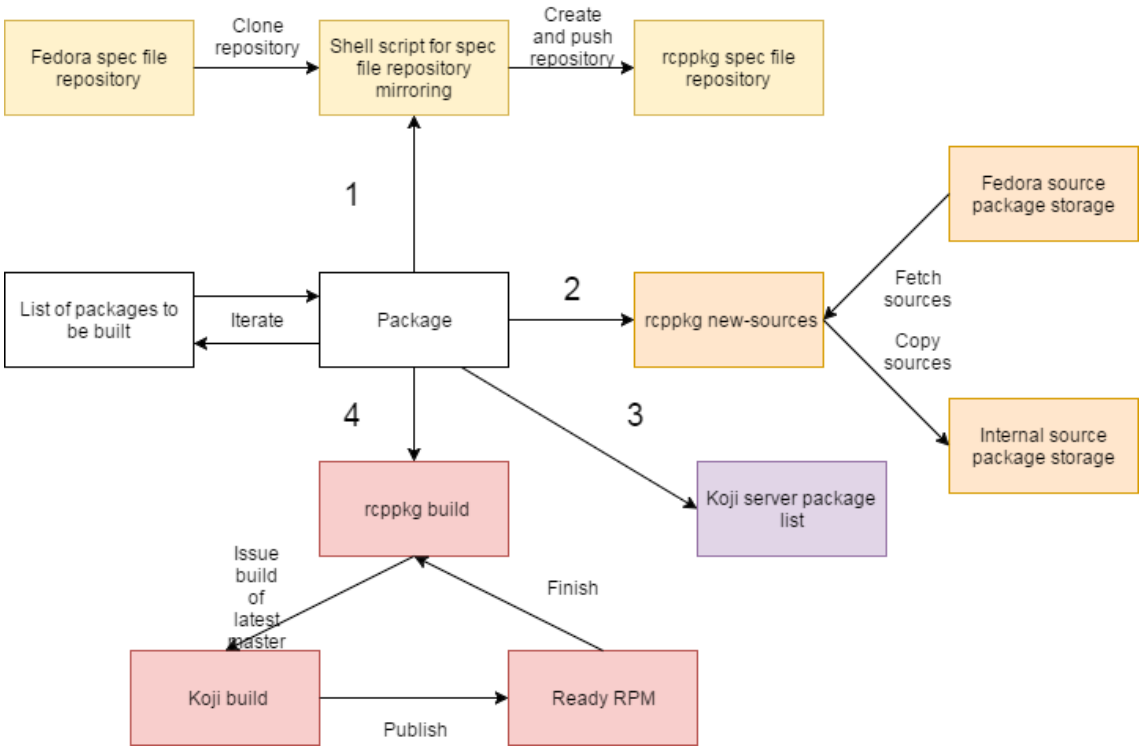

Figure 2: The process of importing a package from Fedora to internally control. The steps from 1 to 4 present different stages in the process required to implement and publish changes for a software component. Arrows in the figure present the direction of process flow and the actions that are being carried out.

After running such a huge job, we obviously have to examine the results. How was our migration able to perform? Most of the package builds succeeded, but there were also many errors that we had to track down and solve.

First thing that we noticed was that the disk that our build roots were created had insufficient space. For example, building Linux kernel failed due to filling the whole disk and not being able to continue. This was easily solved by setting our build root directory to a disk that had plenty of space available for even a large amount of builds.

A common cause of failure was that Koji was executing the build with root user. As building with root user is bad practice, many packages had some sort of check in place that caused the build to fail in case it was being run as root. Again, this issue was easy to fix by configuring Koji to opt for a non-privileged user.

Some less favorable steps we had to take were disabling tests or building debug RPMs for some packages. We took this shortcut in order to start testing with our new OS build process from start to finish. Making sure that all tests that should be successful, are successful, should be our next priority after we find the time for it.

As our Koji had been configured with Fedora 25 repositories, there were some unmet dependencies when building rawhide packages. We started by building rawhide versions of these missing dependencies to be used, we were able to overcome the issues and proceed. This approach, however, would

return to punish us later as we will find out.

Another similar, but not exactly the same issue was that dependants of packages that we had built were sometimes unable to build because they needed an older library to succeed. For example, initially, we had built version 1.63 of boost. There were many packages that were fetched from the public repository and wanted to use boost 1.60 libraries, as that was what had been in use at the time of Fedora 25. Their builds would obviously not succeed, as long as their requirements were not met. The hard version requirement caused our newer version to not be accepted.

As the number of version issues began to grow, we configured Koji to prioritize a rawhide dnf repository when resolving dependencies. This revealed a number of compatibility issues that we had not taken into account. Python abi (application binary interface) version changes in between and conflicting pkgconfig versions forced us to rebuild some of our packages to be able to finish building our list of packages.

6.3 Building OS images

Having built all the packages that we wanted to include in our new operating system, we were to proceed to building the OS itself.

We have been given a kickstart [Red17b] file that we can use for building the operating system image. With this kickstart file and a small script to initialize the build with correct parameters, we were good to go. The kickstart file is actually where we copied the packages that we need so part of the contents were familiar to us already. There were some differences, as Fedora has own naming conventions for packages. However, with a bit of modification were able to create a list that contained only names that were found in Fedora repositories.

The kickstart file had to be configured with an URL where to install the packages from. To be able to do this successfully, we had to create a dnf repository from our packages. We had been instructed to use mash [pag17] for this. This way we had a yum repository containing the packages we had just built up in no time.

There was just one failure during the process that had to be corrected here. This was to change the repository location, as it had been configured to place the repository on the root disk. This had not been an issue before since the number of packages had been much smaller during earlier testing. As with kernel compilation, the operation was unable to complete, as the disk was filled completely before even getting close to finishing.

After configuring the newly built dnf repository into our kickstart file, we attempted building the OS for the first time. This resulted in a quick failure. After examining our first attempt and comparing our package repository to another one, the cause was evident. We must also include a boot media for the

build to use in the repository that we are building from. Copying specific files from a repository that had been used for building via kickstart before, resolved this issue.

Next, we faced an isse with dnf groups of our repository. The kickstart file that was provided for us required the dnf repository to contain two groups, core and cloud-server-environment. These two groups are used for building the Fedora cloud base image. We proceeded to create a configuration file that held the group data and recreated our repository.

Now it seemed as the biggest challenges were over, but in fact there was still problems to be tackled. Our build was failing and the debug information given by Koji was quite minimal. The only debug data that we are given on installation failures like this, is a screenshot of the screen at the time of failure. In this case it was not useful for us as it just indicated that software installation had failed and asked if we want to proceed. The installation fails in this case because Koji installation must be fully automated. No input can be given, so the build will time out and fail.

We then turned to investigate our package list. Quickly, we noticed that the package list was very lacking when used for building a repository. Yes, it listed all the packages that we needed, but their dependencies had not been listed. This means that we had not built those packages at all and therefore, there was no way they would be found from our new dnf repository. We had to proceed to finding out the missing dependencies for all of our packages and recreating our repository.

This task was started by utilizing a script that we wrote to find unique dependencies. The script iterated through all of our packages, searched their dependencies and then searched for providing package for each of these dependencies. The output contained require RPMs names. As one package can contain multiple subpackages, we had some manual work ahead of us. Our build system needed the package names instead of RPM names to function properly. After the scripts completion we still had to manually scramble through the list, replacing the RPM names with the corresponding package names.

As we were proceeding with this build, we noticed there were still issues with pkgconfig versions as many packages had been built with the Fedora 25 packages fulfilling dependencies. After moving to rawhide base repository, these builds no longer succeeded. There were conflicts between pkgconf, that was installed by the dependencies and pkconfig, that part of the packages were still using. Thus, we decided it was best we rebuilt the whole package base against rawhide repository, track down and correct these issues, in order to make sure that there were no more version mismatches left.

This sounds like a huge time investment, but thanks to the available tools and ready-written scripts, this task was easily completed and we were able to continue with our OS image build. In fact, we created a whole new build target and tag to Koji in order to have a clean start. This way we made sure there were no leftover packages as we had added some packages that were not actually necessary during development.

After the complete rebuild of packages, the next step was again to attempt to build an OS image out of them. And again, we faced similar failure. It was clear that we were not able to capture all of the

required dependencies with our script. We decided to investigate the issue in a more manual manner, to ensure our success.

We utilized virt-install to do test build with the same kickstart file and the same repository that we had created earlier. It is clear that this way, we are not able to determine Koji related issues. We were confident that this was not an issue in this case. We had successfully completed image builds with the same kickstart file when building from public repositories. Thus, the most likely cause for failures was the repository that we had created.

The image build again failed and expected an input from us, just as when building with Koji. However, now we were able to examine the system more closely. We dived into the logs of the system and quickly found a long list of missing packages that prevented our installation. After writing these down, we proceeded to build the newly found dependencies.

We soon found out that attempting to find out rest of the dependencies was a very poor decision. The dependency chain seemed to continue endlessly, there was no sense trying to do all this manually. We changed back into the automated approach, and executed manual installations every once in a while, to see how many dependencies the installation is currently missing. When there were only a few dependencies left, we were able to pinpoint them with virt-install and complete the task at hand.

In the end, there were even more dependencies missing than there were packages in our OS installation. The kickstart contained a list of around 200 packages with some additional packages defined by the default dnf groups included in Fedora installation process. The dependencies that we tracked down, turned out to include over 300 separate packages.

We had in no way anticipated that the number of missing dependencies would be so large. For this reason, we believe that our way of handling this task was far from ideal. The end result could have been accomplished with far less effort if approached correctly. We were expecting a relatively small number of missing dependencies and the missing packages that were listed for us after each OS installation attempt stayed quite small after a while. This lead us to believe that we were really close to having built all the required packages. Due to this, we did not attempt to take steps to achieve greater automation for the process. Almost all the way through, we were convinced that there was only a few more packages to go.

For future reference, it should definitely be kept in mind that this kind of dependency tracking should not be attempted manually, nor in steps, as we have done here. An automated process should be developed, so that the amount of man hours can be kept low. One simple possibility is to use the script that we utilized as a basis. However, instead of exiting after looping through one round of dependencies, the script should keep running until such an iteration is executed that no additions at all are made to the list. This way we will have to do close to no manual effort at all, as long as the dependencies are listed correctly in the system that the script is being run on. Repository version differences can cause some packages to be left out causing a bit of manual intervention to again be applied. Still, this

will be very minimal in comparison to what we have done.

6.4 The results

Finally, after struggling to build the needed dependencies, we were able to successfully build an OS image from internally built packages. By using the same kickstart file that Fedora uses and adding the packages that we want to include, we were able to test the new qcow2 image.

By mimicking the Fedora build configuration, we were able to run the test image using qemu without larger issues. The operating system booted up fine with only one service failing, systemd-vconsole-setup.service. We suspect that this is due to the qemu arguments that we used and should be easily fixed when needed to be able to take the images into real use.

We are now able to build Fedora images completely separately from the public resources. This enables the possibility to actually productize the operating system and possibly replace the currently used NetLinux all together. However, there will be many challenges ahead of us in porting the required functionality. Most likely the biggest being that we are using Fedora rawhide package base, our image will be well ahead in terms of software versions. A lot of adaptations will be needed to be able to compile and run the current platform software on the Fedora rawhide image that we have built.

7 Development platform

We want to offer our developers a modern way to complete their programming tasks. Currently the development environment is usually completely separated from the platform where the developed application will be launched. We want to offer better tools to execute software development on the platform. This makes your workflow leaner, as you do not have to move the code around to be able to test it in its native environment[HHA08].

Software development was previously carried out on centralized environments. Then, with the popularity of personal computers the transition was made towards a distributed fashion where each developer was in change for their own environment [Cla92]. Now, we often complete our work in between these two approaches. Developers are offered with easy-to-setup platforms that contain different development tools for them to use. This is the approach we would like to implement for our native development platform as well.

To comply with what DevOps suggests, unifying development and production environments will be an important step. With the new tools and operating system, achieving this will be much easier and the results will be a lot better.

7.1 Approach

As opposed to our previous work, we do not aim to control all the source code. We want to use the same core operating system packages as we are using in our production environments. These will obviously be compiled from the same sources as in our products. However, we do not have to control those parts of the development environment that will never be used in production. We can just utilize readymade RPMs from upstream repositories.

This way we can achieve the greatest amount of customizability. If we were to build every package ourselves, we would soon find ourselves swarmed in work. As people have different preferences on what kind of development tools and desktops to use, we would have to maintain a huge package base that would offer us no direct benefit into our products.

The reasons why we want to maintain the source code, do not apply for development packages. Development environments are in no way as critical. New features and bug fixes do not have to be included instantly. Not offering all the optimal approaches is not a big deal. We can safely take the route that will cater to the widest variety of needs.

Most importantly, it will now be simple to use the production VM that we will provide and install a graphical user interface on top of it from Fedora RPMs. The required efforts to achieve this with NetLinux would be enormous, as it does not contain a package manager at all.

To be able to benefit from unifying the platforms, we must make sure that those packages that are used in production, will be used as they are in the provided development platforms. Outside sources can then be used for installing additional software that the users' personally like to use.

As we are still based on Fedora rawhide, the constant updates will yield the platform quite unstable and as such not very desirable to be used by developers. However, we plan on creating stable releases of our Fedora fork for production use in the future. At this point, it will also be a highly viable option for developers to adopt to be used daily.

8 Package update management

While we have already covered most of the tools needed to achieve the proposed workflow, we are still missing the final touch. Our own dnf repositories are still being created manually and the OS images need to be built from within the Koji server.

For the developers to be able to complete the full process from making code changes to releasing a new platform image, we must offer a better way to take the final steps. Luckily, there exists a suitable tool for this purpose in the Fedora release engineering software family.

Bodhi [Red17c] is an update management software that is part of the Fedora build system. It offers a web user interface that can be used for adding, updating and removing packages in dnf repositories.

The features offered by Bodhi include automated testing, notifications of new packages and highlighting security updates. These are features that should be utilized in every large software project and Nokia obviously has own ways of providing the listed services. However, in many cases the used tools are outdated and not nearly as effective to use as they should be in modern times. What we want to achieve, is better integration among tools used in our build process. Placing these services in a single place and offering an easily approachable user interface to operate them offers great benefits. Users will no longer have to familiarize themselves with multiple different tools and software maintenance tasks can be achieved with higher effectiveness.

Packages that have been added to Bodhi, will move through four different stages during the time they are present:

- PENDING
- TESTING
- STABLE
- OBSOLETE

Before adding a new package, developers will be able to indicate what is the purpose of the new version and whether it fixes any known bugs. When the addition is detected by Bodhi, it will initially be tagged as PENDING. In this stage it the package will be simply waiting to be moved onto the next step, TESTING. Once the package has been pushed to an “updates-testing”-repository it will become usable by the end-users. Now, this package can be pulled and installed by the early adopters that are willing to participate in testing the newest releases. These testers can then give comments on the new release and vote on its viability via a “karma”-system. This means simply giving the package either a plus or a minus vote based on the conducted testing. After getting enough positive karma, a package can be tagged as STABLE and transferred into the main repository. This is the main repository, where most users will fetch their updates from. If package gets so much negative feedback that it seems to not be viable to be used, it can also be deleted from the system at this stage. At the end of the lifecycle

of a package release resides the OBSOLETE stage. Once a newer version of the package is tagged as STABLE the old stable package will be marked as OBSOLETE and will no longer be available in the main repository.

9 Corporate context

What we want to do, is to be able to make use of the achievements of open source community. This way we can avoid redoing a lot of work that has already been completed elsewhere. Open source software usually operates in a very loose environment. This makes the circumstances very different from ours. Many boundaries and guidelines are enforced on our development practices, environments and processes.

There are a lot of things that have to be kept in mind, when transferring software between such different operating environments. In this chapter, we are going to sum up our notions about the special requirements our working environment sets for us. What are the major differences between open source projects and corporations' internal projects. What are the steps that we need to take to adapt the software to specific corporate requirements?

9.1 Storing source code internally

As we have stated, we must store all the used source code in internal repositories. Whether we are handling in-house software or common open source components, they all must be located withinin our own servers.

Most open source components are stored as tarballs and possible internal changes are applied to them from patch files before compiling them. This practice is the most convenient one, as there is usually no major changes that need to be implemented. Upstream pull requests are also often created for the patches. For this reason, most of the patches are temporary as they will be merged to an official release at some point. Still, there are also cases when the changes are too specific, so upstream will not accept them into their releases.

The proprietary software developed in-house is of course stored in various version control systems. These repositories are what the developers conduct most of their work on. There is no real alternative for this storage method on actively developed packages. There must be an efficient way to track the development process. This is achieved by always storing a clear history of the implemented changes for each package.

Open source Linux distributions either just fetch the used source packages from the specific upstream storage or use own storage where source packages are copied from upstream. Each upstream project stores their own code in the place of their choosing where everyone can access this code. Therefore, when building an open source distribution, if the first approach is being used, source code is being pulled from all around the web. This approach is out of the question for us.

Large, profit-making projects need to ensure high availability of the software repositories that are used for storing source code. Corporate build processes need to be reliable to achieve continuous integration

[Tho17]. In the future, the requirements may very well evolve towards continuous delivery which by no means lessens the availability requirements.

Enterprises also have to be able to ensure that the used code cannot be tampered with. The security of our software is essential in upholding customer trust and the quality of our products. Possible points of unauthorized entry into our software products must be minimized. Obviously, security is essential for any party. Still, the effects of vulnerabilities can be much more devastating for software products of corporations than those of open source projects.

Various projects can also gain other benefits from storing code internally. It can make it possible for the build systems to rely on convention, rather than configuration. Reducing the time and effort to set up software on new environments can be beneficial in many ways. The amount of work required from developers to integrate new software components to the build system can also be reduces. Any time saved can always be used elsewhere.

Internal storing is how the software sources are handled currently as well, but we need to be aware of these things when implementing new tools. We have already taken in use a proposal for a new storage method. The requirements that must be met have to be clear before conducting evaluations that help choosing the right approaches for the tools that are being developed.

9.2 Accessing proprietary source code

In the open source community, any code is incorporated in the projects, can be checked out anonymously. In other words, no authentication methods are required to be set. This can apply many corporate components as well. While all the product source code is stored internally, a lot of it can be fetched without any credentials. Especially the open source packages are often handled this way. However, in-house subsystems usually need to be protected from unauthorized access. Copyrights need to be protected and competitors must not be allowed to access the source code of proprietary software.

Corporate repositories often reside within internal network so there is no direct threat towards stealing code from them. Still, adding an extra layer of security by requiring authentication, even when requesting access from within the internal network, is a good additional security measure. It should also be noted that if no authentication was required anyone with access to the internal network could pull the sources. For example, an outside consultant could get access to information that is not related to his work. This, conducted either intentionally or unintentionally, poses a threat of unwanted parties acquiring private source code.

When cloning internal [Pro17] repositories in automated scripts, ssh key authentication should be set up on the system that you want to clone the code to. When checking out repositories manually, internal credentials need to be given in case of cloning via https. As we want our supporting tools to be automated and simple to use, required user interaction should be attempted to be kept to a minimum. This

is why, the tools that are created for automation, should be able to utilize ssh keys. Having to input credentials multiple times a day can become taxing.

There is no way, that we could avoid the need of offering support for authorization in the tools that we are developing. Even if the source code would not be stored in internal repositories, many outsourced components would have to be. The owners of these components definitely would not like their clients leaking proprietary software that they provide.

What we also need to keep in mind, is that we must support checking out code from multiple locations with many different access methods. We must be able to handle git repositories as well as tarball stored on webservers and fetched via wget. In addition to these differences, there are different approaches to parsing the source location, depending on the package.

Usually, finding out the right fetching method is not hard at all to achieve. In most cases, it is enough to use some specific configuration files to issue authentication information to be used. At worst, we have to make code changes to some of our tools just to support authenticated code checkout. However, at the moment, users' existing authentication methods, are enough.

This keeps the required amount of job minimal. So far, we have had to make changes related to this requirement only to tools used in our build system. If we were to face making changes to developer tools, the convenience of our approach could suffer.

Clearly this characteristic of our own environment is not very challenging to overcome. However, it affects many parts of our build system. We must be aware of this and keep it in mind throughout the implementation of new features.

9.3 Source package control

Software products almost never use the latest versions of open source packages. There are always stability concerns in adopting newly released packages and sometimes the teams responsible just might not have time to update every package they would want to.

Another thing that we must account that sometimes even different file extensions cause trouble. For example, there are many packages that are packaged in tar.gz format on our internal storage but Fedora spec files are configured to use tar.bz2. These cases are easy to fix by reconfiguring the name in spec files and recalculating the checksum. Still, with this high number of packages, any extra work required for each of them, turns into a lot of work.

Different packages also have different conventions and use additional source file in addition to the source code. In these cases, we again must modify the configuration files before building and add extra files into our spec file repository. Were we to control the package repository, adding packages and structuring them conveniently would be easy.

Thus, trying to compile the same packages that NetLinux uses, with Fedora rawhide spec files becomes challenging. Depending on the case, we may need to use an older Fedora release for that package or make manual changes.

One thing that causes a bit of extra effort for us is that we do not control the source used for building packages. The sources are stored on a web server, and fetched for builds. We might have a chance of examining some packages and updating them according to our needs. However, as we have progressed with creating Koji builds for new packages, it has become evident that the version differences are widespread. Updating packages on this scale is not possible for us. The team that is hosting the source package repository would not like to see that many new tarballs suddenly emerging.

These are issues that we were also able to overcome by reimplementing the source package storage. Using Fedora source files directly obviously ensures that the versions and file names are exact matches.

One challenge here is that as we are attempting to present a new operating system to be used with our cloud product, we should attempt to maintain versions are in use in the old system (NetLinux). Surprising problems can arise from updating packages and more so when issuing version changes for over a hundred packages at once. Integrating the same applications to run on top of the new operating system could become a nightmare.

After initially following the package versions used by NetLinux, we decided to move on to using Fedora rawhide versions. With this decision, we also accepted the fact that we will not be using these tools for the current platform. At this point the decision was made to aim at composing a new operating system to be one option to be used in the next version of our platform.

The decision that we were allowed to take into use our own, structured source package repository helped us immensely in building all the necessary packages. However, the possibility of not being able to control the used versions should be kept in mind. There are often differing opinions and we will not always be able to do exactly what we want.

10 Discussion

In this chapter, we will briefly discuss what this work implies in practice. We will iterate through the research questions presented in the beginning of the paper and try to clarify how we have addressed the related issues and what remains to be solved.

Serving as a basis to the other research topics we have attempted to find answers for the research questions 03 and 05. We have summarized the findings related to research question 03 in chapter 9, but additional information can be found throughout the paper. Before jumping to building the new operating system, we spent some time on researching the possibilities on how to store the needed building blocks in chapter 5. While this was only a brief glance into research question 05, it proved quite useful in our work.

We have created a simple image to offer a clear picture of the new developer workflow that we aim to offer. As seen in Figure 3, we propose a workflow of three different steps. Currently steps one and two are most easily approachable. These steps are no doubt the most commonly repeated ones, so we want to offer the best support for them.

Using rcppkg, developers are able to implement, test and publish changes to RPM packages with minimal effort. They are free from worrying about necessary initial steps such as setting up RPM build environment, acquiring build dependencies or finding the correct source and spec files. We have stored a source code dump of current state of rcppkg to github [rcp17]. While we have removed some parts and the commit history to ensure no protected information is leaked, this should offer a fair basis for usage in any operating environment.

With our work on the tool set, we have also approached the likely future requirements. In our attempts to provide answers for research question 03, we have taken the first steps towards container build functionality with the same tool set that we use for building RPMs.

Step three still requires manual effort. A developer must access the Koji server, mash new dnf repository and issue a “koji image-build”-command. This is simple enough, but there is still room for improvement to streamline our image delivery even more.

The previously mentioned process is our view for solving the problems related to research questions 01 and 02. We believe that with the proposed tool set developers with be able to conduct their work much less manual effort. Our take on the research question 08 can be found along the progress we have made towards the developer tools, written down in the previous chapters.

We need to work on getting our own Bodhi deployment operational before our tool set is complete. However, our testing has proved promising and we are already building functional platform images for demo purposes using our new tools.

Our initial demo image included Fedora cloud base packages and a chosen set of subsystems from the

upcoming 2.0 iteration of the cloud platform. We wanted to just include the essential core services. This way we were able to keep the services up and running in their new operating environment without too many issues.

We will continue to build more needed functionality into our new Fedora based image. We will need to start by recompiling Linux kernel with internal configuration and build more internal subsystems into RPMs to be able to offer the full functionality of the platform at some point. We also want to have the existing test packages built and in use as soon as possible to be able to validate our image all through the way of including more and more platform functionality.

In addition to providing the new Fedora image, we have laid out the groundwork for the transition towards it's full blown usage. We have tried to find the answers for research question 06 and make the necessary preparations, so that the operating system could be adopted with as little disturbance as possible. Our target for the near future is to provide all the required platform services on top of the internally built Fedora image.

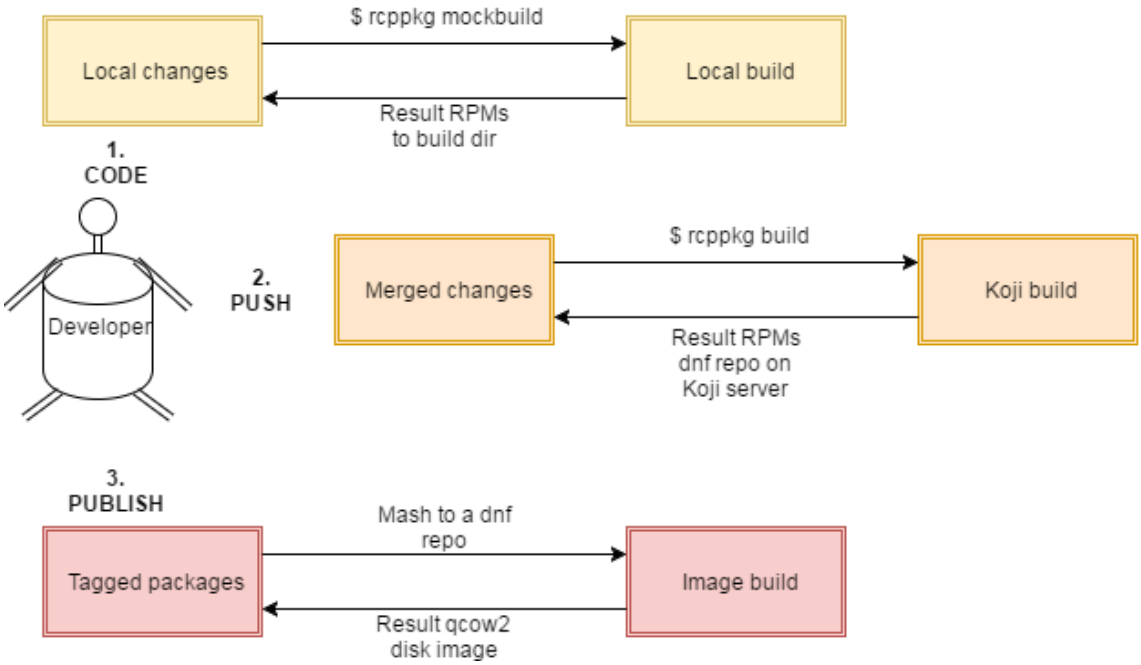

Figure 3: The developer workflow that results from using the tools introduced in this paper. Parts 1, 2 and three represent the three distinct tasks required from a developer to be able to implement and release code changes to a software package. The states enclosed in the squares can be considered the most vital actions needed to accomplish the task at hand and the arrows represent interaction between the actions carried out by the developer.

To be able to find the answers to research questions 07 and 09, we turned to the state of the art DevOps research. By examining the recent developments and trying to find the best practices has been the main focus in preparation for attempting to figure out how we will approach the issue.

According to DevOps guidelines, similar environments should be used for both development and production purposes. Previously this has not been possible for us. NetLinux has proved quite difficult to adopt for day-to-day use and thus people have not been willing to start using it for development. If we are able to push this Fedora operating system into production, adopting it for development should not be an issue. With a small amount of effort, we can offer desktop packages and start building images that include graphical user interface and some essential development tools.

A huge issue that we have discussed during this paper, has been offering different flavors of the operating system. All different user of our platform, have been provided with the same OS packages. With the current build process, changing this would require immense efforts. With our proposed tools, creating the optimal OS image for each use case will become easy. When the build process is configured via kickstart files, the only thing you have to do, is to modify the package list there. This is a task that pretty much anyone is able to complete, but clearly, decisions on what packages to include, should be made carefully.

In addition to setting up and taking in use Bodhi, we are still facing the need to deploy our Koji server on a better platform. Currently it is running on a development virtual machine where processing performance, disk space and availability are quite limited. Before we can offer our new build tools to wide developer use, we must acquire resources to be able to offer better service.

11 Conclusions

During this research, we have been able to implement solutions to most of the problems that we set out to solve in the beginning. As a basis, we have examined what things we need to take into account, when taking into use the new build system and developer tools. The goal of most of our tasks has been to start adopting DevOps practices in one way of the other. We offer a developer tool that allows users to interact with the basic functionality of the Koji build system easily, while supporting the needed version control systems that the developers interact with daily. We have tested the basic setup needed for building containers with the same tool set that we are introducing. Two future lines of research have also been listed. These should be looked into as soon as we handle the more pressing matters at hand.

Most importantly, we have internally built a Fedora image and started including the cloud platform packages on top of it. With this work, we are rapidly moving towards being able to replace the current NetLinux distribution with Fedora that we have built.

We have also taken in use a new source code storage for storing the open source packages that we are using. The new directory layout has proven to be more than adequate to fulfill our ambitions for easy adoption and better structure. Along the way, we have come across special requirements that we face while implementing systems for internal developer use. These are listed and discussed in a dedicated chapter of this paper to make it easy for us return to examine them when implementing new build system features and tools to be used.

The first development task that we took on during this research was implementing a development tool to offer a simple way for changing, testing and publishing code. As we started with this project, we have been able to utilize the developer tool while working on the implementation of other parts of the build system. This has allowed us to constantly keep evaluating and improving the tool and its available features. We have been making changes to adapt to new features elsewhere in the build system, fixing bugs that we have located while operating the tool and cleaning up the code. We are confident that the core features work well and a few other users have already tested the tool on a few scenarios. Still, before packaging rcppkg to be available for development use, we would like to conduct a little wider testing on it. It is likely that we at least have to improve on error checking and user interaction.

We have set up the minimal deployment for building containers and demonstrated the process of creating Docker images with rcppkg. While this functionality is not as mature as the other topics discussed in this paper, we are confident that laying out the basics will help us in the future. Should we opt to keep using Koji build system while using containers for production deployment, the developer tools are already present and groundwork has been done to handle the initial setup of the build environment. Right now, this topic will most likely be left to the background while we focus on offering support for adopting the newly built Fedora and the build tools discussed in this paper. When packaging applications into containers for deployment starts to become timely, we will surely come back to examine the

viability of using the combination of Koji and OpenShift.

Creating a new source code storage proved vital in building the packages needed to offer the internally built Fedora cloud base image. Because of this decision, we were able to directly reuse almost all of the needed spec file repositories. Without the simple automation that the mirroring has enabled us to achieve, we would surely have needed much more time to be able to build the hundreds of packages that we now have available. The new directory structure that we have placed the packages in allows us to better grasp the large package base that is already present. We look forward to see if we will utilize the new tree structure in more ways.

The differences in setting up the discussed build tools in a corporate environment versus an open setting have proven to be surprisingly small. There has really been no major obstacles to overcome. The smaller challenges that we have come across in taking tools into internal use have been quickly resolved and documented in this paper. We admit that the resolutions of the problems are not optimal in all cases and should be examined before setting up an environment to support production builds. For now, the current workarounds will work just fine as long as we keep in mind the additional steps that we need to take whenever setting up new build hosts.

Testing the Fedora image that we have built has yielded very positive results during the early phases. At the moment, a single failing OS level service is present and this has been marked as a release blocker for Fedora 26 so we expect to see an upstream fix shortly. It is also not relevant for the currently installed platform services and does not even emerge on all deployment platforms. For these reasons, we have not looked more closely into the bug ourselves and have continued to work on including more internal platform subsystems in our image. The initial testing has not yet revealed any major problems in changing the underlying operating system. Most of the test cases that have been run on top of the Fedora cloud platform deployment have succeeded. The currently failing test cases are known to also fail when the platform is built on top of NetLinux. We are very pleased with the current state of the Fedora based platform and will keep on striving towards being able to use this OS for official builds.

We would also like to note that with the tools and means introduced in this thesis, it is quite simple to create a Fedora fork of your own and keep maintaining it with minimal effort. The biggest challenge will be setting up a personal Koji server because we have not covered this topic here. A good start will be provided by koji-setup of Russian Fedora project [Rus17]. This was used as a basis for our internal deployment.

12 Future work

Of the two topics left for later implementation, one should prove really simple and the other quite demanding. Offering the internally built Fedora operating system for development purposes should not take too much time. The people that are continuing on this task, will be able to take the production Fedora image that we have built and simply install the needed development tools on top of it by using dnf. Most time should likely be spent on finding out what development tools and desktop environments to offer for the users to choose from. Setting up Bodhi to offer the simple update management and image building functionalities, will most likely require a lot more effort. The available documentation is limited to say the least. If we do not account for luck, achieving a production-ready setup will consist of mostly reading the source code and taking small steps forwards through trial and error. Additionally, it still is not clear what other software tools we need to include in our setup from the Fedora release management family, to be able to offer full Bodhi functionality. Thus, it is hard to determine the required amount of effort beforehand. However, it is clear that this task will not be completed over night and it should be left for a time when we have more resources available.

References

ABD16 Artač M., Borovšak T., Di Nitto E., Guerriero M., Tamburri D. A., Model-driven continuous deployment for quality DevOps. Proc. 2nd International Workshop on Quality-Aware DevOps, Saarbrücken, Germany, July 2016, pages 40-41.

ADT11 Abate P., DiCosmo R., Treinen R., Zacchiroli S., MPM: a modular package manager. Proc. 14th international ACM Sigsoft symposium on Component based software engineering, Boulder, Colorado, USA, June 2011, pages 179-188.

Alp17 Alpine Linux Development Team, Alpine Linux. https://alpinelinux.org/ [22.5.2017].

Ata17 Atarakt, Koji. https://fedoraproject.org/wiki/Koji [20.1.2017].

Bee16 Beekmans G., Linux From Scratch. http://www.linuxfromscratch.org/lfs/view/stable/ [2016].

BFH16 Brown A., Forsgren N., Humble J., Kersten N., Kersten N., Kim G., 2016 State of DevOps Report. https://puppet.com/resources/white-paper/2016-state-of-devops-report.

BMS00 Bailey E. C., Nasrat P., Saou M., Skyttä V., Inside the Spec File. Maximum RPM: Taking the RPM Package Manager to the Limit, http://rpm.org/max-rpm-snapshot/ch-rpm-inside.html, Red Hat, Inc, 2000, chapter 13.

Can17 Canonical Ltd., LinuxContainers.org Infrastructure for container projects. https://linuxcontainers.org/ [7.2.2017].

CaZ14 Caneill M., Zacchiroli S., Debsources: live and historical views on macro-level software evolution. Proc. 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Torino, Italy, September 2014, article no. 28.

CaZ17 Caneill M., Zacchiroli S., Debsources. https://sources.debian.net/, [31.1.2017].

Chi14 Chibon P., Pkgdb2. http://pkgdb2.readthedocs.io/en/latest/.

Cla92 Clark T. D. Jr., Corporate systems management: an overview and research perspective. Communications of the ACM, vol. 35, issue 2 (February 1992), pages 61-75.

CYC14 Cois C. A., Yankel J., Connell A., Modern DevOps: Optimizing software development through effective system interactions. 2014 IEEE International Professional Communication Conference (IPCC), Pittsburgh, PA, USA, October 2014, pages 1-7.

DiV11 Di Cosmo R., Vouillon J., On software component co-installability. Proc. 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, Szeged, Hungary, September 2011, pages 255-256.

DJG16 Di Nitto E., Jamshidi P., Guerriero M., Spais I., Tamburri D. A., A software architecture

framework for quality-aware DevOps. Proc. 2nd International Workshop on Quality-Aware DevOps, Saarbrücken, Germany, July 2016, pages 12-17.

Doc17a Docker Inc., Dockerfile reference. https://docs.docker.com/engine/reference/builder/ [12.2.2017].

Doc17b Dockerfile. https://github.com/nipakoo/atomic-reactor/blob/master/Dockerfile [22.2.2017].

ECA16 Escobar D., Cárdenas D., Amarillo R., Castro E., Garcés K., Parra C., Casallas R., Towards the understanding and evolution of monolithic applications as microservices. Proc. 2016 XLII Latin American Computing Conference (CLEI), Valparaiso, Chile, October 2016, pages 1-11.

ECK16 Esposito C., Castiglione A., Choo K. R., Challenges in Delivering Software in the Cloud as Microservices. IEEE Cloud Computing, vol. 3, issue 5 (November 2016), pages 10-14.

EGH16 Ebert C., Gallardo G., Hernantes J., Serrano N., DevOps. IEEE Software, vol. 33, issue 3 (April 2016), pages 94-100.

Fet16 Fetzer C., Building Critical Applications Using Microservices. IEEE Security & Privacy, vol. 14, issue 6 (December 2016), pages 86-89.

Fri08 Fritz T., Composing Knowledge Fragments: A Next Generation IDE. Proc. Companion of the 30th international conference on Software engineering, Leipzig, Germany, May 2008, pages 999-1002.

Fus17a Fusion809, Yum. https://fedoraproject.org/wiki/Yum [6.1.2017].

Fus17b Fusion809, DNF. https://fedoraproject.org/wiki/DNF [17.1.2017].

GBS16a Gilmore D.. Babincak P., Sedlář L., Qi C., Fenzi K., https://pagure.io/fedpkg/.

GBS16b Gilmore D.. Babincak P., Sedlář L., Qi C., Fenzi K., https://pagure.io/rpkg.

GCG15 Guerriero M., Ciavotta. M., Gibilisco G. P., Ardagna D., SPACE4Cloud: a DevOps environment for multi-cloud applications. Proc. 1st International Workshop on Quality-Aware DevOps, Bergamo, Italy, September 2015, pages 29-30.

GiS13 Giessmann A., Stanoevska-Slabeva K., What are Developers' Preferences on Platform as a Service? An Empirical Investigation. Proc. 46th Hawaii International Conference on System Sciences, Wailea, Maui, HI, USA, January 2013, pages 1035-1044.

Hea12 Ben Hearsum, Bug 770990 - make ssh keys available to mock environments. https://bugzilla.mozilla.org/show_bug.cgi?id=770990 [12.7.2012].

HHA08 Han A. H., Hwang Y., An Y., Lee S., Chung K., Virtual ARM Platform for Embedded

System Developers. Proc. International Conference on Audio, Language and Image Processing, Shanghai, China, July 2008, pages 586-592.

hos17 hosts. http://man7.org/linux/man-pages/man5/hosts.5.html [18.2.2017].

JBP16 Jabbari R., Bin Ali N., Petersen K., Tanveer B., What is DevOps?: A Systematic Mapping Study on Definitions and Practices. Proc. Scientific Workshop Proceedings of XP2016, Edinburgh, Scotland, UK, May 2016, article no. 12.

JMN17 John W., Marchetto G., Nemeth F., Skoldstrom P., Steinert R., Meirosu C., Papafili I., Pentikousis K., Service Provider DevOps. IEEE Communications Magazine, vol. 55, issue 1 (January 2017), pages 204-211.

Joy15 Joy A. M., Performance comparison between Linux containers and virtual machines. Proc. 2015 International Conference on Advances in Computer Engineering and Applications, Ghaziabad, India, February 2015, pages 342-346.

KBB16 Khazaei H., Barna C., Beigi-Mohammadi N., Litoiu M., Efficiency Analysis of Provisioning Microservices. Proc. 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Luxembourg, Dec 2016, pages 261-268.

KLZ14 Kuebel H., Limbach F., Zarnekow R., Business Models of Developer Platforms in the Telecommunications Industry – an Explorative Case Study Analysis. Proc. 47th Hawaii International Conference on System Sciences, Waikoloa, HI, USA, January 2014, pages 3919-3928.

Kor17 Kortström N., Added ssh key and known hosts to be copied from root home to chroot home. https://github.com/nipakoo/mock/commit/f17d95f89112b22d266fb167468ad039bc01651d [7.2.2017].

KVL16 Kärpänoja P., Virtanen A., Lehtonen T., Mikkonen T., Exploring Peopleware in Continuous Delivery. Proc. Scientific Workshop Proceedings of XP2016, Edinburgh, Scotland, May 2016, article no. 13.

Lin17a Linux Kernel Organization, Inc., The Linux Kernel Archives. https://www.kernel.org/ [19.2.2017].

Lin17b The Linux Foundation, Kubernetes. https://kubernetes.io/ [14.2.2017].

LZX16 Liu D., Zhu H., Xu C., Bayley I., Lightfoot D., Green M., Marshall P., CIDE: An Integrated Development Environment for Microservices. Proc. 2016 IEEE International Conference on Services Computing (SCC), San Francisco, CL, USA, June-July 2016, pages 802-812.

Mah13 Mahmood M. S., Create a Fedpkg compatible Package Repository for Pidora.

https://wiki.cdot.senecacollege.ca/wiki/Create_a_Fedpkg_compatible_Package_Repository_for_Pidora [15.12.2013].

Man14 Mani S., Improving enterprise software maintenance efficiency through mining software repositories in an industry context. Proc. 36th International Conference on Software Engineering, Hyderabad, India, May 2014, pages 706-709.

Mat17 Mattmccutchen, Package maintenance guide. https://fedoraproject.org/wiki/Package_maintenance_guide [13.1.2017].

Mer14 Merkel D., Docker: Lightweight Linux Containers for Consistent Development and Deployment. Linux Journal, vol. 2014, issue 239 (March 2014), article no. 2.

MeW09 Meneely A., Williams L., On preparing students for distributed software development with a synchronous, collaborative development platform. Proc. 40th ACM technical symposium on Computer science education, Chattanooga, TN, USA, March 2009, pages 529-533.

MFD14 Miranda M., Ferreira R., De Souza C. R. B., Figueira Filho F., Singer L., An exploratory study of the adoption of mobile development platforms by software engineers. Proc. 1st International Conference on Mobile Software Engineering and Systems, Hyderabad, India, May 2014, pages 50-53.

Moc15 Mocevicius R., CoreOS Essentials. Packt Publishing, Birmingham, 2015.

New15 Newman S., Building microservices. O'Reilly Media, February 2015, pages 280.

nip17 nipakoo, atomic-reactor. https://github.com/nipakoo/atomic-reactor [22.2.2017].

Pro17 Project visibility level. https://docs.gitlab.com/ce/api/projects.html#project-visibility-level [18.2.2017].

pro17 projectatomic/atomic-reacor. https://github.com/projectatomic/atomic-reactor [22.2.2017].

PuB16 Punjabi R., Bajaj R., User stories to user reality: A DevOps approach for the cloud. 2016 IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT), Bengaluru, India, May 2016, pages 658-662.

Pyt2017 Python Software Foundation, setuptools 34.2.0. https://pypi.python.org/pypi/setuptools [13.2.2017].

OEC16 O'Connor R., Elger P., Clarke P. M., Exploring the Impact of Situational Context — A Case Study of a Software Development Process for a Microservices Architecture. Proc. 2016 IEEE/ACM International Conference on Software and System Processes (ICSSP), Austin, TX, USA, May 2016, pages 6-10.

ope17a openSUSE, openSUSE Build Service. https://build.opensuse.org [22.5.2017].

ope17b openshift-bot, openshift/origin. https://github.com/openshift/origin [8.2.2017].

pag17 pagure, mash. https://pagure.io/mash [28.4.2017].

pba15 pbabinca, release-engineering/koji-containerbuild. https://github.com/release-engineering/koji-containerbuild [4.12.2015].

PZA17 Pautasso C., Zimmermann O., Amundsen M., Lewis J., Josuttis N., Microservices in Practice, Part 1: Reality Check and Service Design. IEEE Software, vol. 34, issue 1 (January 2017), pages 91-98.

RaG03 Radinger W., Goeschka K. M., Agile Software Development for Component Based Software Engineering. Proc. Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Anaheim, CA, USA, October 2003, pages 300-301.

Ran17 Rancher OS, http://rancher.com/rancher-os/ [5.3.2017].

RaS16 Rana R., Staron M., First International Workshop on Emerging Trends in DevOps and Infrastructure. Proc. Scientific Workshop Proceedings of XP2016, Edinburgh, Scotland, May 2016, article no. 11.

rcp17 nipakoo, rcppkg. https://github.com/nipakoo/rcppkg [25.4.2017].

Red17a Red Hat, OpenShift. https://www.openshift.com/ [2.2.2017].

Red17b Red Hat, Creating the Kickstart File. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Installation_Guide/s1-kickstart2-file.html [28.4.2017].

Red17c Red Hat, Bodhi. https://fedoraproject.org/wiki/Bodhi [22.5.2017].

Ris17 RisingStack, Inc, Operating System Containers vs. Application Containers. https://blog.risingstack.com/operating-system-containers-vs-application-containers/ [5.3.2017].

Riv92 Rivest R., The MD5 Message-Digest Algorithm. https://tools.ietf.org/html/rfc1321 [April 1992].

RoA16 Rodin J., Aoki O., Debian New Maintainers' Guide. https://www.debian.org/doc/manuals/maint-guide/maint-guide.en.pdf. [7.12.2016].

RPA16 Rajkumar M., Pole A. K., Adige V. S., Mahanta P., DevOps culture and its impact on cloud delivery and software development. 2016 International Conference on Advances in Computing, Communication, & Automation (ICACCA) (Spring), Dehradun, India, April 2016, pages 1-6.

Rus17 RussianFedora, koji-setup. https://github.com/RussianFedora/koji-setup [22.5.2017].

SHD16 Salvadori I., Huf A., Dos Santos Mello R., Siqueira F., Publishing linked data through semantic microservices composition. Proc. 18th International Conference on Information Integration and Web-based Applications and Services, Singapore, Singapore, November 2016, pages 443-452.

SiS14 Singh I., Singh A., A survey on various component repositories with detail study of different methods of storage and extraction of components. Proc. 2014 International Conference on Advances in Electronics Computers and Communications, Yelahanka, Bangalore, India, October 2014, pages 1-6.

Spi12 Spinellis D., Package Management Systems. IEEE Software, vol. 29, issue 2 (March 2012), pages 84-86.

Suc17 Suchý M., Mock. https://github.com/rpm-software-management/mock/wiki [2.1.2017].

sys16a systemd. https://www.freedesktop.org/wiki/Software/systemd/ [10.10.2016].

sys16b system.service. https://www.freedesktop.org/software/systemd/man/systemd.service.html [27.11.2016].

SZY16 Salah T., Zemerly M. J., Yeob Yeun C., Al-Qutayri M., Al-Hammadi Y., The evolution of distributed systems towards microservices architecture. Proc. 2016 11th International Conference for Internet Technology and Secured Transactions (ICITST), Barcelona, Spain, December 2016, pages 318-325.

Tho17 ThoughtWorks, Inc., Continuous Integration. https://www.thoughtworks.com/continuous-integration [19.2.2017].

Tra09 Traylen S., RPM Building with MOCK, KOJI and MASH. http://indico.cern.ch/event/55091/attachments/980022/1392900/MockKojiMash.pdf [May 2009].

TSJ07 Tucker C., Shuffelton D., Jhala R., Lerner S., OPIUM: Optimal Package Install/Uninstall Manager. Proc. 29th International Conference on Software Engineering, Minneapolis, MN, USA, May 2007, pages 178-188.

YCB99 Ylonen T., Campbell A., Beck B., Friedl M., Provos N., De Raadt T., Song D, SSH(1). http://linuxcommand.org/man_pages/ssh1.html [21.1.2006].

Vau12 Vaupel R., Operating Systems Development Driving Forces, Critical Steps, Decision Processes. Proc. ACM SIGSOFT symposium on Industry Day, Bertinoro, Italy, June 2012, pages 15-18.

vaus17 vauss, apt-get. https://wiki.debian.org/apt-get [31.1.2017].

VKK14 Forsgren Velasquez N., Kim G., Kersten N., Humble J., 2014 State of DevOps Report. https://puppet.com/resources/whitepaper/2014-state-devops-report.

VoD13 Vouillon J., Di Cosmo R., Broken sets in software repository evolution. Proc. 35th International Conference on Software Engineering, San Francisco, California, US, May 2013, pages 412-421.

VMw06 VMware, Inc., Virtualization Overview. https://www.vmware.com/pdf/virtualization.pdf [2016].

Wei16 Weins K., New DevOps Trends: 2016 State of the Cloud Survey. http://www.rightscale.com/blog/cloud-industry-insights/new-devops-trends-2016-state-cloud-survey. [11.5.2016]

Wha17 What is Linux From Scratch? http://www.linuxfromscratch.org/lfs/ [15.2.2017].

Wil03 Willy, configure; make; make install. http://tldp.org/LDP/LG/current/smith.html [22.11.2003].

Wor15 Worley J., Customizing hosts file in Docker. http://jasonincode.com/customizing-hosts-file-in-docker/ [5.8.2015].

Zac15 Zacchiroli S., The Debsources dataset: two decades of Debian source code metadata. Proc. 12th Working Conference on Mining Software Repositories, Florence, Italy, May 2015, pages 466-469.

Appendix 1. Basic spec file for building a git maintained package

Here is a basic template for a spec file that can be used to build an RPM directly from git source code. It should be noted that our tools follow certain conventions and thus some of the fields must contain specific information. The “Source”-field should be set as seen here. This way the tarball can be located after it has been created by the scripts in our build system. We have also utilized the “URL”-field to specify the location to clone the source code from. This differs from the Fedora conventions, but we found the approach to work well when followed correctly.

```
Name:           example
Version:        0.1.1
Release:        1%{?dist}
Summary:        This package demonstrats basic spec file structure
Group:          Platform/Examples
License:        Nokia License
URL:            git@example.github/RCPPKG_spec_files/example.git
Source:         %{name}-%{version}.tar.gz
BuildRequires:  libtool
Requires:       findutils

%description
This spec file will be included in my thesis to demonstrate
the structure of a spec file that we will use for internally
controlled packages. Fedora spec files will differ in a few
fields as we will follow certain conventions to keep the spec
file as simple as possible while supporting building from git.

%package devel
Summary:        Development files for %{name}
Group:          Development/Libraries
Requires:       %{name}%{?_isa} = %{version}-%{release}

%description devel
The %{name}-devel package contains libraries and header files for
developing applications that use %{name}.

%prep
%setup

%build
./autogen.sh
%configure --disable-static
%make_build

%install
%make_install
```

```
%files
%{_libdir}/examplelibs.so*
%{_libdir}/examplelog.so*

%files devel
%{_libdir}/pkgconfig/examplelibs.pc
%{_libdir}/pkgconfig/examplelog.pc
%{_includedir}/%{name}/*.h
%{_libdir}/examplelibs.so
%{_libdir}/examplelog.so
```

Appendix 2. Script used for mirroring listed Fedora spec file repositories

With this bash script, we were able to fetch the spec file repositories into the spec file group within our internal gitlab.

```
#!/bin/bash

# Based on original script written by Petr Hruska
# that mirrors a single Fedora repository

GROUP_ID=1234               # Insert GitLab group ID of our spec file group
DESTINATION_TAG=dist-foo    # Insert tag that you want to tag the packages to

while read PACKAGE
do

if ! git clone --bare https://src.fedoraproject.org/git/rpms/$PACKAGE.git ; then
        echo "Clone repo $PACKAGE failed"
        exit 1
fi
pushd $PACKAGE.git
if ! gitlab project create --name $PACKAGE --public True --namespace-id $GROUP_ID ; then
        echo "Create $PACKAGE in gitlab failed"
        exit 1
fi
if ! git push --mirror <GITLAB_URL>:<SPEC_GROUP_NAME>/$PACKAGE.git ; then
        echo "Mirror $PACKAGE to gitlab failed"
        exit 1
fi

popd
rm -rf $PACKAGE.git
if ! git clone <GITLAB_URL>/<SPEC_GROUP_NAME>/$PACKAGE.git ; then
        echo "Clone repo from IT GIT $PACKAGE failed"
        exit 1
fi
if ! koji add-pkg --owner kojiadmin $DESTINATION_TAG $PACKAGE ; then
        echo "Creating package in koji failed"
        exit 1
fi

rm -rf $PACKAGE
done < packages.txt

exit 0
```

Appendix 3. Script used for building listed packages internally

We utilized the following script in building the required software packages with the least possible effort. Separately issuing source fetch and build commands for each package would be cumbersome. With the provided automation, we were able to leave the build running for a night and come back in the morning having RPMs ready to be installed.

```
#!/bin/bash

while read PACKAGE
do

if ! git clone <GITLAB_URL>:<SPEC_GROUP_NAME>/$PACKAGE.git ; then
        echo "Clone $PACKAGE failed"
        exit 1
fi

cd $PACKAGE
rcppkg new-sources
rcppkg build

cd ..
rm -rf $PACKAGE

done < packages.txt
```

Appendix 4. Script used for tracking down package dependencies

This is the script that we utilized to track some of the missing runtime dependencies. A lot of effort was put into being able to locate all the required packages for completing our image build and the script proved helpful in reducing the amount of time spend on the task.

```
#!/bin/bash

while read PACKAGE
do
        dnf repoquery --requires $PACKAGE >> reqs.txt
done < PACKAGES_LEFT

sort -u reqs.txt > reqs_sorted.txt
rm reqs.txt

while read REQ
do
        dnf whatprovides $REQ | grep fc25 >> provs.txt
done < reqs_sorted.txt

rm reqs_sorted.txt
sort -u provs.txt > provs_sorted.txt
rm provs.txt

while read PROV
do
        echo $PROV | cut -d '.' -f 1 | rev | cut -d '-' -f2- | rev >> MISSING_PACKAGES
done < provs_sorted.txt

rm provs_sorted.txt
sort -u MISSING_PACKAGES > MISSING_PACKAGES_SORTED
rm MISSING_PACKAGES
```