VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
INSTITUTE OF INFORMATICS
SOFTWARE ENGINEERING SUTDY PROGRAMME

Backport automation using Version Control Systems

Atgalinių perkėlimų automatizavimas naudojant versijavimo sistemas

Bachelors thesis

Author: Arnas Verpečinskas

Supervisor: Karolis Uosis, Lect.

Reviewer: Asta Slotkienė, Ph.D.

Vilnius - 2024

Santrauka

Programinės įrangos priežiūra tampa sunkesnė, kai sistema turi keletą palaikomų versijų. Klaidas bei kitas iškilusias problemas dažnais atvejais tenka taisyti vienu metu keletoje programinės įrangos versijų. Tokiais atvejais sistemų kūrėjai dažniausiai naudoja populiarų procesą, vadinamą atgaliniais perkėlimais (angl. *backporting*). Versijavimo sistemos šį procesą palengvina, tačiau kūrėjai vis tiek turi perkelti pakeitimus rankiniu būdu.

Atgalinių perkėlimų automatizavimas nėra naujas konceptas. Viešumoje yra keletas produktų gebančių įgyvendinti šią automatizaciją. Tačiau šie įrankiai turi savitų problemų, dėl kurių yra naudojama didesnis kiekis resursų bei suteikiama sunkesnė vartojimo patirtis.

Šis darbas peržvelgia atgalinių perkėlimų procesą, esamus įrankius, bei pateikia naują sprendimą atgalinių perkėlimų automatizacijos problemai, kuris pagreitina programos veikimą lyginant su kitais sprendimais bei išsprendžia kitas rastas problemas esamuose įrankiuose.

Raktiniai žodžiai: Atglainiai perkėlimai, Versijavimo sistemos, Podėliavimas, Virtualizacija

Abstract

Ever-evolving modern software can have multiple versions of the same product. This introduces some issues in the software maintenance field as developers have to maintain all of the available versions by fixing bugs or adding additional features. It is quite common that newer versions are branch-offs of the older versions. For this reason, these versions usually contain the same bugs or other issues as they are directly interconnected.

To bring the fixes to the mentioned versions of the software a *backporting* process is used. Even though Version Control Systems can be used to simplify this process it still has to be performed manually by the developers. Analysis of backporting flows in Git and Mercurial version control systems shows the possibility of complete automation of backports.

The idea of complete Backport Automation is not a completely new concept as it is visible from several solutions out in public. On the other hand, the aforementioned solutions have significant drawbacks that impact user experience and performance. As such, this paper introduces a new solution that has increased performance in specific scenarios and higher availability including a larger set of use cases.

Keywords: Backports, Automation, Version Control System, Caching, Virtualization

Contents

Glossary **4**

Introduction **5**

1 Backports **6**
1.1. Definition . . . 6
1.2. Version Control System . . . 6
1.3. Cherry-picking . . . 6
1.4. Process . . . 6
1.5. Workflow using Git . . . 7
1.6. Workflow using Mercurial . . . 9
1.7. Conclusion . . . 10

2 Automation **11**
2.1. Proposed workflow . . . 11
2.2. Possible problems . . . 12
2.3. Time saved . . . 13

3 Existing solutions **15**
3.1. LinkedIn Automated Cherry-Picking . . . 15
3.2. Sourcegraph backporting tool . . . 17
3.3. The Backport Tool . . . 18
3.4. Conclusions . . . 19

4 New solution **20**
4.1. Idea . . . 20
4.2. Virtualization and Containerization . . . 20
4.3. Database . . . 21
4.4. Message Broker . . . 24
4.5. Manager . . . 24
4.5.1. Volume caching . . . 25
4.5.2. Launching backports . . . 26
4.6. Runner . . . 27
4.7. Tracker . . . 28
4.8. API . . . 30

5 Analytics **32**
5.1. New solution vs. existing solutions . . . 32
5.2. Real world use case . . . 32

6 Results and Conclusions **34**

References **35**

A New solution's source code repository **37**

Glossary

API application programming interface, a software intermediary that allows two applications to talk to each other.

backporting the practice of applying specific changes or improvements from newer versions of software to older versions.

caching a process of storing copies of files or data in a separate location to be reused at a later date more quickly.

cherry-pick Git version control system's command to apply the changes introduced by existing commit.

cloning a version control system's command to download source code of a repository.

continuous integration a practice where developers integrate their code into a shared repository with each integration being verified by an automated build and test.

docker set of platform as a service products that use OS-level virtualization to deliver software in packages called containers.

git most widely used distributed Version Control System developed by Linus Torvalds.

GitHub a developer platform that allows developers to create, store, manage, and share their code.

GitHub Actions a continuous integration and continuous delivery platform meant to be used together with GitHub.

goroutine a lightweight thread managed by the Go runtime.

gRPC cross-platform open source high-performance remote procedure call framework.

mercurial distributed Version Control System.

merge conflict type of exception that occurs in version control systems when the system is unable to resolve differences between two points of code to combine them.

message broker software that enables applications, systems, and services to communicate with each other and exchange information.

repository software code storage location, usually managed by a version control system.

version control also known as source control, is the practice of tracking and managing changes to software code.

version control system a software tool that automates version control.

virtualization technology that is used to create virtual representations of servers, storage, networks, and other physical machines.

volume mechanism for persisting data generated by and used by Docker containers.

Introduction

It is well known that the maintenance of software is a costly procedure. According to Erlikh [Erl00] maintenance costs can reach up to and sometimes over 90% of the total cost of the software. This is due to the fact that software is a complex system that is constantly evolving. As a result, software maintenance is a process that is required to keep software in a working state. This process includes fixing bugs, adding new features, improving performance, etc.

Nowadays more and more software consists of multiple versions available to the clients. This introduces even more complexity to the maintenance process as each version requires to be handled separately. This is especially true for software used by a large user base. For example, the Android operating system has several versions [Gooa] that are actively supported and used by millions of people. This is one of many examples of such software that has several versions supported, other similar examples would be .NET [Mic], Java SE [Ora], Unity [Uni], and more.

Whenever software has multiple versions available, the versions are usually successive: meaning they are based on an older version of the same software (a branch-off). According to Cadar; Hosek in software based on successive versions, due to shared code, vulnerabilities can appear both in newer and older versions [CH12]. This means that fixes have to land in all supported versions that contain the issue. Due to this nature of the problem, developers tend to create a fix for the latest version and *backport* the changes to the remaining versions.

Backporting is a process where a part of a code is taken from a newer version of the software and is brought back to the older version. This works only for the cases where the code around the change is similar. For developers, the creation of backports can be a tedious and time-consuming process, especially if the software has numerous versions that require the same backport. The use of modern Version Control Systems helps tackle this problem by having *cherry-pick* solutions available. Even though Version Control Systems improve the workflow by reducing the work time needed it still requires manual human intervention.

Currently, there are several Backport Automation tools on the market that specialize in complete automation of backports from start to finish. The tools contain some drawbacks that might not be suitable for specific use cases. Due to this, the **main goal** of this paper is to design and develop a new Backport Automation software that solves most of the currently existing drawbacks. To accomplish this goal following tasks need to be accomplished:

1. Investigate the basic backporting theory and identify common patterns between Version Control Systems
2. Investigate the benefits of Backport Automation and identify its limitations
3. Analyze currently existing Backport Automation solutions and their drawbacks
4. Propose ways to solve the drawbacks and limitations
5. Create a technical design of a new system, selecting and comparing tools and/or frameworks to use for the implementation
6. Implement the proposed Backport Automation solution with selected tools
7. Compare the implemented solution's performance with other available tools

1 Backports

1.1. Definition

Backporting is a process where a part of a code is taken from a newer version of the software and is brought back to the older version. This process is typically done to ensure that users of the older version of the software get critical updates without the need to update to the latest version. According to Chakroborti; Schneider; Roy [CSR22] the backports are most commonly done for bug, test, document, and feature changes. Backporting requires careful consideration of dependencies and conflicts that could arise when transferring code from a newer codebase to an older one.

1.2. Version Control System

Version Control System also known as *Source control* is a software that enables tracking and managing changes to the software code. Such software has highly sophisticated tracking capabilities, enabling backtracking of actions. This software also helps developers collaborate on the same projects due to its merging capabilities. Examples of such source control systems are Git[1], Mercurial[2], Apache Subversion[3], Perforce[4] and others.

1.3. Cherry-picking

Cherry-picking is a process where a changeset is taken from one branch and applied to another branch. This process is usually done manually by developers. However, some Version Control Systems have automated cherry-picking solutions available. For example, Git has a command [Thea] called

```
git cherry-pick
```

that allows one to cherry-pick a changeset from one branch to another. Mercurial also has its own solution [Mac] for cherry-picking by using

```
hg graft
```

command.

This action is different from **merge** as merge implies that two branches are combined together. Meaning that all differences between two branches are merged. When branches are of different versions of software this means that all new features are also merged. This is not the case for cherry-picking as it only takes a single changeset and applies it to another branch. This means that only the changeset is applied and not the whole branch.

Of course, cherry-pick is not ideal tool and merge conflicts can still occur. Code can differ between branches and as a result cherry-pick will not know how to merge the changeset. In this case, the developer will have to resolve the conflicts manually.

1.4. Process

Backports work only because newer versions of the system are successors of one point in time of the older version. In other words, a newer version is usually a branch-off or a fork

[1] `https://git-scm.com/`
[2] `https://www.mercurial-scm.org/`
[3] `https://subversion.apache.org/`
[4] `https://www.perforce.com/`

of the older version that has new features developed on top. Because of that, if an issue occurs, it has a high chance of being present in both versions. As this is the case, it makes sense to make changes to the latest version first. After doing so, the changes can be brought back to the older version.

When we have a fix ready in the latest version the backport process can be started. The process would be as such:

1. Identify the changeset(s) that fixes the issue.
2. Navigate to the latest changeset of the target version.
3. Transfer the changes that fix the issue.
4. Resolve any conflicts that might occur.
5. Test the backport.
6. Upload the backport to the repository.

This is the standard way of doing a backport. It can differ between different Version Control Systems but the main principles are the same. The process is not fully automatic; Even though nowadays the Version Control Systems merge changes automatically, it still requires manual human intervention for changeset management, branch configuration, and testing. On top of that, merge conflicts can occur during the *cherry-pick* stage. This halts the process and awaits the input of the developer. As a result, the developer has to resolve the conflicts manually.

1.5. Workflow using Git

Figure 1 illustrates how the process would look like using the Git Version Control System

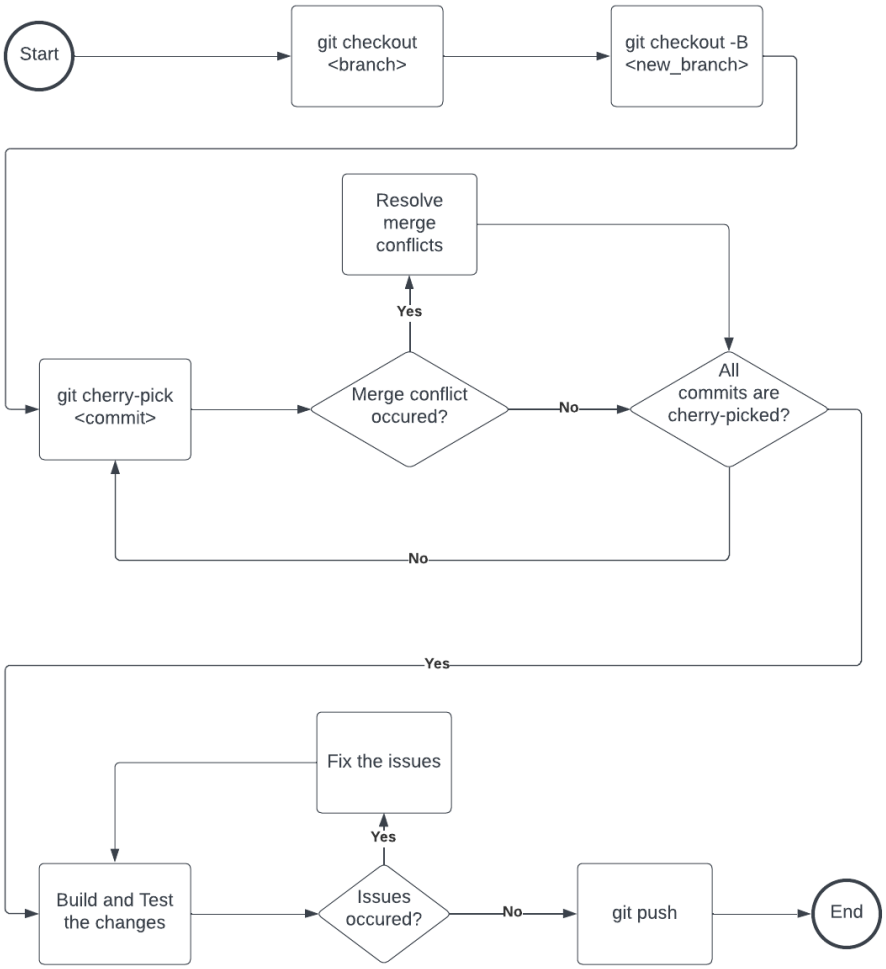

Figure 1: Backport workflow diagram using Git

The process starts out by using

```
git checkout <branch>
```

this tells Git to track the provided branch. The branch should be the one where changes need to be transferred to. At this point it is also possible to do

```
git pull
```

as it will fetch and download latest changes of the branch. In any case using

```
git checkout -b <new_branch_name>
```

will create a new branch based on last branch top commit. In case where projects don't use pull request system and changes are pushed directly to branch this step can be skipped.

Now the process of porting begins. To transfer changes from other version a list of commits that contain required changes is needed. If the changes were introduced with pull request it is possible to use single *merge commit* of the pull request instead of each separate commit. In case any merge conflicts occur they are ought to be resolved by the developer. When all commits are transferred the changes need to be built and tested. If any issues occur, they need to be resolved by developer as well. After everything is done

```
git push
```

will upload changes to the remote.

1.6. Workflow using Mercurial

Figure 2 illustrates how the process would look like using the Mercurial Version Control System

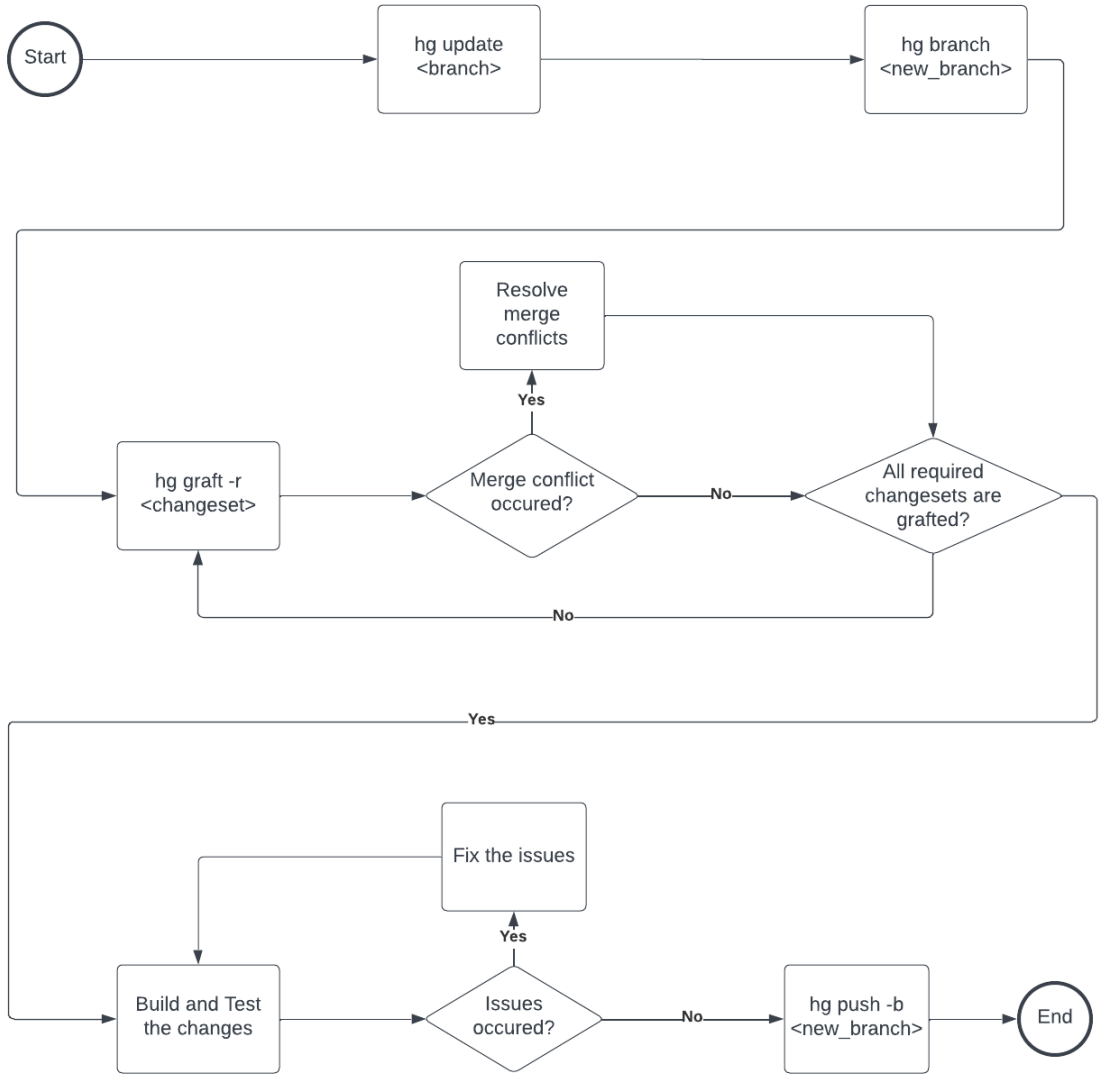

Figure 2: Backport workflow diagram using Mercurial

Backporting process using Mercurial is pretty similar to Git's. It starts out by doing

```
hg update <branch>
```

to update tracking to provided branch. Then a deviating branch is created using

```
hg branch <new_branch>
```

creating a new branch where all new changes will be stored. At this point the changes can be transferred from a newer version to an older. To do that, there is command

```
hg graft -r <changeset>
```

Essentially this command is almost identical to

```
git cherry-pick
```

as they both bring singular changes to the branch. The main difference being that they work on different basis: changesets for Mercurial and commits for Git. The grafting can also result in merge conflicts that need to be resolved manually by the developer. After doing so the changes need to be tested and any occurring issues have to be fixed. When everything is good to go, using

```
hg push -b <new_branch>
```

syncs local tracking with remote tracking and the changes are uploaded.

1.7. Conclusion

A typical backport process requires manual human intervention. Using modern Version Control Systems such as Git, Mercurial or SVN makes backporting simple as long as there are no merge conflicts. The issue becomes more complex when there is a need to do multiple backports for multiple versions per single bug fix or any other implementation. The process is quite easy to follow and perform but can become time-consuming. This especially becomes a problem when we take into consideration overworking in tech companies [Sha02].

According to Chakroborti; Schneider; Roy [CSR22] only 13% of backports had incompatible code and only 10% failed to be accepted. This data suggests that there is a possibility to automate the backporting process. On top of that, taking into account that both Git and Mercurial backporting processes have similar patterns it can be concluded that it would be possible to create a universal solution that would work similarly between different Version Control Systems.

2 Automation

Automation of backports is not necessarily a new idea [Sørb] [Sou] [Vij23]. The idea has been raised before and has been implemented under different names. There are not many public examples of such automation as development teams tend to implement this solution by creating internally used scripts.

2.1. Proposed workflow

Automation means offloading processes in Figures 1 and 2 to an application that can do that for developers automatically. The proposed workflow of automation is shown in Figure 3

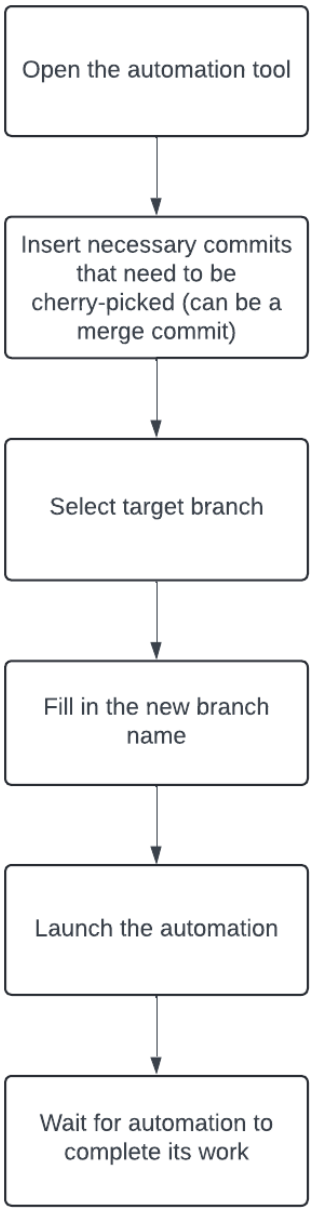

Figure 3: Proposed backporting workflow using automation

The idea behind this workflow is to have an application in which the backports could be queued up with any additional required information. After doing so the process is started

and no additional manual input is needed. When automation is done working, on success the results would be placed in the repository and the user could proceed with standard post-backport workflows.

2.2. Possible problems

Backport Automation comes with its own set of problems as the goal is to remove human interaction completely. These issues raise questions about quality and effectiveness. Here are some of the possible issues:

- **Merge conflicts**. While doing the backport cherry-picking part, there is a possibility of getting a merge conflict. As discussed before, usually the problem is left for the developer to resolve. If the backport process is automated there is no one to solve the merge conflicts and the process would fail. There are some possible solutions to this problem:

  a) **Strategy options**. Version Control Systems can have additional options to specify merging strategies. Git has several strategy options implemented into it [Theb]. Thus some cases of merge conflicts could be solved by using *theirs* or *ours* strategies. On the other hand, Mercurial does not have support for different strategy options.

  b) **Remote merge conflict solving**. There is a possibility to solve merge conflicts remotely when they occur in automation. For this to be valid the system has to be designed specifically with that ability in mind. This would most likely require automation to run on virtualized containers with the ability to be accessed remotely. In such a scenario a developer could get a notification about the failure and get the online editor to fix the conflict.

  c) **Ignoring**. Instead of trying to solve the merge conflicts automation could choose to accept the failure and decide to cancel the backport. Doing this would cause riskier backports to be done by developers manually instead of relying on automation.

- **Testing**. A successful cherry-pick without merge conflicts doesn't indicate a successful patch. Most of the backport cases are tested by developers before merging the changes to the main branch. If automation takes over then there is no one to test the changes. Having proper build and test pipelines with unit tests, integration tests, etc. can help in catching the issues but most of the time it is simply not enough. Test suites cannot detect issues in code that are not covered thus problems can slip through.

  Taking this into consideration, it would be recommended for the developer to assess the risk of the backports before using automation. In some situations, the risk might be non-existent e.g. documentation or test backports. In such instances the risk is low as the changes can be caught by the Continuous Integration pipelines or the risk might be worth taking.

  Possible solution to this problem would be an offloading of the testing to other personnel such as Quality Assurance teams. According to a survey on developer ecosystem [Jet22] conducted by JetBrains, Test / Quality Assurance Engineers have become more in-demand, and the share of projects where there is more than 1 QA per 10 developers has risen to 42%. From that, a conclusion can be drawn that in organizations there are usually Quality Assurance teams that work with manual testing that could take up the challenge of testing the backports.

2.3. Time saved

The reason for implementing such automation is the time-saving possibilities for the users (developers). According to Vijapure in their implementation of automation, they reduced time spent by 99.6%, from around 10 minutes to "a few seconds" [Vij23]. However, it is not that clear in what areas the time saving occurs. A comparison can be made between the processes in Figures 1 and 2 against Figure 3 to calculate theoretical time requirement differences for each step by the developer.

Figure 4: Time spent doing backports manually versus using automation

Task	Time spent in manual way	Time spent using automation
Backport complexity and risk evaluation	3 minutes	3 minutes
Queueing backport using automation tool	-	1 minute
Checking out to an existing branch	2 minutes	-
Fetching latest changes from the remote	2 minutes	-
Creating a new branch	30 seconds	-
Cherry-picking the changes from the original Pull Request	1 minute	-
Pushing changes to the remote server	1 minute	-
Interrupting tasks and task-switching	5 minutes	-
	Total: **14 minutes 30 seconds**	Total: **4 minutes**

Figure 4 illustrates the time differences between doing backports manually versus doing backports using the proposed automation workflow in Figure 3. For this theoretical example a large-scale repository was used (Unreal Engine [Epi]). In smaller-scale projects, the time spent doing backports manually would be significantly smaller due to the number of files and smaller differences between branches. It is important to mention that the results are affected by the specifications of the computer, where having a computer with faster disk drives and CPU could improve results.

The *Interrupting tasks and task-switching* is the main time factor affecting manual backporting results. According to Jin; Dabbish information workers tend to switch up tasks every 12 minutes [JD09]. This phenomenon is even more amplified in situations where a worker is "waiting for a slow program to respond". In one example the participant performed a series of different tasks for 13 minutes before checking back on the program. Comparing this to the commands used to manage the Version Control System the situation is quite similar as the processing of commands takes small amounts of time. This can also be considered as "loading". Because of that, it is expected that a worker will wander off to do some tasks while waiting for the Version Control System commands to finish running. That takes out a significant amount of time which is solved by automation.

Backporting can be described as a mundane task. After the initial learning curve of backports, there is not much to do except repeat the boring process. The only challenge

comes up in situations where merge conflicts occur or there are inconsistencies in testing. According to Boring; Griffith; Joe boredom at work makes employees experience lapses in attention, take longer to notice and correct errors [BGJ07]. This further raises concerns about time excess for backports.

Looking at the figures, backport automation could save up to ten minutes of developers' time. In the table, the time difference is calculated for one backport. In modern software the backports are usually required for multiple versions at once. This amplifies the boredom and task-switching problems of human nature. Taking that into consideration a conclusion can be made that automation could be beneficial for any scale of the system even though the Version Control System commands would be executed quicker.

3 Existing solutions

3.1. LinkedIn Automated Cherry-Picking

Vijapure has documented their implementation of automatic cherry-picking solution used at LinkedIn[5] in an article "How LinkedIn automates cherry-picking commits to improve developer productivity"[Vij23]. The author claims that the solution has saved around 9 hours of developers' time with 54 cherry-picked Pull Requests created from 28 Pull Requests at the launch of the article. The article doesn't show behind-the-scenes of the implementation but gives some figures that can be analyzed.

One of the figures provided is the "Manual Cherry-pick vs Automated Cherry-pick: User Actions" diagram shown in Figure 5

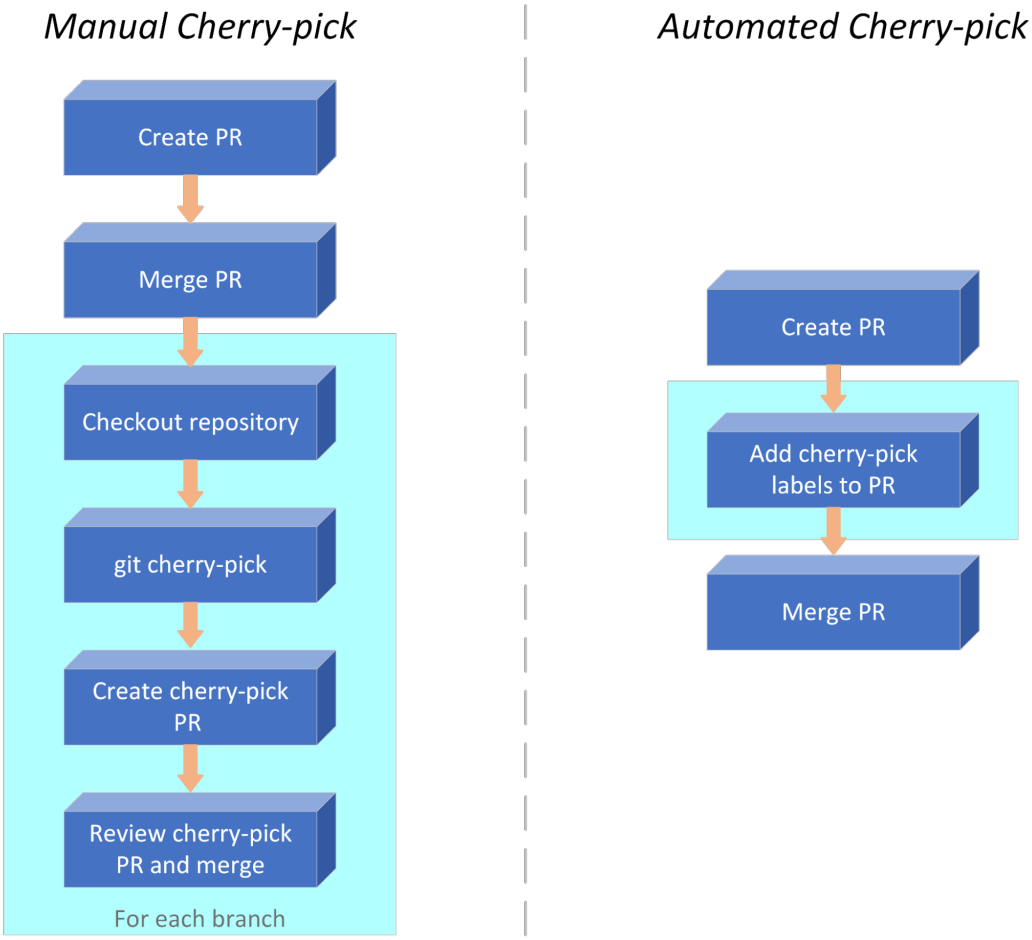

Figure 5: Manual Cherry-pick vs Automated Cherry-pick: User Actions. Taken from Vijapure article "How LinkedIn automates cherry-picking commits to improving developer productivity" [Vij23].

The figure is quite similar to the proposed workflow in Figure 3. The process that should be executed manually by a developer is being run by a separate tool that is executed by the developer. In this implementation, the tool itself is a combination of *GitHub Action* triggers and workflows. In this case, it is executed by adding a label to the Pull Request with the branch name of the version that requires a backport as such:

```
cherry-pick to {version}
```

[5] https://linkedin.com

GitHub Actions is an extension of the GitHub platform acting as a repository automation platform within the ecosystem [6]. Part of the system serves as an orchestrator, responding to triggers, such as code commits or pull requests, and initiating preset workflows. These workflows are essentially a series of commands that are categorized into distinct phases. The platform uses an internal containerization system based on Docker to manage the virtual machines, ensuring a consistent and replicable execution environment.

Using *GitHub Actions* for the automation is a great way to implement the solution. In the article, it enables the detection of changes in labels. Doing so triggers GitHub Action Workflow which boots up a Virtual Machine that runs a pre-determined script. On the other hand, this solution forces a dependency on GitHub and GitHub Actions. The logic is not isolated to a separate service. This can cause some issues if there is a need to migrate to a different cloud-based service such as GitLab[7] that has different CI/CD implementations. On top of that, it can limit the solution for expansion as it will require user input via GitHub Actions instead of having automation that could be triggered on demand by any outside factors e.g. Jira[8] comments.

It is also worth mentioning that the solution in the article limits itself to be used only with the Git Version Control System. This means that if there were some projects in the organization that was to use different source control systems (e.g. Mercurial) due to legacy or any other reasons the backporting solution would not work.

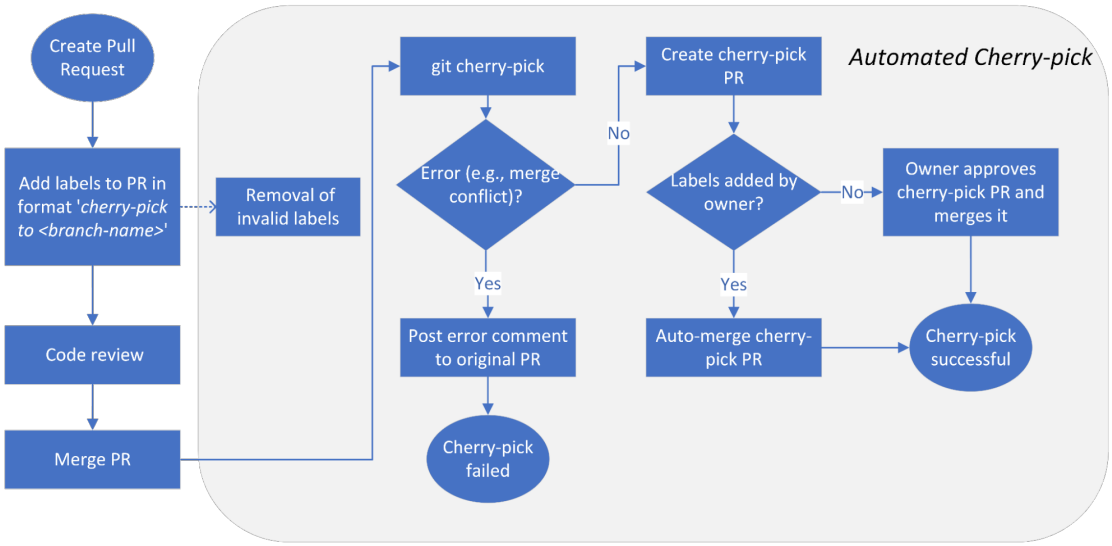

Figure 6: Automated Cherry-pick: End-to-end Workflow. Taken from Vijapure article "How LinkedIn automates cherry-picking commits to improving developer productivity" [Vij23].

Testing is a major flaw that is overlooked in the article. Figure 6 illustrates the process that is going on when automation is queued. It is mentioned that every Pull Request generated goes through the usual testing as any other Pull Request: Pull Request description check, code sanity checks, unit testing, and compatibility testing. Once this is done the Pull Request review is requested from the owner, or if the requester is the owner of the repository, it is merged automatically. Based on that and the automation end-to-end workflow assumption can be made that the workflow does not expect or require manual testing from a developer as discussed in 2.2. Possible problems. This might not necessarily be a problem

[6] `https://github.com`
[7] `https://gitlab.com`
[8] `https://www.atlassian.com/software/jira`

for this use case as testing is also dependent on the scale of the project but for large-scale projects that use trunk-based development, this is a major issue that could cause problems.

3.2. Sourcegraph backporting tool

Sourcegraph has its own implementation of a backporting automation tool [Sou]. The documentation and the source code repository indicate the design of the solution. In the same way as 3.1. LinkedIn Automated Cherry-Picking is designed, Sourcegraph's implementation is also based on GitHub Actions. This time there is access to the source code that can be analysed. Overall the system works as expected and should work well for small to medium-scale projects but some problems could occur with large-scale systems.

After investigating the source code it seems that for every backport requested, a virtual machine is started up. This virtual machine will run the backporting tool written in JavaScript. Noteworthy is part of the implementation that has this part of the code:

```
info(‘Backporting ${mergeCommitSha} from #${number}.‘);

const cloneUrl = new URL(payload.repository.clone_url);
cloneUrl.username = "x-access-token";
cloneUrl.password = token;

await exec("git", ["clone", cloneUrl.toString()]);
await exec("git", [
  "config",
  "--global",
  "user.email",
  "github-actions[bot]@users.noreply.github.com",
]);
await exec("git", ["config", "--global", "user.name", "github-actions[bot]"]);
```

This suggests that every time a virtual machine is booted up a repository will be cloned. In some scenarios with a project of high size and history (e.g. Unreal Engine [Epi]), the cloning can take up to an hour or even above. This would mean that for each backport, a large amount of resources would be used up. To solve this problem, it would be possible to run scheduled jobs that do the cloning and save the virtual machine state into a reusable image.

Another insight for this implementation is that the error handling is quite limited. Looking at Figure 6 it seems that this problem might also affect the LinkedIn Automated Cherry-Picking solution. Sourcegraph's backporting executes this code while trying to do a backport:

```
await git("switch", base);
await git("switch", "--create", head);
try {
  await git("cherry-pick", "-x", commitSha);
} catch (error: unknown) {
  await git("cherry-pick", "--abort");
  throw error;
}

await git("push", "--set-upstream", "origin", head);
```

From this, it can be seen that errors are being caught and handled in the cherry-pick stage. The same way is shown in Figure 6 diagram. This would mean that any other issues caught in different stages as clone, fetch, and push are overlooked and could result in unknown failures. Such issues definitely can occur, for example, if the branch with the identical name that is provided to the label already exists, the push command would result in a failure. In this instance, it would not be caught.

3.3. The Backport Tool

Søren Louv-Jansen has created The Backport Tool [Sørb]. It is a Command Line Interface (CLI) tool written in JavaScript using Node.js. The repository has around 240 GitHub stars and around 90 thousand weekly downloads from Node Package Manager [Sørc]. This tool serves as backport automation for manual use locally. To set it up user has to install it either globally

```
npm install -g backport
```

or project-wise

```
npm install backport
```

For it to work user has to create a project config at the root of the repository. The config looks like this (example taken from The Backporting Tool documentation):

```
// .backportrc.json
{
  // Required
  "repoOwner": "elastic",
  "repoName": "kibana",

  // the branches available to backport to
  "targetBranchChoices": ["main", "6.3", "6.2", "6.1", "6.0"],

  // Optional: automatically merge backport PR
  "autoMerge": true,
  "autoMergeMethod": "squash",

  // Optional: Automatically detect which branches a pull request should be backpor
  to based on the pull request labels.
  // In this case, adding the label "auto-backport-to-production" will backport
  the PR to the "production" branch
  "branchLabelMapping": {
    "^auto-backport-to-(.+)$": "$1"
  }
}
```

besides that user has to fill in a global config located at %user%/.backport/config.json file:

```
// ~/.backport/config.json
{
  "accessToken": "ghp_very_secret"
}
```

This tool has a user-friendly CLI and can cherry-pick, create branches, and even Pull Requests. When the tool is used for the first time to make a backport it creates a second copy of the repository. This is made to not intervene with the original repository.

The author has also created a GitHub Action to be used with this tool [Søra]. This enables the tool to work in a virtual machine when specific conditions in the workflow are met. As the implementation is hosted on GitHub Actions it will in the same way clone the source code of the repository taking additional processing time.

As the tool is made for local use it has some drawbacks regarding work in multiple repositories. Since repositories can be stored in different servers the authentication to them is different. As such the access token can also be different between repositories. In such scenarios, the tool will not work as it would have access to the repositories specified by the access token in the global configuration file. This problem is somewhat solved when using the GitHub Action variant of the tool as it requires an access token to be specified in each repository that the tool is used in.

As mentioned before this tool requires adding a configuration file at the root for each repository. The configuration adds additional unnecessary complexity and additional dependency.

3.4. Conclusions

Three backport automation tools were analyzed. Comparing all the sources these conclusions were made:

1. All analyzed tools were limited to Git version control system meaning there is no solution for other version control systems such as Mercurial

2. All analyzed tools are based on GitHub Actions as their UI and backend meaning organizations that use other source platforms such as GitLab can not utilize the tools

3. Downloading of some repositories can take a significant amount of time. For code repositories such as Unreal Engine [Epi] *git clone* command can take around 30 minutes. Doing this task on each backport takes up processing power, network traffic, and additional wait time.

4. Error handling is limited: there is no good way to find the cause of a problem without going through the logs. This increases the knowledge requirement to use the tool effectively for non-technical users

5. Limited extensibility. With current implementations, it would be difficult to extend the functionality to support other wanted functionality, for example: sending notifications after backport completion, closing tickets on task tracking software

6. Complex configuration. The Backport Tool has shown to require additional configuration in the repository and outside it. This makes setup more complicated to users

4 New solution

4.1. Idea

To solve all drawbacks found in 3.4. Conclusions there has to be a change in the idea itself. Instead of depending on currently available source management software, this tool has to become independent. Meaning that the solution has to become a standalone software. To successfully accomplish this, open-source containerization software Docker [Doc] could be used.

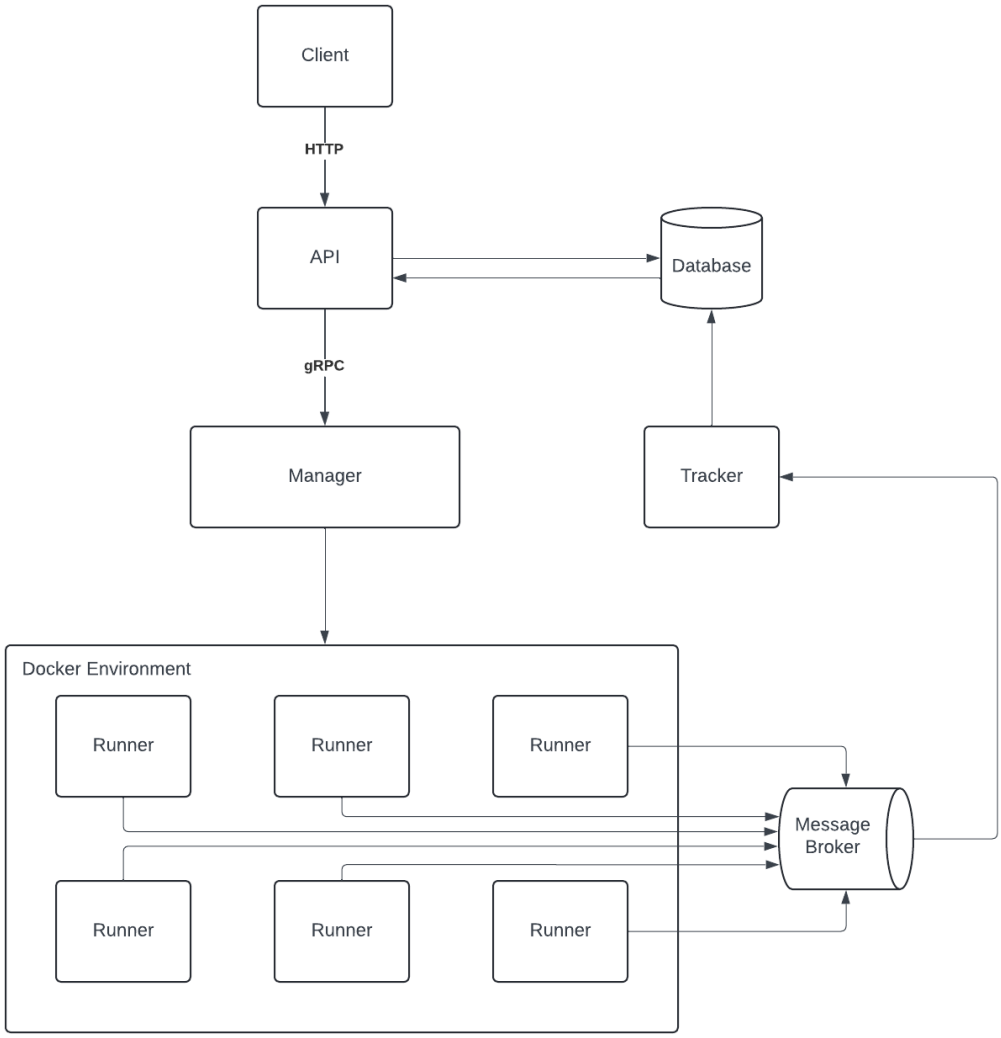

Figure 7: New Backport Automation solution's high-level overview diagram

Figure 7 illustrates how the new solution would work. The application is split into three main components: API, Manager, Tracker, and Runner. API and Tracker connect to the same database to update entries. Runners connect to the Message Broker server as publishers and the Tracker connects to Message Broker server as a subscriber.

4.2. Virtualization and Containerization

One of the key points behind the backport automation is virtualization. To have a functioning system without downtime backports need to be created on demand. It would be possible to have a solution that has a queue and creates backports on one machine

but its downtime would be substantial as previous backports would need to finish before continuing to the next one. As such it makes more sense to have a solution that can scale automatically. This is where virtualziation comes in. Virtualization enables the creation of a virtual environment for the task with a specified amount of resources.

There is a distinct difference between virtualization and containerization [YGR19]. According to Yadav; Garg; Ritika Containerization can be described as a lightweight virtualization technique as it uses fewer resources. This is due to them sharing the same kernel where as Virtual Machines have another layer of isolation. When comparing virtual machines to containers author mentions that containers are more suited to provide high availability and scalability.

For Backport Automation containerization would enable to create separate lightweight environments for each backport, removing the containers after the job completion. This would enable software to be scalable on both sides: resources are not used when no backports are ran and resources are distributed properly when there are multiple requests at once.

Docker is the most popular containerization platform that is used by millions of people [Mat21]. According to the article on Docker statistics of 2021, Docker has had 396 billion Pulls on Docker Hub. A single pull is a single download of a published pre-built container image in Docker Hub (image registry). For this reason, the new solution will be using Docker for its containers.

4.3. Database

The core of the application is the database. All the required information is stored there. MongoDB database was chosen for this application. MongoDB is the most popular NoSQL database technology [Mon]. MongoDB is a document-oriented database program. The document-oriented database works very well in the Backport Automation scenario as everything revolves around one type of document: Backport. As such, there is no need for traditional relational databases.

In the Backport Automation context the database stores two types of documents: repositories and backports.

```
{
  "_id": {
    "$oid": "6633f2af1b1af490fae04c5f"
  },
  "name": "4rgetlahm.event-tracker",
  "clone_url": "https://4rgetlahm:ghp_F8JhFLHLDdT1kjLAWbtPe1uGA8FDTv3bvpGa
  @github.com/4rgetlahm/event-tracker.git",
  "volume": {
    "name": "4rgetlahm.event-tracker.repo",
    "status": "ready",
    "last_updated": {
      "$date": "2024-05-02T20:08:16.638Z"
    }
  },
  "version_control_system": "git"
  "date_created": {
      "$date": "2024-05-02T20:08:10.214Z"
  }
}
```


Figure 8: Example repository entry in MongoDB database

Repository stores 5 properties:

1. _id - Stores unique object ID
2. name - Unique name of repository that is set by user
3. clone_url - HTTPS URL with optional login credentials that will be used for source code download
4. volume - Object that contains critical information about cached volume (read 4.5.1. Volume caching)
   (a) name - Generated or generating volume name
   (b) status - Volume status (one of not_initialized, creating, ready)
   (c) last_updated - Date of the last update to this volume
5. version_control_system - Defines what version control system should be used for backport processing
6. date_created - Date of repository creation

```
{
  "_id": {
    "$oid": "6633f2bf1b1af490fae04c60"
  },
  "author": "4rgetlahm@gmail.com",
  "commits": [
    "0c9ae3c1d791513e6da1977caaed8fb6cf995f26"
  ],
  "repository": {
    "_id": {
      "$oid": "6633f2af1b1af490fae04c5f"
    },
    "name": "4rgetlahm.event-tracker",
    "clone_url": "https://4rgetlahm:ghp_F8JhFLHLDdT1kjLAWbtPe1uGA8FDTv3bvpGa
    @github.com/4rgetlahm/event-tracker.git",
    "volume": {
      "name": "4rgetlahm.event-tracker.repo",
      "status": "ready",
      "last_updated": {
        "$date": "2024-05-02T20:08:16.638Z"
      }
    },
    "version_control_system": "git"
  },
  "target_branch_name": "main",
  "new_branch_name": "test-backport-automation6",
  "events": [
    {
      "action": "VirtualMachinePreparing",
      "content": null,
      "date_created": {
        "$date": "2024-05-02T20:08:31.512Z"
      }
    },
    ...
    {
      "action": "VirtualMachineExited",
      "content": {
        "container_id": "9acecae4217de6caaef472f73fa410430a634f2eece92d
        0f86ff5c925361eee3"
      },
      "date_created": {
        "$date": "2024-05-02T20:08:35.872Z"
      }
    }
  ],
  "date_created": {
    "$date": "2024-05-02T20:08:31.509Z"
  }
}
```


Figure 9: Example backport entry in MongoDB database

Backport stores 8 properties:

1. _id - Stores unique object ID
2. author - Email or name of the user who requested the backport
3. commits - Array of string type elements containing SHA256 hashes on what commits to cherry-pick
4. repository - Repository object with its information on the time of creation
5. target_branch_name - Branch name that will be used to base backport on
6. new_branch_name - Name of a branch that will be created after cherry-picking
7. events - Array of events that contain information about the Backport process
   (a) action - Event name
   (b) content - Optional additional parameters relevant to the event
   (c) date_created - Date of event creation
8. date_created - Date of backport creation

4.4. Message Broker

A message broker is software that enables applications, systems and services to communicate with each other and exchange information [IBM]. It is different from other ways of communication as it serves as an intermediary between other applications. One of the key points of a message broker is its ability to retain messages, replay messages, and deliver messages to multiple applications at once. These points make it a crucial element of the application because of its ability to store status updates of backports in case of errors or downtime on the other components of the system.

In this implementation, Google's Pub/Sub was chosen as a message broker due to it being a cloud service that is already hosted and does not require additional setup. Other solutions such as Apache Kafka, RabbitMQ, or Redis Streams require to be set up locally to be used. The message broker has one topic with one subscriber (Tracker) named

`backport.runner.updates`

with the goal to enable Runners to publish status updates and Tracker to read them.

4.5. Manager

The Manager is a core component of the system. It is responsible for two major actions: cache preparation and backport environment creation. Go programming language was selected to create this component.

Go is a statically typed, compiled high-level language that has faster performance than interpreted languages such as Python or JavaScript. Golang has become an appealing choice for DevOps engineers because of its performance and high consideration for concurrency. Besides that it has necessary packages for this system such as Docker Client.

The application uses gRPC to receive two types of requests: launch a backport and prepare a cache. gRPC is a high-performance, open-source framework designed for efficient communication between services using a binary protocol and HTTP/2 [H24]. With the use of Protocol Buffers, it creates a solid and performant way to communicate between applications. It is especially useful in communication between several internal programs (in this context Manager and API) due to already having a pre-defined request and response structure.

4.5.1. Volume caching

The first main task of the Manager component is the ability to prepare the reusable environment for backports. As investigated before, one of the main problems of Backport Automation is downloading the source code repository each time. It does not cause problems when the repository is smaller and can be cloned in less than 30 seconds but for repositories such as Mozilla Firefox [Moz], Linux Kernel [Lin], Google's Android Open Source Project [Goob], Unreal Engine [Epi] this is not the case. Downloading of the source code can take up to or even over an hour. It also uses unnecessary processing power and a large amount of network traffic.

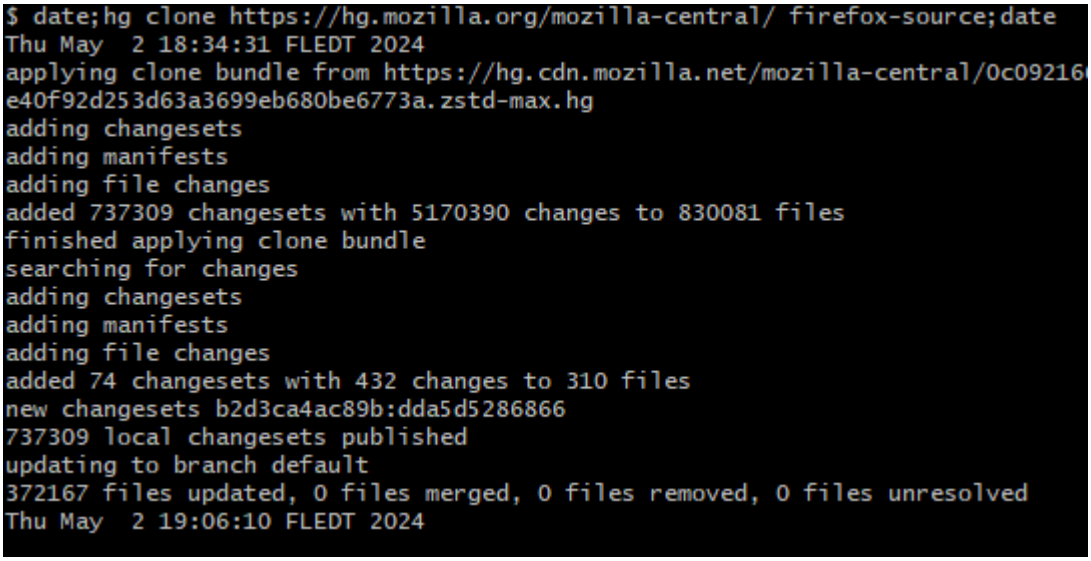

Figure 10: Mozilla Firefox source download time takes 31 minutes 39 seconds

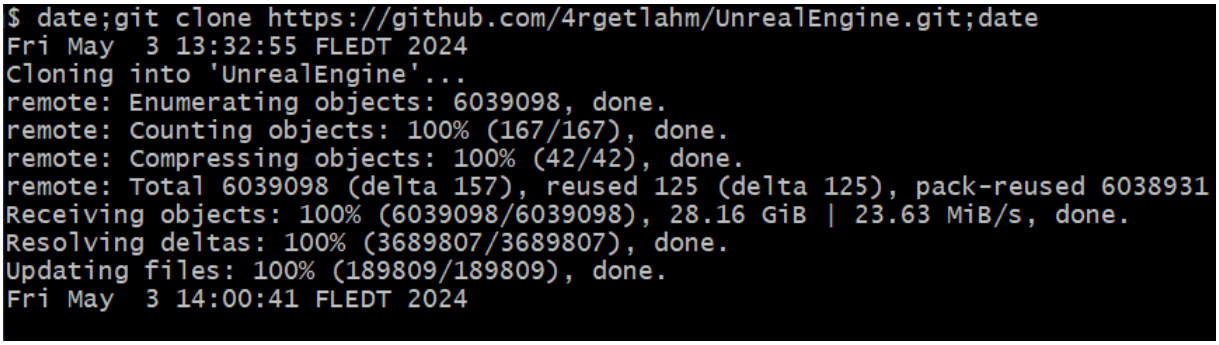

Figure 11: Unreal Engine source download time takes 27 minutes and 46 seconds

Docker has the ability to attach persistent storage to its containers. As such it is possible to create one persistent volume as the single source of truth that can get periodic updates (e.g. weekly) for each repository. With this volume, it would be possible to create on-demand copies for each backport that would get deleted after use. On-demand copies are needed for the Version Control System not to overlap with other running backports, corrupting the main volume.

With this in mind, Manager service waits for gRPC requests with 4 parameters:

1. volume_name - Name of a new volume that will be created
2. vcs - Version Control System that will be used for cloning

3. clone_url - HTTPS URL that will be used for cloning

4. (optional) overwrite - Setting this flag to true will remove previous volume with the same name if it exists

After receiving such a request a new container is started that has a newly created volume attached. This container runs a custom Docker image that uses a custom Python script to download repositories depending on the environmental values provided. If everything is successful, a new volume is created and is marked so in the database.

When a new backport request is received, the first action is to check if the volume exists. If it does, a new Docker container is launched that creates a temporary volume and copies all the repository files from the main volume. Depending on the size and file amount in the repository it can be fast or can take some time. Copying Unreal Engine source code to a new image takes around 4 minutes but it is still 10 times improvement compared to cloning the source code for 40 minutes. After the backport job is completed the temporary volume is removed.

4.5.2. Launching backports

The second task of the Manager service is launching the backports. It is done by receiving a gRPC call with 6 mandatory parameters:

1. Reference number - a unique 12-byte value generated by MongoDB that is used to identify the backport

2. Volume - Name of generated volume that will be used to make a repository copy

3. VCS - Name of used Version Control System

4. New Branch Name - Branch that will be created in case of successful backport

5. Target Branch Name - Name of a branch that will have the backport based on

6. Commits - A list of hashes that will be applied to the target branch

At this point, as mentioned in 4.5.1. Volume caching, a new container is created that copies files from *Volume* to a new temporary volume. After doing so a launch of the Runner script can begin. To achieve that there was a custom Docker image created.

```
FROM python:3.12.2-slim

# Clone the backports repository
RUN apt-get update && apt-get install -y git && apt-get install -y mercurial
RUN git clone https://github.com/4rgetlahm/backports.git

COPY . .

# Install requirements
WORKDIR /backports/runner
RUN pip install -r requirements.txt

# Set the entrypoint
ENTRYPOINT ["python", "backport_runner.py"]
```

Figure 12: Dockerfile of backport-runner image

Figure 12 shows the contents of *backport-runner* Dockerfile. It takes pre-existing Python image, installs necessary version control packages, and clones the repository containing 4.6. Runner. Then it installs the necessary Python packages and sets entry point to be the main script.

This creates a single image that can be run on demand with a volume attached for any backport. When the container is created, variables from initial gRPC call are passed down to the container as environment variables. In such way, the Runner script will be able to understand what version control system it will need to use, what branch to create, what commits to cherry-pick, etc.

The container automatically shuts down after the script is finished running. The manager detects its exit and follows the clean-up procedure. For clean-up it removes the container that ran the backport and after that volume with the repository copy is removed.

4.6. Runner

Runner is the smallest unit in the system. This is a script that interacts with the Version Control System. It could be equated to minified version of The Backport Tool [Sørb]. Essentially, the runner will follow the backporting process described in Figure 1 with the addition of status reporting to a remote server. To work correctly the script takes information from environment variables that are passed down by the Manager:

1. REFERENCE - ID of backport that will be used to send notifications to the remote server
2. NEW_BRANCH_NAME - Name of a branch that will be generated in case of success
3. TARGET_BRANCH_NAME - Name of the branch that the new branch will be based upon
4. COMMITS - Comma-separated list of SHA256 commits used for cherry-picking
5. REPORTER_CONFIG - Base64 encoded JSON containing Google Pub/Sub project, topic, and credentials
6. SOURCE_PATH - Absolute path of the repository location
7. VCS - Version Control System that will be used

At the start of each stage of the process, the runner will send a message to the Pub/Sub server. The message would be in JSON format and would look like this:

```
{
  "reference": "663011745054fcf63b6d9536",
  "stage": "cherry-pick",
  "status": "start",
  "payload": {
      "commit": "9108f5ea8392992221863559d5f3c89b7c8fef76"
  }
}
```

After the running necessary VCS command in the process (git cherry-pick as in the example) another message is sent informing about the status of the command. The message looks like this:

```
  {
    "reference": "663011745054fcf63b6d9536",
    "stage": "cherry-pick",
    "status": "failure",
    "payload": {
        "commit": "9108f5ea8392992221863559d5f3c89b7c8fef76",
        "error": "Cmd('git') failed due to: exit code(1)
  cmdline: git cherry-pick -m 1 9108f5ea8392992221863559d5f3c89b7c8fef76
  stdout: 'On branch test-backport-automation
You are currently cherry-picking commit 9108f5ea8392.
  (all conflicts fixed: run "git cherry-pick --continue")
  (use "git cherry-pick --skip" to skip this patch)
  (use "git cherry-pick --abort" to cancel the cherry-pick operation)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
modified:   Engine/Source/ThirdParty/jemalloc/build/build.sh
modified:   Engine/Source/ThirdParty/libPNG/libPNG-1.6.37/ltmain.sh

no changes added to commit (use "git add" and/or "git commit -a")'
  stderr: 'The previous cherry-pick is now empty, possibly due to conflict resoluti
If you wish to commit it anyway, use:

    git commit --allow-empty

Otherwise, please use 'git cherry-pick --skip''"
    }
  }
```

To develop the Runner component Python programming language was chosen for its rich collection of modules and libraries such as base64, JSON, GitPython, Google PubSub. Besides that Python usually comes pre-installed on operating systems making it an better pick for ease of setup.

4.7. Tracker

Tracker is a service that runs asynchronously inside Manager as a *goroutine*. This component connects to the database and Message Broker (Google Pub/Sub) as a subscriber and listens for any messages in

```
backport.runner.updates
```

topic. It converts the messages reported by the Runner into predefined events that are used in the rest of the program. The complete list of events that a backport can have is shown in Figure 13.

```
ActionVirtualMachinePreparing = "VirtualMachinePreparing"

ActionVolumeCreateStart   = "VolumeCreateStart"
ActionVolumeCreateSuccess = "VolumeCreated"
ActionVolumeCreateFailure = "VolumeCreateFailure"

ActionVirtualMachineError   = "VirtualMachineError"
ActionVirtualMachineCreated = "VirtualMachineCreated"
ActionVirtualMachineExited  = "VirtualMachineExited"

ActionRunnerStarted = "RunnerStarted"

ActionGitFetchStart   = "GitFetchStart"
ActionGitFetchSuccess = "GitFetchSuccess"
ActionGitFetchFailure = "GitFetchFailure"

ActionGitCheckoutStart   = "GitCheckoutSuccess"
ActionGitCheckoutSuccess = "GitCheckoutSuccess"
ActionGitCheckoutFailure = "GitCheckoutFailure"

ActionGitCheckoutNewBranchStart   = "GitCheckoutNewBranchStart"
ActionGitCheckoutNewBranchSuccess = "GitCheckoutNewBranchSuccess"
ActionGitCheckoutNewBranchFailure = "GitCheckoutNewBranchFailure"

ActionGitPullStart   = "GitPullStart"
ActionGitPullSuccess = "GitPullSuccess"
ActionGitPullFailure = "GitPullFailure"

ActionGitCherryPickStart   = "GitCherryPickStart"
ActionGitCherryPickSuccess = "GitCherryPickSuccess"
ActionGitCherryPickFailure = "GitCherryPickFailure"

ActionGitPushStart   = "GitPushStart"
ActionGitPushSuccess = "GitPushSuccess"
ActionGitPushFailure = "GitPushFailure"

ActionRunnerExited = "RunnerExited"
```

Figure 13: Possible Backport events

For each received message the tracker follows these steps:

1. Deserialize JSON message into a structure
   (a) On deserialization failure the message is acknowledged but abandoned (does not proceed the flow)
2. Check the status of the structure
3. Check the stage of the structure

4. Get the payload from the status message
5. Find event from Figure 13 based on status and the stage
6. Add a new event to a backport event array in the database

In this way all created backports will have a history of events that occurred with any additional data provided. Consequentially, all relevant backport information will be tracked inside the database without any need for logs.

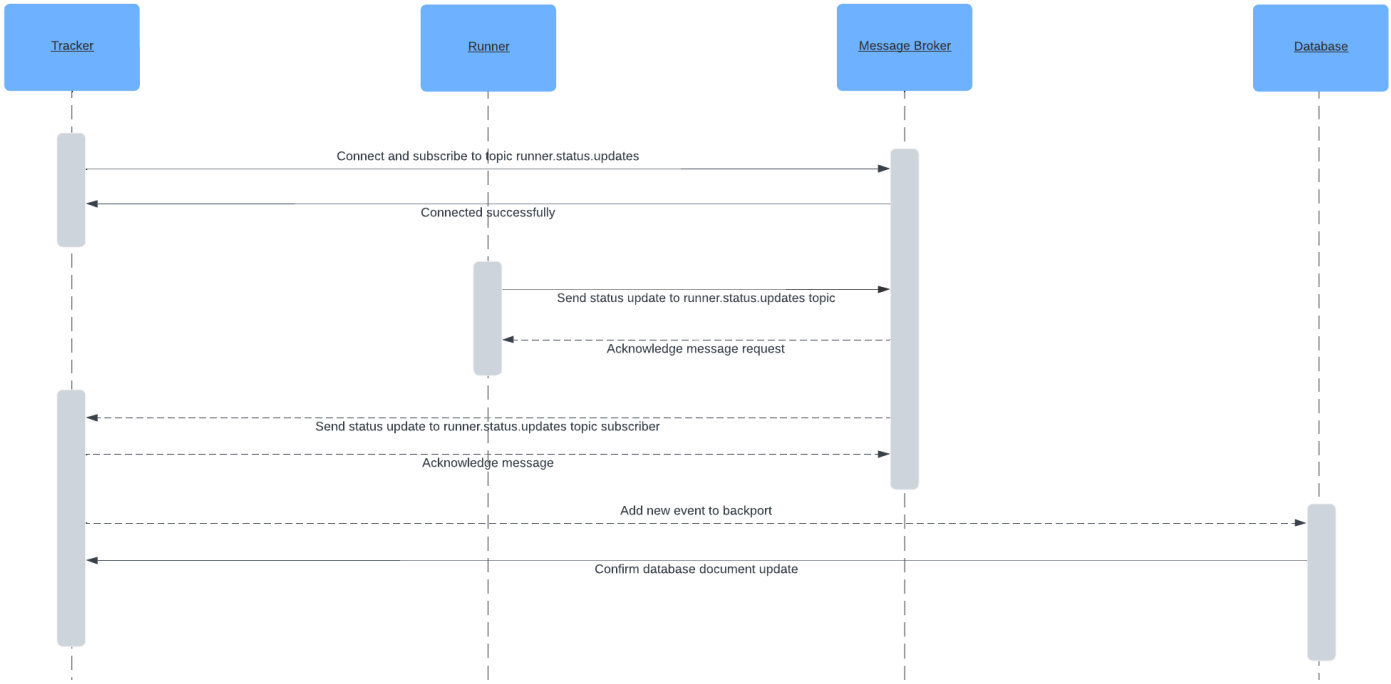

Figure 14: Sequence diagram showing backport status update process

With the Tracker workflow in mind, Figure 14 illustrates how the pipeline of state transfer from runner to database works:

1. Tracker connects to Message Broker
2. Runner sends a message to Message Broker
3. Message Broker sends the message to Tracker
4. Tracker processes the message
5. Tracker updates the database with new status update

4.8. API

API in this context is an HTTP web server that handles the creation and retrieval of the backports to and from a database. Along with handling backports, it also handles requests to add new repositories to the registry so the backports in the repository can be created later.

The API has 5 routes in total:

1. 2 for repositories

(a) GET /v1/repositories - Retrieves all repositories and returns them in JSON format

(b) POST /v1/repository - Creates a new repository

2. 3 for backports

(a) GET /v1/backports/:from/:to - Fetches *from* - *to* amount of backports from database skipping first *from* elements. Used for pagination on client. Returns results in JSON format

(b) GET /v1/backport/:id - Fetches all information about backport with id being *id*. Returns result in JSON format

(c) POST /v1/backport - Creates a new backport

In the same way as the Manager component, this component was also developed with the Go programming language additionally using the Gin framework. This was meant to ease up the development process by keeping common types together. Besides that, as Go is a compiled language its performance is substantially better than its alternatives such as the Express.js framework in Node.js [Cho23].

Besides receiving data from the database in GET-type requests. API serves two main purposes exposed as POST requests:

`POST /v1/repository`

Takes in 3 parameters via JSON body:

1. version_control_system - Specifies what VCS the repository is using
2. clone_url - HTTPS URL with optional authentication information where to clone the repository from
3. name - Unique name of the repository that will be used to queue backports

After the request is received with these 3 parameters, API sends a request via gRPC to the Manager to start the initialization of a new volume (see 4.5.1. Volume caching).

`POST /v1/backport`

Takes in 5 parameters via JSON body:

1. author - Email or name of the user who requested the backport
2. commits - Array of string type elements containing SHA256 hashes on what commits to cherry-pick
3. repositoryName - Registered repository name that contains information about the volume
4. targetBranchName - Name of a branch that backport will be based on
5. newBranchName - Name of a branch that will be created after cherry-picking the commits

After the request is received with these 5 parameters, API creates a new entry in the database with all the required data (see Figure 9) and a request is sent via gRPC to the Manager to start a new backport (see 4.5.2. Launching backports)

5 Analytics

5.1. New solution vs. existing solutions

The new solution can be compared only to Sourcegraph's solution due to Vijapure solution not being open to the public.

Comparing the New solution to the solution by Sourcegraph (3.2. Sourcegraph backporting tool) we get a significant performance increase on repositories that take a long time to clone. This is due to the Sourcegraph tool cloning the repository for each backport. But ability to use new solution's caching system requires time for initial set up.

Figure 15: New solution vs Sourcegraph Backporting tool backporting a commit to Unreal Engine source

Task	Time spent using new solution	Time spent using Sourcegraph backporting tool
Source code download (cloning)	4 minutes 20 seconds	27 minutes 46 seconds
Backporting process	28 seconds	28 seconds
	Total: **4 minutes 48 seconds**	Total: **28 minutes 14 seconds**

Results show a significant decrease in total execution time per backport due to caching ability. On top of that the new solution fixes all other crucial points:

1. Tool is not limited to a single version control system. Support for new version control systems can be added by updating the Runner module with instructions on how to create a backport for the specified Version Control System
2. Solution uses Docker for its virtualization backend. As Docker is open-source this makes the tool independent of any paid dependencies
3. Source download time problem is solved by the introduction of volume caching
4. Each backport in the tool contains an event history showing detailed information about each state change. This enables easier error detection.
5. New solution can be easily extended to support additional functionality for pre-processing or post-processing of backports
6. Repository configuration is simple, requiring a user to specify their used version control system, repository download URL, and custom name for identification

5.2. Real world use case

A similar solution to the described one in section 4. New solution was developed and used in Unity Technologies since July 11, 2023, for internal backports. The difference between the solution introduced in this thesis and the one used at the organization is the access to internal tooling and virtualization software. Unity maintains 3 separate versions of its' software [Uni]. As such backport automation took a crucial part in the organization. In 9 months since the tool's release it has received 3592 backport requests.

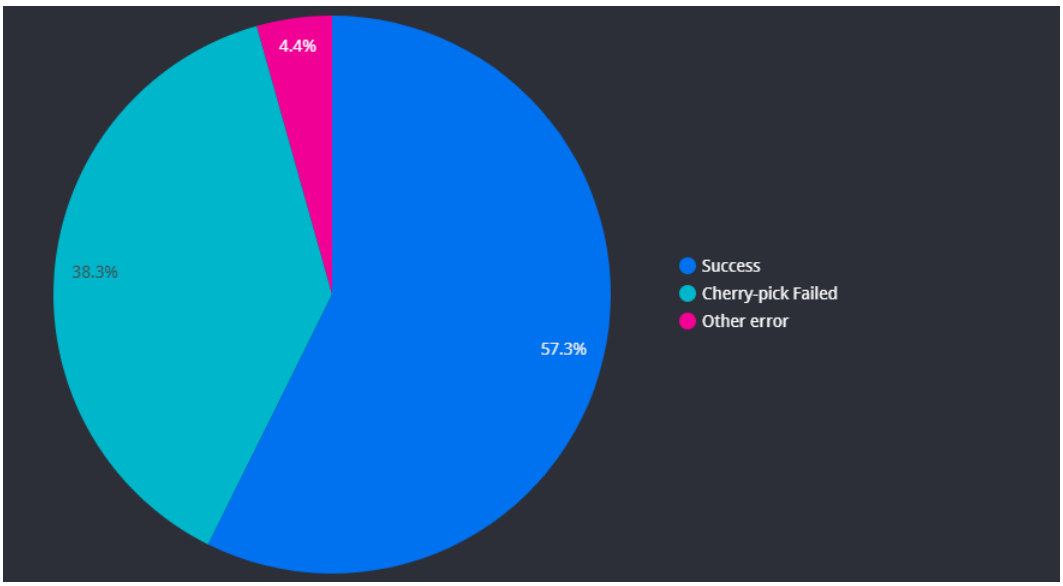

Figure 16: Distribution of backport request final states

Looking at total statistics from Figure 16 it can be seen that from 3592 requests 2416 were successful, 1614 resulted in merge conflicts, and the rest 184 backports failed due to other errors. Other errors imply failures at other stages: virtual machine failed to start, fetch command failed, pull command failed, push command failed, etc.

As the tool was also expanded to create Pull Requests after backport completion there are also statistics on how many backport requests resulted in merged Pull Requests. Of the 2416 successful backport requests 2063 of those have been merged into their respective branches. Taking this information and time-saving estimations given in Figure 4 it can be concluded that Backport Automation in 9 months of runtime has saved

$$2063 * 10,5 = 21661,5$$

minutes of developers' time. That is over 361 hours saved.

There was an internal survey conducted regarding Backport Automation and its performance. One of the questions was "Has Backport Automation saved you time? If so, how much?". The results showed that from the user's perspective backport automation saved from 20 to 30 minutes per backport. Taking the middle ground of 25 minutes a new figure comes up

$$2063 * 25 = 51575$$

This new figure equates to 859,58 hours of developer's work time.

As the tool was mostly used to create backports for the Unity Engine repository that is large in disk size and cloning would take around 30 minutes, the tool was able to utilize the volume caching mechanism. The average runtime of the backport from request until the finish of the backport was around 5 minutes. This figure is the same and is not affected by the state of the backport (success or failure). This very well colludes with tests done on the Unreal Engine repository where the average run time of a backport was also around 5 minutes (see 5.1. New solution vs. existing solutions)

6 Results and Conclusions

In this study, the main objective was to create a new solution that would solve existing drawbacks from other solutions. This objective was successfully achieved by solving 6 major drawbacks found in other existing solutions.

In this study following milestones were achieved:

1. Explored the fundamental theory of Backports on several Version Control Systems
2. Examined the theoretical Backport Automation process, offering solutions to its primary issues, and compared the advantages of automation with manual workflows
3. Assessed publicly existing implementations of Backport Automation, identified their drawbacks
4. Generated an idea for a new Backport Automation solution that would solve existing solution drawbacks
5. Implemented the new Backport Automation solution that solves existing solution drawbacks
6. Compared the performance of new solution against existing solutions

From the achieved results conclusions were drawn:

1. Backporting processes exhibit similarities across various Version Control Systems
2. Manual resolution of backports can be a time-consuming and tedious task, impacting long-term product quality
3. Automation of backporting can result in substantial time savings, from 10 to 30 minutes per individual backport
4. Usage and need for already existing tools shows demand for Backport Automation as The Backport Tool has over 90 thousand weekly downloads
5. Currently existing automation implementations have inherent drawbacks that could be mitigated through alternative approaches to the problem
6. Newly developed solution solves major drawbacks found in existing solutions and shows better performance
7. A new data source from Unity Technologies shows demand for Backport Automation technology as well as a large positive impact on productivity

Given the results and conclusions, it is clear that backport automation is an area that has not yet fully matured and has the potential to be researched even further. Implementing a new solution gives additional standing ground for its expansion but can be further improved.

References

[BGJ07] BORING, R. L.; GRIFFITH, C. D.; JOE, J. C. The Measure of human error: Direct and indirect performance shaping factors. In: *2007 IEEE 8th Human Factors and Power Plants and HPRCT 13th Annual Meeting*. 2007, pp. 170–176. Available from DOI: 10.1109/HFPP.2007.4413201.

[CH12] CADAR, C.; HOSEK, P. Multi-version software updates. In: *2012 4th International Workshop on Hot Topics in Software Upgrades (HotSWUp)*. 2012, pp. 36–40. Available from DOI: 10.1109/HotSWUp.2012.6226615.

[Cho23] CHOUBEY, M. URL shortener service benchmarking: Node (Express) vs Go (Gin). *Medium.com* [https://medium.com/deno-the-complete-reference/url-shortener-service-benchmarking-node-express-vs-go-gin-f30519685386]. 2023.

[CSR22] CHAKROBORTI, D.; SCHNEIDER, K. A.; ROY, C. K. Backports: Change Types, Challenges and Strategies. In: *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. Virtual Event: Association for Computing Machinery, 2022, pp. 636–647. ICPC '22. ISBN 9781450392983. Available from DOI: 10.1145/3524610.3527920.

[Doc] DOCKER. *Docker web page* [https://www.docker.com/]. [No date].

[Epi] EPIC GAMES. *Unreal Engine Source Code Repository* [https://github.com/EpicGames/UnrealEngine]. [No date].

[Erl00] ERLIKH, L. Leveraging legacy system dollars for e-business. *IT Professional*. 2000, volume 2, number 3, pp. 17–23. Available from DOI: 10.1109/6294.846201.

[Gooa] GOOGLE LLC. *Android Developers Page* [https://developer.android.com/about/versions]. [No date].

[Goob] GOOGLE LLC. *Android Open Source Project* [https://android.googlesource.com/]. [No date].

[H24] H, J. gRPC vs. REST: Key Similarities and Differences. *DreamFactory blog* [https://blog.dreamfactory.com/grpc-vs-rest-how-does-grpc-compare-with-traditional-rest-apis]. 2024.

[IBM] IBM. *What is a message broker?* [https://www.ibm.com/topics/message-brokers]. [No date].

[JD09] JIN, J.; DABBISH, L. A. Self-Interruption on the Computer: A Typology of Discretionary Task Interleaving. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Boston, MA, USA: Association for Computing Machinery, 2009, pp. 1799–1808. CHI '09. ISBN 9781605582467. Available from DOI: 10.1145/1518701.1518979.

[Jet22] JETBRAINS. The State of Developer Ecosystem 2022. *JetBrains Developer Ecosystem*. 2022.

[Lin] LINUS TORVALDS AND LINUX FOUNDATION. *Linux Kernel* [https://github.com/torvalds/linux]. [No date].

[Mac] MACKALL, O. *Mercurial source code management system* [https://www.mercurial-scm.org/doc/hg.1.html]. [No date].

[Mat21] MATT CARTER. Docker Index Shows Momentum in Developer Community Activity. *Docker Blog*. 2021.

[Mic] MICROSOFT CORPORATION. *.NET Support Policy* [https://dotnet.microsoft.com/en-us/platform/support/policy]. [No date].

[Mon] MONGODB. *Most Popular NoSQL Database* [https://www.mongodb.com/resources/basics/databases/nosql-explained/most-popular-nosql-database]. [No date].

[Moz] MOZILLA CORPORATION. *Firefox Source Tree Documentation* [https://firefox-source-docs.mozilla.org/]. [No date].

[Ora] ORACLE CORPORATION. *Oracle Java SE Support Roadmap* [https://www.oracle.com/java/technologies/java-se-support-roadmap.html]. [No date].

[Sha02] SHARONE, O. Engineering consent: Overwork and anxiety at a high-tech firm. *Berkeley Collection of Working and Occasional Papers*. 2002, p. 52.

[Søra] SØREN LOUV-JANSEN. *Backport Github Action repository* [https://github.com/sorenlouv/backport-github-action]. [No date].

[Sørb] SØREN LOUV-JANSEN. *The Backport Tool* [https://github.com/sorenlouv/backport]. [No date].

[Sørc] SØREN LOUV-JANSEN. *The Backport Tool npm registry* [https://www.npmjs.com/package/backport]. [No date].

[Sou] SOURCEGRAPH. *Sourcegraph backporting tool* [https://handbook.sourcegraph.com/departments/engineering/dev/tools/backport/]. [No date].

[Thea] THE GIT DEVELOPMENT COMMUNITY. *git-cherry-pick Documentation* [https://git-scm.com/docs/git-cherry-pick]. [No date].

[Theb] THE GIT DEVELOPMENT COMMUNITY. *Merge Strategies Documentation* [https://git-scm.com/docs/merge-strategies]. [No date].

[Uni] UNITY SOFTWARE INC. *Long Term Support Releases* [https://unity.com/releases/editor/qa/lts-releases]. [No date].

[Vij23] VIJAPURE, A. How LinkedIn automates cherry-picking commits to improve developer productivity. *LinkedIn Engineering Blog*. 2023.

[YGR19] YADAV, A. K.; GARG, M. L.; RITIKA. Docker Containers Versus Virtual Machine-Based Virtualization. In: ABRAHAM, A.; DUTTA, P.; MANDAL, J. K.; BHATTACHARYA, A.; DUTTA, S. (editors). *Emerging Technologies in Data Mining and Information Security*. Singapore: Springer Singapore, 2019, pp. 141–150. ISBN 978-981-13-1501-5.

Appendix A New solution's source code repository

The source code of the implemented solution mentioned in the paper can be found in the GitHub source code repository: `https://github.com/4rgetlahm/backports`