Institute for Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Change tracking and observability for complex software development

Nistha Bhawsinka

Course of Study:	Computer Science
Examiner:	Jun.-Prof. Dr. Benjamin Uekermann
Supervisor:	Ishaan Desai, Christoph Schlameuß (IBM)
Commenced:	June 1, 2023
Completed:	September 25, 2023

Abstract

The complexity involved in the wide range of software products that are developed around the globe varies in many aspects. It is often easier to manage small projects because of less development tools involved, but for larger projects it can get overwhelming fairly quickly. Having an option to view all the important parts of a development process at one place can help in such complex scenarios. The objective of this study is to make this process simpler by introducing an observability solution for the development teams. The work revolves around two use cases, one from IBM having more than 250 repositories managed by a small group of developers. And the other one is preCICE, which is an open-source project with around 50 repositories. Both the use cases have their respective needs which are recorded by surveying the developers. The solution developed is intended to be generic so that it can be used in other projects in the future. This accounts to the pressing need of creating a reusable solution.

Major components of the proposed solution include a Grafana dashboard with custom scripts written in Python. It is a client server architecture, shipped as docker images for portability purposes. One of the key aspects of this study was to keep the developers informed and understand their needs. This helped while designing questions for all the surveys that were conducted with them as subjects. Results of those surveys acted as a knowledge base while planning the solution. The work also contributes in streamlining the software development process in complex projects by reducing the mental effort and development time of the developers. There is an added advantage of better collaboration with such an overview of the project. It will allow the new developers to get familiarized with the organizational pulse and make contributions to the source code in a more informed manner. The results showcase how projects with different level of complexity and perspectives can benefit from such a solution.

Kurzfassung

Die Komplexität der zahlreichen Softwareprodukte, die weltweit entwickelt werden, ist in vielerlei Hinsicht unterschiedlich. Kleine Projekte sind oft einfacher zu verwalten, weil weniger Entwicklungswerkzeuge beteiligt sind, aber bei größeren Projekten kann es schnell unübersichtlich werden. Eine Möglichkeit, alle wichtigen Teile eines Entwicklungsprozesses an einem Ort abzubilden, kann in solchen komplexen Szenarien helfen. Das Ziel dieser Studie ist es, diesen Prozess durch die Einführung einer Beobachtungslösung für die Entwicklungsteams zu vereinfachen. Die Arbeit dreht sich um zwei Anwendungsfälle, einen von IBM mit mehr als 250 Repositories, die von einer kleinen Gruppe von Entwicklern verwaltet werden.Der andere ist preCICE, ein Open-Source-Projekt mit etwa 50 Repositories. Beide Anwendungsfälle haben ihre jeweiligen Bedürfnisse, die durch Befragung der Entwickler erfasst werden. Die entwickelte Lösung soll generisch sein, so dass sie in Zukunft auch in anderen Projekten eingesetzt werden kann. Dies trägt der dringenden Notwendigkeit Rechnung, eine wiederverwendbare Lösung zu schaffen.

Zu den Hauptkomponenten der vorgeschlagenen Lösung gehört ein Grafana-Dashboard mit benutzerdefinierten, in Python geschriebenen Skripten. Es handelt sich um eine Client-Server-Architektur, die aus Gründen der Portabilität als Docker-Images ausgeliefert wird. Einer der wichtigsten Aspekte dieser Studie war es, die Entwickler zu informieren und ihre Bedürfnisse zu verstehen. Dies half bei der Gestaltung der Fragen für alle Umfragen, die mit ihnen als Probanden durchgeführt wurden. Die Ergebnisse dieser Umfragen dienten als Wissensgrundlage für die Planung der Lösung. Die Arbeit trägt auch dazu bei, den Softwareentwicklungsprozess in komplexen Projekten zu rationalisieren, indem der mentale Aufwand und die Entwicklungszeit der Entwickler reduziert werden. Ein zusätzlicher Vorteil ist die bessere Zusammenarbeit mit einem solchen Überblick über das Projekt. Neue Entwickler können sich so mit dem organisatorischen Pulsschlag vertraut machen und fundiertere Beiträge zum Quellcode leisten. Die Ergebnisse zeigen, wie Projekte mit unterschiedlichem Komplexitätsgrad und unterschiedlichen Perspektiven von einer solchen Lösung profitieren können.

Contents

List of Figures

Acronyms

API Application Programming Interface. 25

APM Application Performance Monitoring. 24

HPVS Hyper Protect Virtual Server. 11

JSON JavaScript Object Notation. 25

REST Representational State Transfer. 25

SCA Souftware Composition Analysis. 2

SCM Source Code Management. 1

SSH Secure Shell. 45

URL Uniform Resource Locator. 29

VCS Version Control System. 3

VPC Virtual Private Cloud. 11

YAML YAML ain't markup language. 30

1 Introduction

Software development is one of the core sectors responsible for the growth and development of the era. There are many sectors inside of software development which can be of interest for different minds. This field of study is evolving at a fast pace and has a steep learning curve for new-comers. Because of the increased demand and reliance in everyday lives, it is important that the developed systems meet the requirements and are efficient in their operation. Efficiency should be ensured not only during the performance stage but also during the development phase of the solution. Traditional methodologies of software development were time consuming and error-prone due to more repetitions and manual touch points. Such approaches are no longer feasible in modern distributed architectures with faster release cycles [7]. Hence, tools like source code managements systems and automatic build and test systems are in place, to support developers plan their work and even collaborate with other developers for better synchronization of work.

With great tools comes great responsibilities, which means that even though the tools ease up the development efforts, they have some overheads of their own. Developers need to be careful about following guidelines while using Source Code Management (SCM)s to avoid over-writing modifications or breaking some other functionality while repairing their bugs. While viewing the repositories of a small application, it may be legible to make sense about their current state. But as the size of the organization grows, it becomes more and more complex to get an idea about what is going on in the application as a whole. Such a bird-eye view will help the developers to quickly overview the activities which might need their attention or inform the concerned peer to take action. In a longer run, it will result in reducing the downtime of the application and decreased stress for developers which was earlier caused by multiple navigational challenges between various development tools such as Jenkins, GitHub etc. This bird-eye view including various aspects of software development to understand the state of the system is called observability.

This work revolves around the concept of observability and explores various observability solutions present in the market. There are two use cases (softwares) under consideration, one hosted on an enterprise infrastructure and another on an open-source infrastructure. The goal is to investigate suitable tools for both the use cases and try out the selected tools for getting feedback from the development team about their experience. With projects which are bigger in size, oftentimes there are a lot of inter-dependencies. Even for code repositories, there is a graph connecting the dependant repositories with one another. Due to the size and complexity factor, it is not possible for the developer to back-track and find root causes of failures in such a scenario. This is where change tracking becomes crucial.

Change tracking in distributed systems is called distributed tracing. An effort was made to find existing tools for making the dependency graph available to the developers for their GitHub repositories. If such a tool already exists, then its benefits were evaluated using feedback from the users. Otherwise, a custom solution was proposed to be used by the developers. With the size of the project, such a solution should be open for enhancements in the future. To make the study more

realistic and closer to the expectations of the users, there were surveys circulated to the development for their inputs. Starting from an initial survey to understand the problem statement to a final survey for recording the feedback on this study's results.

In the first chapter, there is a detailed explanation about the state of the arts related to the central topic of this research. Since version control systems are the base of modern software development cycle, there is a brief introduction about the three types of VCs present around us. The most popular one being Git, also used in both the use cases of this study[1]. Moving on to a brief introduction about Source Code Management systems and some of the features offered by them. Code repositories are an integral part of any SCM, they are the logical locations where developers save their code base for easy collaboration.

Furthermore, there is an explanation about how the concept of repositories are extended to create modular repositories. There are two major kinds of repositories mentioned, monolithic and multiple repositories. Along with the overview about both the variants of repositories, the pros and cons are also discussed in this chapter. Additionally, there are many other tools and processes which are used to develop, test, deploy and manage a software application. One such process is Souftware Composition Analysis (SCA), it introduces automation in dependency management of the software. An application can have many external and internal dependencies which have to be up to date in order to keep the application free from vulnerabilities and security risks. SCAs are responsible for taking care of these aspects. The next section explains the role of automation tools during the software development life cycle. Jenkins is such an automation tool used for continuous integration and delivery. There is also a detailed explanation of terms like observability and dashboards since the results of this study revolve around them.

The two use cases are explained in Chapter 3. It is highlighting the current software development process along with the tools used within the development teams. Both the use cases differ in a lot many ways which poses an interesting angle to this research. There is a comparison in the end describing the usability aspects of the proposed solution. After being familiar with the two use cases, it will be natural to set a baseline for the research. This baseline is achieved by circulating surveys with the prospective users of the system and the survey results are also presented in Chapter 4. After establishing the baseline, Chapter 5 outlines the approach taken by keeping in mind various available solutions in the software market. After studying about these solutions, a comparison is made to check the feasibility on the lines of requirements gathered during the initial survey. Furthermore, this approach is then concretized in Chapter 6 for both the use cases. This chapter focuses on detailed explanation of every module that is a part of the solution along with a few figures supporting it visually. In every research it is particularly important to get feedback from the users. Chapter 7 presents the feedback which was collected via surveys after the developers were made aware of the proposed solutions. Chapter 8 then concludes along with some final remarks and outlook which can be useful in continuing the research.

[1] Jason-Pat. *Top 10 Best Version Control Systems for 2023*. [Online; accessed 3 September 2023]. URL: https://www.accuwebhosting.com/blog/best-version-control-systems/.

2 Background of Tools and Technologies

2.1 Version Control in Software Development Life Cycle

Software development is a vigorous field in today's marketplace, almost everything around us has a software component involved whether it is in the production stage or in the procuring stage. But in the bigger picture, behind every successful software product lies a huge set of processes such as test, maintenance, migration, evaluation, refactor etc. This brings along the necessity of employing multiple developers, devOps engineers, software architects, testers, security developers and many more. It is highly likely that this group of individuals do not dwell in a single geographic location because of the cost factor. Since they all are working on the same product, there is some level of coordination or synchronization needed between them. For instance, in order to avoid repeated effort, the tester needs to have the latest version of code developed by the developer. Same goes for devOps engineers and even for developers who need to keep the codebase synchronized with each other, to avoid overwriting or deletion of someone else's work. Because of such use cases, the idea of *version control* evolved greatly over time. There are majorly three variants of version control systems that were used to manage the code repositories or files in general [3]:

1. **Local Version Control System**: In the olden days, the people involved in a project would copy files around directories for tracking different versions. Often the naming of such directories is given such that they indicate timestamp in a readable fashion. However, it is very evident that one wrong copy paste command can lead to a huge mistake and there is a lot of manual effort needed which further increases the chances of human error. Hence, local version control systems came into existence. They are like a local database which keeps track of files that undergoes any change. They would do so by storing the patch files which indicate only the changes that have happened since the last version. Software such as Quilt is used to combine multiple patches into one patch, making it easier for developers to test and evaluate the performance of the modified piece of code. Local Version Control System (VCS) comes with a major drawback of non-collaborative environment. Developers need to manually share the code base with fellow developers or choose not to share at all as it is meant for the local file system.

2. **Centralized Version Control System**: Since, the above approach lacks the feature where multiple developers could contribute to a single project or even a single file sometimes. This gap was filled by the concept of centralized version control systems where developers could check out projects from a single server to their local repositories and work independent of the other developers involved in the project. The major bottleneck of such a solution is that, if the central server fails for some reason there is no backup server to continue serving the users. This may even cause data loss if there was some uncoordinated work going on the server. In bigger projects, such a single point of failure would not be acceptable. Furthermore, if two developers want to work on the same piece of code or in similar scope, file locking allows

them to lock the file while they are modifying it and release the lock once they are done. This way, they can avoid over-writing each other's piece of work and hence reduce development effort. However, waiting for someone to release the lock may lead to increased development time.

3. **Distributed Version Control System**: The issue of single point of failure in centralized VCS was solved by the introduction of distributed VCS. In here, the entire working repositories are mirrored when a client checks out the file. Except for pushing the changes, all the other activities with the repository can be performed offline which offers the benefit of working independently from the network. In case of a server failure, any of the clients can be available to copy the checked-out files onto the server. Git is one of such VCS widely accepted in the industry[1]. There are also some downsides of using a distributed VCS, such as it is not always legible to find out who made the most recent change because of a long trail of commits. Furthermore, care has to be taken while setting up repositories so that the security rules are up to date, and all the branches are protected from vulnerabilities.

Software Development process is often carried out using a life cycle approach with some predefined stages of development [6]. These stages include requirement gathering and analysis, planning, development, testing, evaluation, deployment etc. like in Figure 2.1. It is a common practice to use distributed version control system in all the stages of a software life cycle and not just the development part. There are several other software tools that help the development team to create an ecosystem wherein they can code, test, package and deploy efficiently. Throughout the development process the roles played by an individual can also change over time. For instance, a developer can test their code and take the role of a tester or even deploy the software to act as a devOps engineer. All this is a part of the software development life cycle[2].

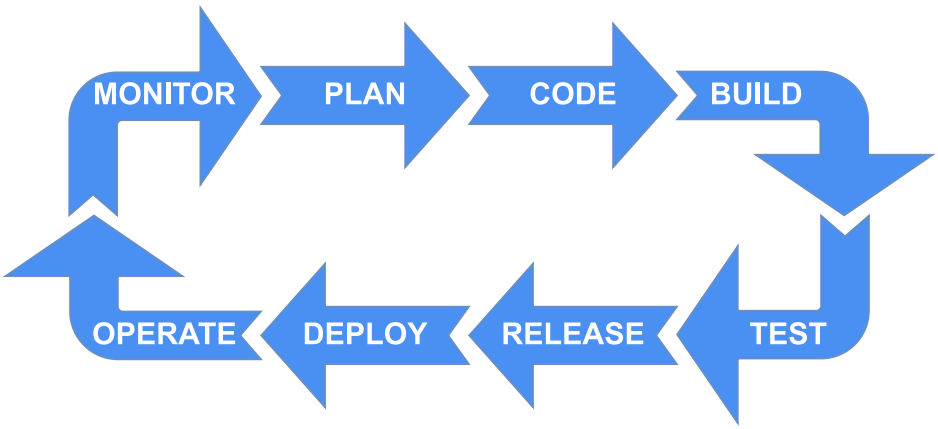


Figure 2.1: Sample Development Pipeline

[1] Jason-Pat. *Top 10 Best Version Control Systems for 2023*. [Online; accessed 3 September 2023]. URL: https://www.accuwebhosting.com/blog/best-version-control-systems/.

[2] Mabl. *Development Pipeline*. [Online; accessed 3 September 2023]. URL: https://www.mabl.com/hubfs/CICDBlog.png.

2.2 Source Code Management

Source code management is a practice useful in tracking modifications within a repository. For instance, if developer *A* is working on a functionality and they edit a configuration file which may be common to certain modules of the project and developer *B* is also working on another functionality with same configuration file. If the first developer checks in their code first, there is a high chance that developer *B* will overwrite the changes made by developer *A*. This problem was addressed by the centralized VCS through file locking mechanism but with distributed VCS, there is a need to track conflicts while merging the edits made by multiple users of the repository. Furthermore, as the complexity of any software increases, the necessity of such a mechanism also increases by many folds. SCM provides this functionality and hence reduces the development effort and delivery time of a project. One could only imagine how it was able to be managed during the local version control era. SCM provides a safety mechanism against accidental overwrites or loss of data by informing the involved people about the conflicts and giving them a chance to resolve such conflicts with full visibility of current change logs.

SCM also facilitates the project with bookkeeping functionality. It can be used to take a look back and gather insights based on the growth of the project or milestones which can be beneficial for future planning and optimizations. Compared to the previous approaches, SCM has also increased the speed with which releases are being performed. The reason being, decreased overhead of communication needed between individual developers. Due to its numerous advantages source code management has become an indispensable part of any software development project[3]. On major component of any SCM is code repository, the next sub-section describes the importance of a code repository which will be helpful to further understand the best practices of source code management systems.

2.3 Code Repository

A repository is nothing but a storage location for software components, often referred to as repo. These components are the development assets of a project, it can be code, configurations, documentations, artifacts etc. Repositories are used to structure a project's codebase along with its documentation and help developers in collaborations. And version control systems are used to track the changes within a repository made by the collaborators. It allows developers to create branches and work independent of the other developers. While VCS is not a mandatory part of a repository, it is an essential one for ensuring maintainability and reliability of the repository. Although repository and VCS are an integral part of any SCM, VCS and SCM are often used interchangeably. Branching is an especially important aspect of VCS which allows contributors to start their own stream of work based out of their desired state of repository. They can develop, test and document their code independent of the main line of development and they have an option to merge their work to the part branch once their peers or the maintainers review and approve their request. There are several best practices or guidelines which should be followed for efficient collaboration without messing up with the main line of development.

[3] Atlassian. *Source Code Management*. [Online; accessed 14 August 2023]. URL: https://www.atlassian.com/git/tutorials/source-code-management.

Git is the most widely accepted industry standard for managing code infrastructure. It is a distributed version control system originally developed around 2005 by Linux Torvalds for managing the Linux kernel project. It was inspired by BitKeeper, a versioning tool used by the Linux kernel project before the birth of Git [3]. Git organizes its repositories in the form of a huge tree with several branches and it can also be visualized using git command line. And to use git in a more convenient way, there are cloud-based hosting services such as GitHub, GitLab, BitBucket etc. which provide an endpoint for using git as a VCS. The focus of this work is on GitHub as a VCS from here onward. GitHub allows users to have three types of accounts namely: personal, organization and enterprise accounts. The enterprise accounts can be either hosted on GitHub servers or they can also be self-hosted. The difference between all these account types lies in the complexity of projects ranging from personal projects to huge company software products. For security, there are possibilities to assign roles to the participants of any project. These access roles describe the level of authority one has while making changes to the repository, there are restrictions for the roles with less privileges. These roles are read, triage, write, maintain and admin. Admin is usually the owner of the repository who has the right to add other contributors and decide upon their privileges. Each developer can create a replica of the main repository by using the feature called branches. There is a main or master branch which is used for synchronizing the work of all the developers and releasing the final software to the users. One common problem while using branches is merge conflict. Multiple developers can work on the same file or sometimes even the same lines and suppose one of them merges their work with the main branch. Now when the second developer tries to merge their work to the main, they will first have to resolve the conflicts which may have occurred to avoid overwriting other developer's work. Also, it is a good practice to follow naming conventions while working with branches so that the purpose of every branch is clear from its name.

GitHub is not limited to be used by only developers, even project managers (PMs) can use it for planning the stages of development. It offers a section called 'Projects' wherein PMs can organize the tasks for developers and label them to categories such as in-progress, to-do, backlog, completed etc. There is automation available to first create an issue and the link it with a task on the project board (Kanban-like board) for better bookkeeping. One can also assign milestones to include timeliness in the planning phase of the project. Issues can be part of an 'epic' which is the central topic around which the issues revolve. In an agile software development process, such tools come in particularly useful to the team and helps them to stay flexible with the development methods.

2.4 Modular Repositories

Modular programming is a design technique which helps teams modularize their code repositories[4]. It not only gives a cleaner structure to the code but also helps the developers to keep track of features individually, reuse, and debug them. The piece of code which may be used by multiple other features is extracted and kept as an independent module. It can either be in the form of a sub-module within an existing repository or kept outside as an independent repository of its own.

[4] K. Chandrashekar. *Modular Repositories*. [Online; accessed 2 May 2023]. URL: https://www.boulderes.com/resource-library/modular-repositories-with-git.

The complexity of modular code repositories increases if there are multiple programming languages used within a project. There are two broad categories in which repositories can be organized namely, monolithic repositories and multiple repositories.

Mono Repo stores all the code from a business at one location. They can grow exceptionally with the size of the business or projects using such a technique. It offers much visibility to the contributors in terms of viewing the hierarchy of dependencies between them. This can also help the collaborators to better support each other and create a more standardized code culture. However, there are some disadvantages of following this approach. Without proper documentation, monolithic code repositories can easily overwhelm the new developers with their enormous size. They also introduce lags in integrated development environment (IDE) resulting in longer build time. Monolithic repositories have risks, if there is one wrong commit it can affect the main body of the software due to the tight coupling. Such a scenario will be hard to trace back which can also cause monetary loss to the profitable applications. To make the modules more loosely coupled, multi repos can be used.

Multi Repo store the code across various locations, it can be split between different repositories or projects as per the team organization. Such repos are easier to manage because they are smaller. It has been a regular practice to adapt to the multi-repo approach as the project grows over time. Multi repos have clear boundary of ownership and lets the developers focus on their individual goals. Unlike monolithic repo, if there is a wrong commit which can potentially break the code, it is easier to detect in multi repo approach. Also, the damage caused by the commit will be contained only in the repository where the commit was made. This also means that the versioning of components becomes easier. With small repositories it is easier to onboard new developers and safer to let them carry on development without breaking the main body of the code. Yet as the software product grows bigger, multi repos may get complicated and introduce tremendous repetition in the boilerplate code. The overhead of making minor changes across multiple repositories can be very high. Standardization is often tough for such projects because different developers may practice different coding guidelines resulting in varied non-standard development standards across repositories. However, the most difficult problem is to manage dependencies between the repos, as there can be multiple levels of repositories linked with one another and yet kept isolated.

It can be concluded that creating modules is a good practice, but it is often a hassle to organize them efficiently. The decision of whether to use a mono repo or multi repo approach is taken based on the size of the project and potentially many other factors involved in a complex software development process. The important thing is to have the ability to scale up as per the demand of the software and realize when it is time to adapt to a different approach. Hybrid approaches are also becoming popular because they provide a good trade-off between the pros and cons of both the techniques. Although it is good to make the decision based on the current project status with future projections instead of simply following some hard guidelines from the books.

2.5 Software Composition Analysis

There are often many libraries and packages involved as imports while working with multiple code repositories. Many of such imports are subject to a license, and developers must be mindful to comply with their usage permissions. Software composition analysis (SCA) is an automated process

which identifies open-source software in a codebase[5]. It ensures that the code follows license compliance and if not, it will identify it as a vulnerability which the development team can take under consideration [9]. SCA tools looks inside the source code, artifacts, images etc. and identifies open-source packages. These packages are then compiled into a file called BOM (Bill of materials) which in simple terms is a file with a list of packages with their licensing information. BOM files are checked against a database to identify if there are any deprecated packages or vulnerabilities that need addressing.

SCA also analyzes overall code quality of the repository to ensure quality and reliability in the development process. If this is done manually, it will be much less efficient and may also result in errors which can cause vulnerable packages to go undetected within a project. One such widely accepted SCA is Mend, it also offers static application security testing (SAST) to further tighten the security of any proprietary or open-source code[6]. It can also be integrated with IDEs to enable immediate detection of vulnerabilities while the code is being created. This can reduce the release time significantly and allow development of more secure applications. Mend also has a feature called Renovate, which is a dependency management tool. It scans through the code repository to identify dependencies that can be updated with newer versions and automatically creates pull requests with the updated dependency. This is not only applicable to open-source packages but also to packages within the project.

2.6 Automation Server

For complex software development processes, automation can prove to be beneficial as it can be inferred from the previous section. In developing a software there is much more to it than just writing a piece of code. It is also important to make sure that the code is executable, distributable, and reliable for the users. Jenkins is such an automation tool which helps the development team to build, test and deploy their code systematically[7]. It supports the concept of continuous integration and delivery, which in turn allows faster release cycles. There can be trigger mechanisms setup which tells Jenkins when to start its course of action. For instance, when a developer pushes their changes, there will be a web-hook that will trigger build command on the latest change made by the developer. It allows the developer to monitor the status of their change, if anything is broken it can be caught at an early stage with such build pipelines. Similarly, test cases can also be made to run by setting up rules in Jenkins. Once all the test cases pass and the build is successful, it can be deployed at the target location. The set of rules and configurations are often passed on as a deployment descriptor file to Jenkins.

There are a few other tools similar to Jenkins, such as Travis, Tekton, GitHub Actions etc. All of them provide almost the same feature set but with slight differences. For instance, Travis is not an open-source tool, but it has an integration with GitHub which means less hassle. GitHub Actions is also growing rapidly because of its advantages such as asynchronous pipelines, compatibility with

[5] I. Synopsys. *SCA*. [Online; accessed 2 May 2023]. URL: https://www.synopsys.com/glossary/what-is-software-composition-analysis.html.

[6] Atlassian. *Mend*. [Online; accessed 2 May 2023]. URL: https://whitesource.atlassian.net/wiki/spaces/WD/overview?homepageId=32342093.

[7] Wikipedia. *Jenkins*. [Online; accessed 3 September 2023]. URL: https://en.wikipedia.org/wiki/Jenkins_(software).

any environment, no installation needed etc. However, Jenkins offers plugins and APIs to integrate with other SCMs as well. Now even though these tools are automated, they need some level of monitoring overhead. As the size of the project grows, the complexity behind tracking changes across multiple systems is a major challenge. An overall bird-eye view of what is happening, what is stuck, what needs attention and what can be kept on the back stove should be available easily to decrease the development time.

2.7 Observability

Every software generates some outputs, these can be in different formats but there is always some meaning associated with them. Observability is the extent to which the information from these outputs can be used for knowing the internal state of the software. A system can be referred to as observable when the outputs generated from the system can be used for determining the state of the system[8]. There is a slight difference between monitoring and observability in general. Monitoring means watching the system, it can be for any purpose or sometimes with no purpose at all. It is like a meter recording the consumption of resources for billing purposes, while on the other hand, observability is about actually understanding the state of the system. There are three main pillars of observability: metrics, logs, and traces.

Metrics is often used in combination with time-series data. It is a numerical form of data that is collected over a time duration. Any information related to the state of the software system can be derived by applying predictions mechanisms on such metrics. Developers can also decide on a threshold and set up alert notifications if the metric of choice crosses the desired threshold. For instance, if the number of build failures per day crosses over 50 (threshold) then it can be inferred that something definitely is wrong with the code and the concerned administrator must be alerted about this incident. Metrics are preferred to be stored for longer duration of time in order to use historic data as a learning input to the prediction algorithms. Viewing metrics on a dashboard provides quick and more intuitive information as compared to reading the entire output log.

Logs are long textual pieces of data recording during the execution of processes within a system. It is considered a good development practice to create log statements after every milestone within a process and categorize them as per their criticality. These categorizations can either be custom or can be derived from a standard library, for example: info, error, debug etc. Although logs are simple to store and aggregate, they are difficult when it comes to extracting valuable information out of them. Because of their huge size, logs are usually not stored for a long duration of time and there are purging schemes to ensure they are cleaned up at a fixed duration. In order to make sense out of the logs, it is important that the logs are categorized and then depending on the criticality, they can even be showed on an observability dashboard to the developers.

Traces are the most important and the most complicated pillar of observability in complex software development. They contain information about which resources are being consumed by the application and how this interaction takes place. Because of the modular approach followed in the industries, it is often the case that there is a long chain of dependencies between the services. And if there is a failure within this chain of dependencies, traces are helpful in back tracking the root

[8] Wikipedia. *observability*. [Online; accessed 9 May 2023]. URL: https://en.wikipedia.org/wiki/Observability.

cause for such a failure. Without traces, it may get difficult to solve any errors as it will be more complicated to perform root cause analysis manually. Furthermore, in big projects such delays will result in increased downtime which can impact the reputation and cost of the service under failure[9]. For such deeply nested projects, distributed tracing comes into play for understanding performance issues and failures of any kind. To implement distributed tracing, developers have to perform code instrumentation and hence, it is often recommended to perform this step during the initialization of any project. This instrumentation can include trace IDs to identify when the execution leaves one service and enters another service with a different trace ID.

2.8 Dashboards

Dashboards are typical Graphical User Interfaces which are used to provide a quick overview over the chosen topic. They are designed to serve specific business purposes or a particular objective by providing visual representation of the collected information. They can be used for displaying key performance indices (KPIs), server downtimes, build states etc. Dashboards save time and improve decision making capabilities within an organization. They are mostly accessed as web applications with a data source linked to them, these data sources can either be real-time or buffered over a period of time. Earlier people used to read through logs to gain information about various aspects of the application but clearly it was not the optimal solution. Later Executive Information Systems (EISs) were used but they were unreliable because the data was not refreshed so frequently and hence, cannot be trusted[10]. In today's era, dashboards are a frequent practice for increasing observability scope of a system using metrics, logs, and traces.

Dashboards can be classified into various categories depending on their usage and the mode of usage. Based on their purpose, they can be analytical, strategic, informative, or operational. Furthermore, based on their mode of usage, dashboards can be web-based applications, desktop software or widgets. It is very important that the objective of dashboard is clear beforehand to have a useful result. There are a few design practices for designing an insightful dashboard that are listed below:

- The mode of usage: desktop, mobile, tablet, kiosks, or video walls etc.
- Proper visualization technique for every data type, such as line chart for time-series data.
- Counters or scores for strictly numerical data such as number of pull requests.
- Legends to make the visuals more comprehensive.
- Arrangement of various visual elements to facilitate easy extraction of information in a glance. For instance, placing counters on top can be useful if the user wants to keep track of them on a regular basis. 'Z' pattern is a method to arrange the elements according to their priorities from top to bottom and then from left to right.
- Usage of colors and font sizes which are also suitable for users with visual impairments.

[9] S. Lima. *observability*. [Online; accessed 9 May 2023]. URL: https://www.chaossearch.io/blog/three-pillars-of-observability-logs-metrics-traces#:~:text=DevOps%20teams%20leverage%20observability%20to,%3A%20Logs%2C%20metrics%20and%20traces.

[10] Wikipedia. *Dashboard*. [Online; accessed 9 May 2023]. URL: https://en.wikipedia.org/wiki/Dashboard_(business).

3 Use Cases

This Thesis revolves majorly around two use cases with an aim to compare the approaches and study the differences in different software development infrastructures. These use cases not only differ in software infrastructure but also in their team size, scope of the product, and project size. The use cases are described in a more detailed fashion in the next few sections. The first use case is Hyper Protect Virtual Server (HPVS) on Virtual Private Cloud (VPC) carried out at IBM Research and Development, Deutschland and the second one is an open-source project called preCICE developed by a group of developers from Technical University of Munich and University of Stuttgart, Germany [1]. Before going into the detailed explanation of both the use cases, there is a brief comparison between open-source and proprietary software in the below table[1] [8]:

Open Source	Proprietary
The source code is public	The source code is protected
Support is offered via documentations, source code inspections and GitHub issues	Support is offered by documentations and a dedicated support team
Users can use this software free of charge	Users may have to pay for using the services
Bugs are fixed either by the original developers or new collaborators	Bugs are fixed by the closed team of developers
Source code can be modified by the user for personal use	Users cannot modify the source code and have to rely on the development team
Defect detection is performed by the community of contributors	Defect detection is performed by the in-house project members
Example: Linux, preCICE etc.	Example: Windows, HPVS on VPC etc.

Table 3.1: Open Source vs Proprietary

3.1 IBM HPVS on VPC

HPVS is a feature which provides hardware-level security to an application on VPC. It ensures workload and application protection throughout the software life cycle from both internal and external potential threats. This feature is available for LinuxONE and IBM Z systems[2]. The goal

[1] S. Jena. *Difference between Open-source Software and Proprietary Software*. [Online; accessed 7 August 2023]. URL: https://www.geeksforgeeks.org/difference-between-open-source-software-and-proprietary-software/.

[2] IBM. *Hyper Protect Virtual Server*. [Online; accessed 7 August 2023]. URL: https://www.ibm.com/products/hyper-protect-virtual-servers.

of HPVS on VPC is to protect the cloud native applications utilizing confidential computing[3]. It ensures the end-to-end safety of data, which is stationary, in motion or when it is being used [5]. A brief overview can also be obtained from the Figure 3.1[4].

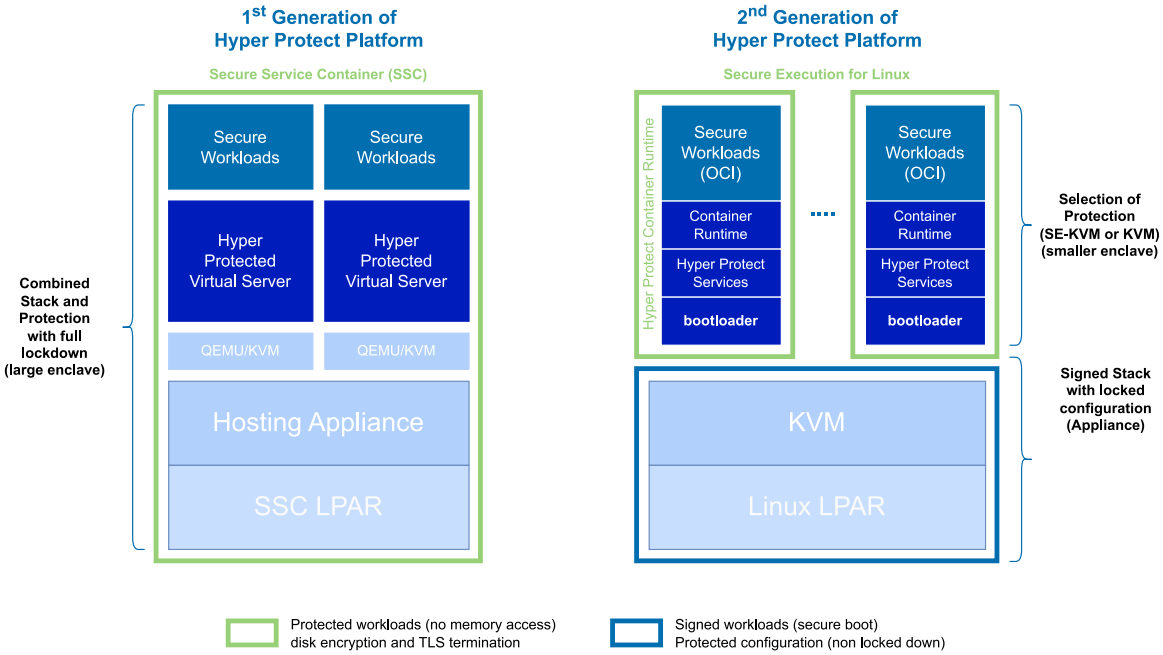


Figure 3.1: The Software Stack of Hyper Protect Platform

The team which is involved as participants or users of this study develop Virtual Server Images for HPVS on VPC. There are around 20 team members with a mixture of developers, architects, project managers, testers etc. The project itself is a moving project, meaning it is possible that it is transferable to a new team at some point in time. This transition phase is one important aspect in a software development life cycle and has to be dealt with using the proposed outcome of this study.

There are a set of standard tools used during the development of this project namely, Jenkins, GitHub and Renovate along with some tools as per an individual developer's preference such as Visual Studio Code. There are more than 250 code repositories used by the development team to deliver the final software product. These repositories have relations between them, so far discoverable up to a depth of 6. However, these dependencies are not centrally visible to the developers. Renovate is an SCA used to timely upgrade and manage the dependencies for around 170 repositories. It browses through the source code to find file formats such as `go.mod`, `requirements.txt`, `package.json` etc. to create a list of dependencies and use that list to further manage them. This list of dependencies are stored with Renovate and is not available to the developers as a visual aid. There are various scenarios where this has proven to be a bottleneck in the development cycles. Few of those use cases are listed below:

[3] IBM. *Hyper Protect Virtual Server on VPC*. [Online; accessed 7 August 2023]. URL: https://www.ibm.com/blog/announcement/ibm-hyper-protect-virtual-servers-for-virtual-private-cloud/.

[4] IBM. *The Second Generation of IBM Hyper Protect Platform*. URL: https://www.ibm.com/downloads/cas/GPVMWPM3#:~:text=In%20its%20second%20generation%2C%20the,systems%20for%20Kernel%20Virtual%20Machines..

- **On-boarding**: When a new developer joins the team, it is difficult to express the relations between the repositories to them and one has to rely on manual knowledge transfer sessions. The root cause being the repositories are inter-linked, but this linkage is not perceivable from an outside perspective. Also, there is no organizational pulse for the changes happening across various parts of the project.
- **Project Transition**: While transitioning a project from an existing team to a new team, there is again a similar problem as on-boarding new developer. However, with a more complex scenario that now the knowledge has to be transferred to more than one person, in most cases to around 10 to 15 developers. This makes it a very special and error prone case because one wrong understanding can lead to wrong changes and thus an avalanche of broken pipelines of dependent repositories.
- **DevOps configuration changes**: This is a very common and yet one of the most important issues. In a software development process, it is not uncommon to encounter situations where there has to be configuration changes in the pipeline files. If the developer responsible for making that change is not aware about the dependent repositories or if they miss a few dependent repositories, it may happen that the configuration change is responsible for their build failures (breaking changes). If this goes unnoticed it takes up a lot of debugging time to back-trace the changes and find the root cause. And for projects that have a tight release schedule, this may cause delays and possible need for pushing the release date.

These are just a few of the observed bottlenecks based on previous experience, there is a possibility that there are more. It is expected that having a central observability solution can decrease the shortcomings of having a modular repository structure i.e., increased complexity with number of repositories.

3.2 preCICE

preCICE is an open-source coupling library for partitioned multi-physics simulations, including, but not restricted to fluid-structure interaction and conjugate heat transfer simulations. The core library is written in C++, with adapters and bindings written in other languages such as Python, Fortran, C++ etc. [1]. The project itself is divided into different components, namely: Core Library, Language Bindings, Adapters, Utilities, Tutorials, Website and Documentation. There are around eight developers in the team, however, being an open-source project, it is possible that there are new collaborators every now and then. The majority of the development work revolves around GitHub, ranging from development to CI/CD activities.

The project itself has nearly 50 code repositories following a modular repository structure. So, the problems that were discussed in the previous section are to some extent true for this project as well. The complexity of this project is however less as compared to the previous use case. Hence, it is possible that the problems faced by the overall software life cycle are also fewer. Furthermore, the release cycle of this project is not so frequent which means if there are any failures due to dependency management, the team is to be able to afford some time in fixing them before the next release. The overview of the preCICE project can be inferred from the Figure 3.2 [1]. Some of the scenarios which act as probable use cases for this problem statements are:

- **Overwhelming Notifications**: Developers have expressed that in order to know what is going on in the project, they have to subscribe to a repository's activities. These subscriptions often result in email notifications to the subscribers. And it is possible that there are a lot of emails for an active repository even if the changes are not so significant. There should be a way to get information on demand and not overwhelm the subscribers.
- **Organizational Pulse**: While working with multiple repositories it is difficult to get an overview of the project as a whole. Even GitHub insights are specific to a single repository. Hence, there is a need to consolidate information at a single location and give insights about the organization as a single entity.
- **Dependency or Relations between Repositories**: As the number of repositories increases, their relations with other repositories also grow. This growth is not easy to perceive from an outside perspective. Furthermore, it may very well be the case where, after a few months, the author of the repository himself/herself forgot what other repositories are dependent on it. A visualization showing these relations will help resolve dependency upgrade failures more effectively and also present a clearer picture about the organizational dependency graph.

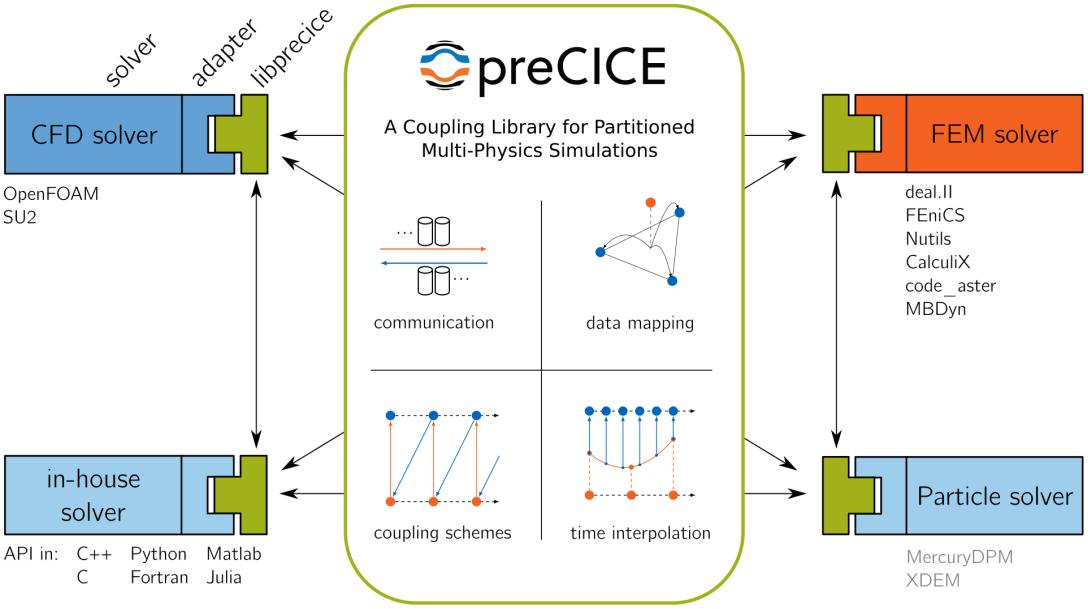


Figure 3.2: preCICE Overview

4 Results of the Initial Survey

At the beginning, it was necessary to first understand the existing system and then navigate around it like any new developer may do. Although, it was helpful in getting an idea about the complexity of the project infrastructure, it did not expose the bottlenecks which were expected by default. To better understand the system and create a baseline before starting with the research, surveys were conducted with the development teams. The motivation was to get to know the projects from developer's point of view and to gather information about their difficulties with the project. Changing the perspective and thinking from their point of view really helped in shaping the survey questions. These difficulties focused on issues related to the development effort in the respective teams. Since the use cases are different in nature, the survey was tweaked in several iterations to fit best with the team dynamics. Otherwise, it may have resulted in redundancy or rather unfruitful attempt of information gathering. In this section, the survey design is explained along with the learnings that helped in creating a baseline for this Thesis.

4.1 Survey Design

Before circulating the survey, there was an attempt made to choose a survey tool which is familiar to the developers and is not overly complicated. Few of the available options were Google Forms, Slido, IBM internal survey tool etc. The tool that was eventually selected to conduct the surveys is Slido. It is an easy to use tool which allows users to collaborate while creating a survey and to gather real-time analytics from the survey responses. It offers an easy sharing capability and also, no tutorial was needed before providing the participants with the actual survey.

The survey itself was divided into two parts namely initial and final survey for both the use cases. For the preCICE use case, the sample set contained eight developers out of which six participants took part in the initial survey. For the IBM use case, the sample set contained around twenty participants out of which twelve participants filled out the initial survey. Both the surveys consisted of roughly ten to eleven questions of various categories such as, MCQs, ratings, open text etc. The survey was carried out over a duration of about 1 week for each of the two set of participants along with in between reminders.

To improve the quality of the survey, questions were finalized after going through a few rounds of iterations. Some of the questions such as *How many repositories do you monitor on a daily basis* were used to get a quantitative measure of the vastness of the project. Many of the questions also aimed at bringing out the efforts that the developers are putting in. One of such question being *On an average how many links or relations per repository do you click/follow to discover repository activities such as pull requests, issues, commits etc.*. Answers of those questions made it clear that the problem was real and an improvement could really benefit them in many aspects. Furthermore, there were a few free text questions to understand the situation with more sincerity and think in the

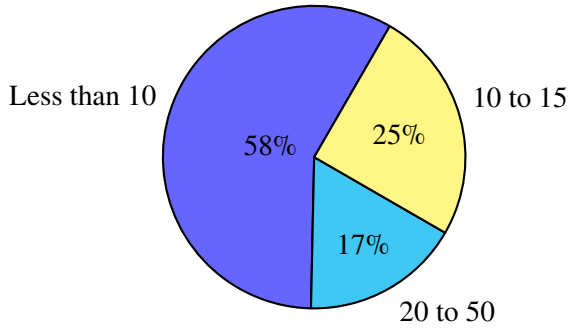
direction of their requirements. One such example would be *Please describe few pain points that you face in day to day development process related to the observability of code repositories*. The answer to that question helped in designing the elements of the solution for both the use cases.

The results presented in the following section have been interpreted manually after reading through all the results and statistics on Slido application.

4.2 Initial Survey: IBM

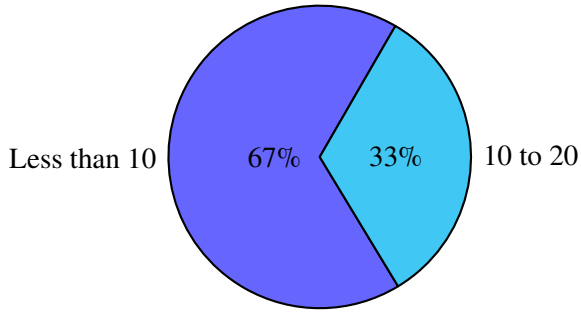
Participants: 12

Q1: How many repositories do you monitor on a daily basis?



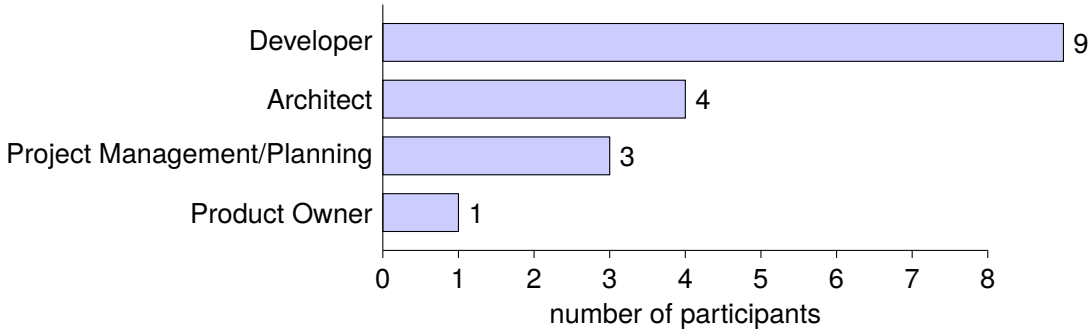
Average: 14 repositories per person.

Q2: In how many repositories do you make changes or edit on a daily basis?

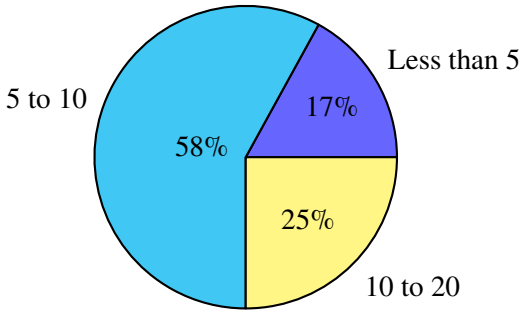


Average: 8 repositories per person.

Q3: Which of the given terms most closely relates to your work in the team?

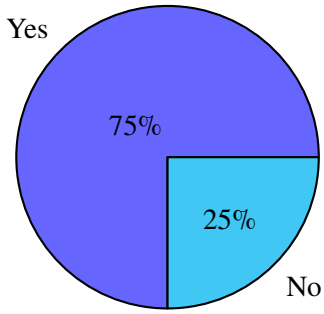


Q4: On an average how many links or relations "per repository"do you click/follow to discover repository activities such as pull requests, issues, commits etc.

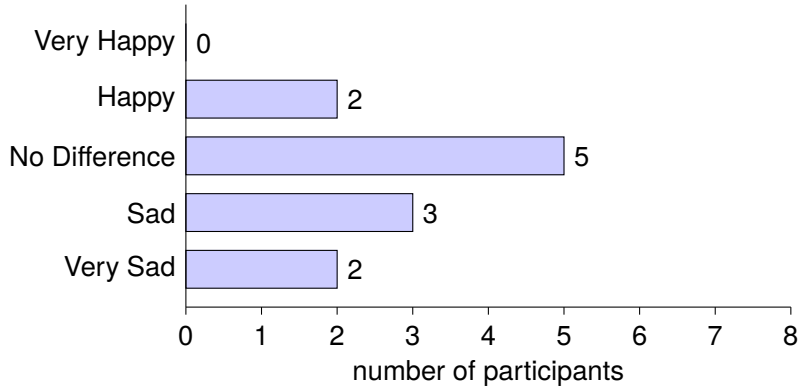


Average: 9 links per repository

Q5: Do you switch between different pages to get build state information about a repository?

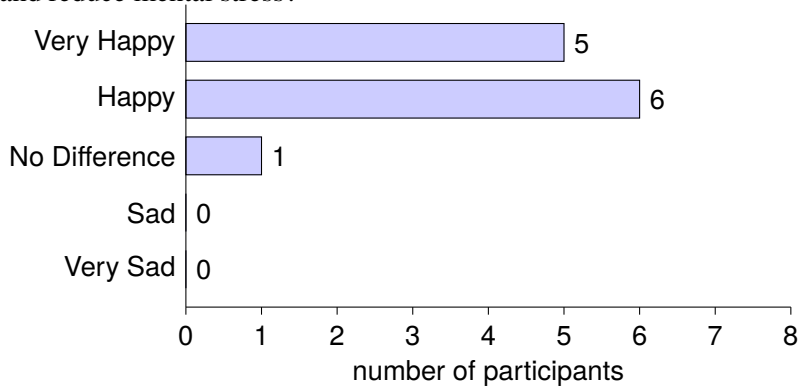


Q6: On what scale does it cause mental stress or increased mental effort to discover changes within a repository using the current arrangement?



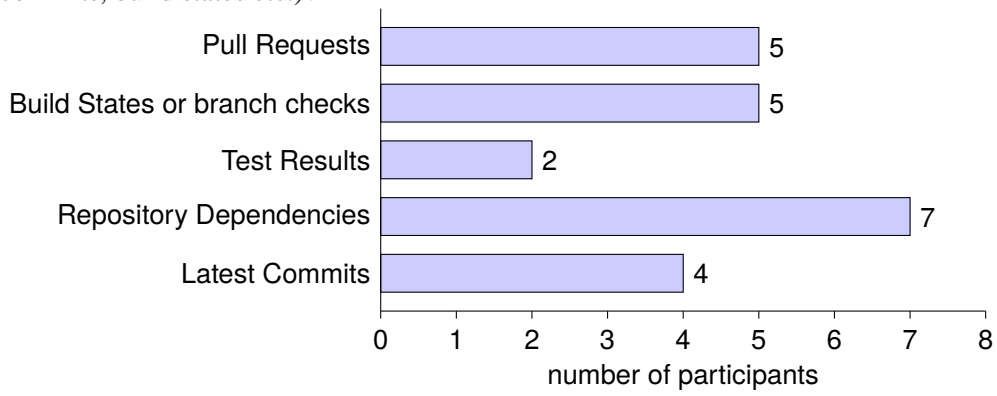
Average Score: 2.6

Q7: On what scale will some level of centralized observability help to ease the development effort and reduce mental stress?

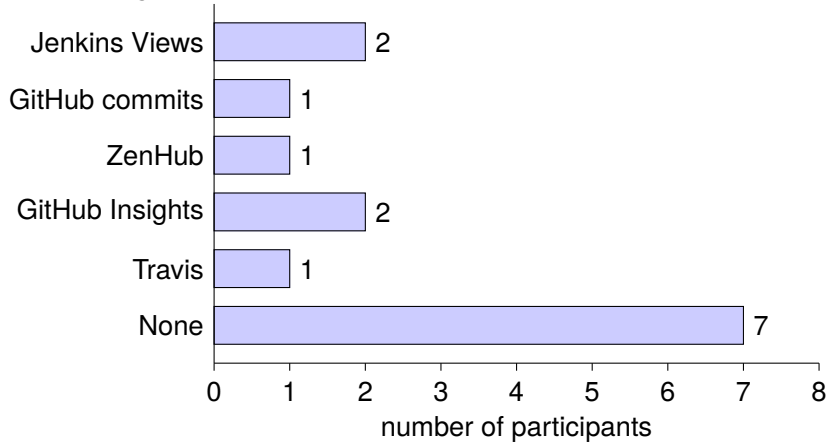


Average Score: 4.3

Q8: What kind of elements would you like to see on a dashboard for code repositories (like PRs, commits, build states etc.)?



Q9: Is there any tool that you're currently using to keep track of repository activities such as GitHub insights etc?



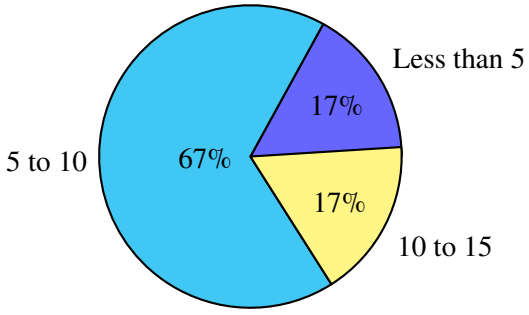
Q10: Please describe few pain points that you face in day to day development process related to the observability of code repositories.

- Nontransparent Dependencies between repositories.
- Status of repositories are not easily visible.
- Hard to track if build failure is due to a dependent repository or not.
- Missing an overview about the organization of repositories which makes debugging very difficult.
- Management of numerous tabs to back-trace the dependent repositories.

4.3 Initial Survey: preCICE

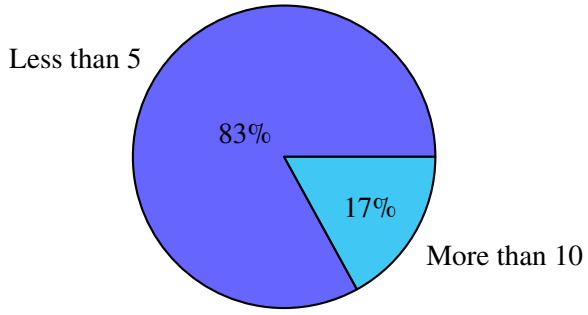
Participants: 6

Q1: How many repositories do you observe on a weekly basis?



Average: 9 repositories per person.

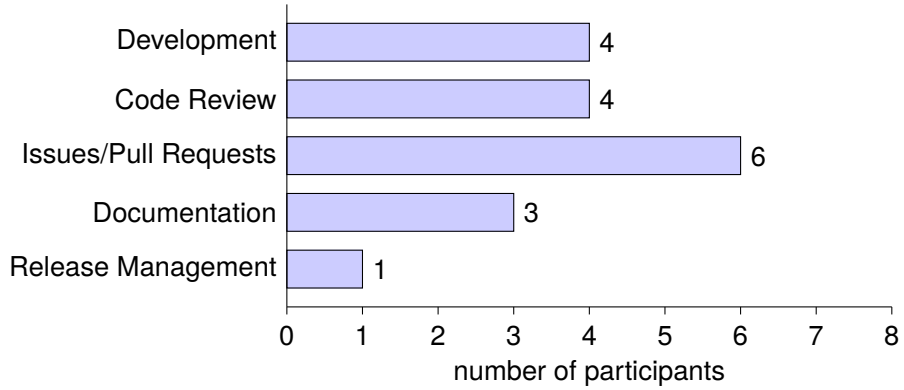
Q2: On an average how many links or relations "per repository"do you click/follow to discover repository changes such as pull requests, issues, commits etc.



Average: 4 links per repository

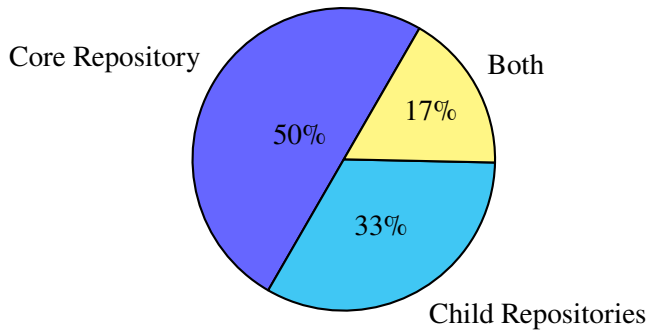
Q3: In how many repositories do you make changes or edit on a weekly basis?
Unanimous response: Less than 5 repositories.

Q4: What kind of operation do you perform most of the time?



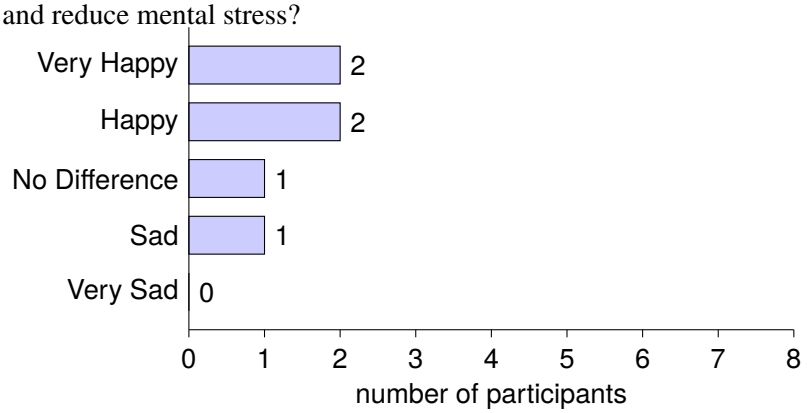
Q5: Is there some sort of categorization between the preCICE repositories, for example: core repo/ sub-repo, solvers/utilities or adapters etc.
Unanimous response: Core Library, Language Bindings, Adapters, Utilities, Tutorials, Website and Documentation

Q6: For instance, if we categorize the repositories of preCICE as core repo and child repos, which category of repositories do you work with more often?



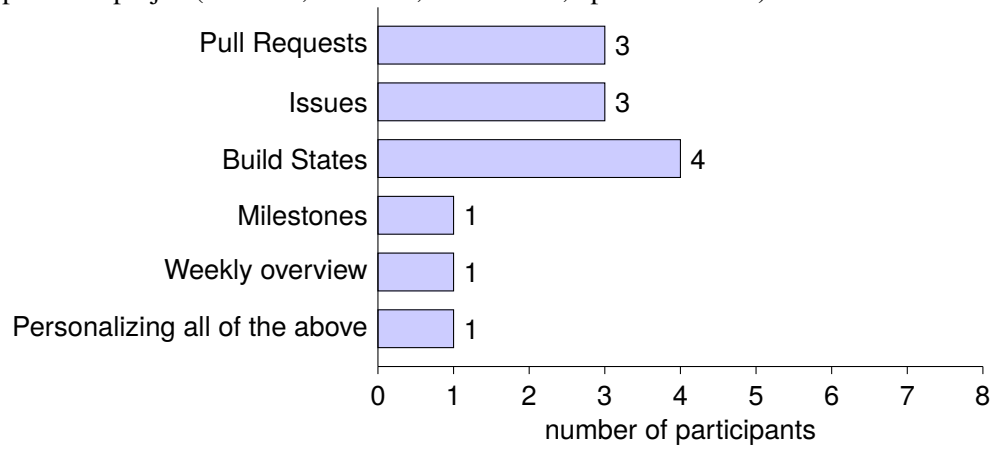
Q7: Do you get overview of build states of all the repositories in precice in a single view
Unanimous response: The Build State is not present in a single view for this project.

Q8: On what scale will some level of centralized observability help to ease the development effort and reduce mental stress?

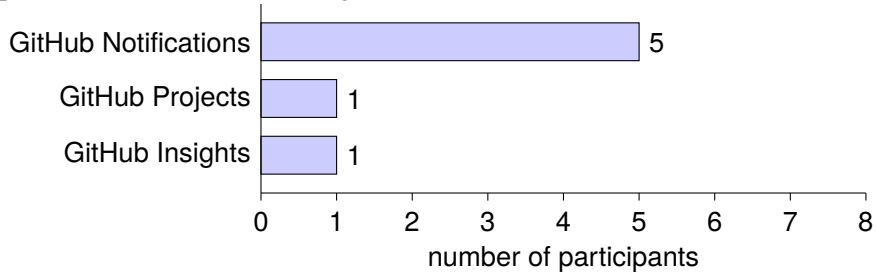


Average Score: 3.8

Q9: What kind of elements would you like to see on a dashboard for all the code repositories of preCICE project(like PRs, commits, build states, open issues etc.)?



Q10: Is there any tool that you're currently using to keep track of repository activities across preCICE such as GitHub insights etc?



Q11: Please describe few pain points that you face in day to day development process related to the observability of code repositories.

- Less Overview available with notifications.
- Quick summary not available unless the user 'watches' the repository proactively.

- GitHub notifications get crowded very easily hence, making it easy to lose track of repositories.
- No organizational pulse for the projects makes it hard to consolidate information from each repository individually.

The above survey gave clarity about the problems with the current setup and how it can be improved in terms of observability. The major issue that can be realised is that GitHub notifications is a subscription model, wherein developers get notified about activities they are subscribed to such as: watching repositories, related PRs and Issues, discussions etc. However, as the number of repositories increase this can quickly overwhelm the developer with a lot of information piling up on one another. This situation can be improved by a 'deliver on demand' approach, wherein developers are shown the information only when they proactively ask for it such as by opening a dashboard. There should be a consolidated view about status of PRs and issues rather than a detailed list of updates on those matters. This detailed list can sill be obtained by specifically opening GitHub through the consolidated overview but not unless it is interesting to the developer.

5 Initial Solution Design

After analyzing the requirements from both the use cases, there was an understanding that the solution should not be something that adds to a lot of maintenance overhead. Hence, there is no need to reinvent the wheel and something that is available in the open-source market must be used. There can be situations where all the requirements are not solved by the available solutions, in that case, there can be custom scripts to fill the gaps and serve the purpose to the users. There are several solutions that have been explored during this phase, however, none of them solves all the requirements on their own. In order to fulfill those requirements, there will be a couple of back-end scripts to support the existing application.

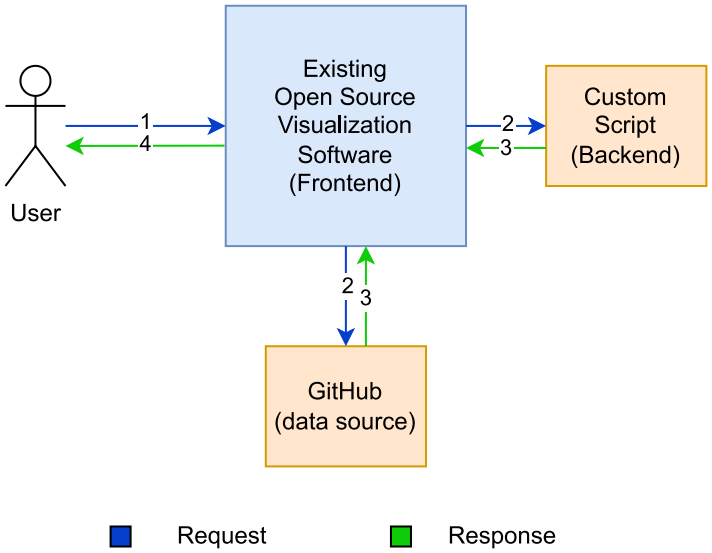


Figure 5.1: Initial Design of the Solution

The numbered arrows represent the sequence of request and response.

It can be inferred from Figure 5.1 that the user is interacting with an existing open-source software (visualization tool). This visualization tool gathers data related to the repositories from GitHub. If there is still some information not available to the tool directly from GitHub, it can use custom scripts to collect the missing data. All these elements are described as black boxes in this section, but they can be further refined based on the open-source software which was selected out of the available options in the software market. The focus while researching the available solutions was to find something which can gather data from the software development tools such as GitHub and visualize them in a legible manner for the users. The choice of the final software was made based on the feasibility aspects of that solution with the existing software development setup. Four such available options are described below:

1. **Kibana** is one of the important parts of the ELK (Elasticsearch, Logstash and Kibana) stack. It is a visualization tool which displays data from elastic store and lets its users navigate through this data at a very fast rate. Three major categories of features offered by Kibana are: Explore and Visualize, Management and Monitoring, Solutions. These categories further have a lot of features offered by them such as dashboards, charts, data tables, data visualizer, full stack monitoring, metric dashboards etc.[1]. All these features run in combination with the elastic search database. The overall or core feature is data querying and analysis. The users can query the data stored in Elasticsearch which is indexed for faster performance and this queried data can then be visualized on a dashboard using various visualizations such as bar charts, pie charts, tables, heat maps etc. The main usage of Kibana is to trace or diagnose applications by analyzing their log messages. There are a few advantages and disadvantages of Kibana which played a key role in deciding if it will be the chosen solution[2].

   Advantages:

   - Kibana is preferred for analyzing large amounts of log or textual data as it has stronger filtering ability for extracting more insightful data.
   - Provides some ready to use dashboards for visualizations of a few use cases.
   - Works in close association with Elasticsearch.
   - Easier to learn and intuitive.
   - It is an open-source visualization tool.

   Disadvantages:

   - Less customization options available and hence less flexibility in visual design.
   - Not a suitable choice for time-series data analysis or even for structured data.
   - Needs extra configuration for larger or more complex datasets.
   - Does not support any other data source except Elasticsearch.
   - Subscription to Elasticsearch is needed which may not be feasible for all the projects.

2. **Instana** was one of the obvious tools under consideration because it is a software tool from IBM and the primary use case of this study is also IBM. It is an observability tool majorly used for Application Performance Monitoring (APM). The goal of Instana is to provide APM with much improved performance and ease of use. Traditionally, before using any APM developers had to go through some weeks of training however, with Instana they can pick it up on the go hence eliminating the need to have a handful of trained or expert users. It also facilitates prediction and resolution of issues by utilizing AI powered capabilities. Few of the major features of Instana include full-stack observability, remediation, automation and intelligence and support for more than three hundred technologies[3]. There are a few advantages and disadvantages of Instana which are listed below, and they helped in deciding if it can be used for the final solution:

[1] ElasticSearch. *Kibana Features*. [Online; accessed 14 August 2023]. URL: https://www.elastic.co/kibana/features.
[2] A. Yigal. *Grafana vs Kibana*. [Online; accessed 14 August 2023]. URL: https://logz.io/blog/grafana-vs-kibana/.
[3] IBM. *IBM Instana Observability*. [Online; accessed 14 August 2023]. URL: https://www.ibm.com/products/instana.

Advantages:

- It is a powerful tool for monitoring the performance of applications.
- Offers dynamic querying for application parameters.
- Has AI enabled features for faster and accurate issue resolution.

Disadvantages:

- Not an open-source software and hence, may not be feasible for small scale or open-source projects.
- Needs integration with Grafana for observability of GitHub repositories.

3. **Datadog** is another major visualization software which provides observability with good performance and security. Some of the key features of datadog are APM, automatic support for infrastructure issues, log analysis, full stack dashboards and alerts for early detection of issues. It offers a SaaS (Software as a Service) based approach, which makes it easy for on-boarding purposes however, there are times where users might want to use the services on their own infrastructure (on-premises) and that cannot be achieved while using Datadog. It also offers a watchdog feature to enable machine learning based alerts for identifying anomalies or defects in the applications under consideration. Unlike traditional systems datadog does not generate reports as PDFs it exposes Representational State Transfer (REST) Application Programming Interface (API)s which can be used to get the desired data on demand in JavaScript Object Notation (JSON) format[4]. Below are some of the advantages and disadvantages mentioned about datadog[5]:

   Advantages:

   - Provides good set of features for analyzing the infrastructure of applications.
   - Machine Learning based alert capabilities are also present.
   - Various data sources can be connected with several integration opportunities.

   Disadvantages:

   - Proprietary and hence expensive as compared to open-source solutions.
   - Needs to be learnt, ergo not simple to implement.
   - Allows less customization possibilities which provides less flexibility.

[4] Datadog. *Datadog Unified Observability and Security*. [Online; accessed 14 August 2023]. URL: https://www.datadoghq.com/lpg/?utm_source=advertisement&utm_medium=search&utm_campaign=dg-google-brand-ww&utm_keyword=%2Bdatadog&utm_matchtype=b&utm_campaignid=9551169254&utm_adgroupid=95325237782&gad=1&gclid=EAIaIQobChMI94HNl4fcgAMVsRtlCh2HaAm8EAAYAiAAEgL_0fD_BwE.

[5] A. WELEKWE. *Datadog vs Instana*. [Online; accessed 14 August 2023]. URL: https://www.comparitech.com/net-admin/datadog-vs-instana/.

4. **Grafana** is an open-source software with a free tier service available. It allows users to connect with different kinds of data sources and visualize the data using time-series, bar charts, pie charts, tables etc. [2]. There are two major flavors of Grafana, one for the cloud version and the self-managed version. In both these versions, one could opt for open-source or enterprise level functionalities. It also offers a forever free plan for the Grafana cloud flavors which comes with some data limits but is still suitable for small scale projects. A major advantage of using a cloud version is that the users do not need to maintain the versions or updates or scaling like they will have to in case of an on-premise setup. Some of the key features of Grafana include querying, visualizing, and alerting[6]. There is also an option to create a dashboard just by configuring a JSON file. These configurations can be done as per the extensive guide provided by the Grafana team on their official website[7]. Just like other available solutions above, there is a list of advantages and disadvantages mentioned below:
   Advantages:
   - It is a free and open-source framework with an active community of developers.
   - Customization opportunity is high and has a wide range of visualization options for more informative and interactive dashboards.
   - It supports various data sources and is often preferred for monitoring and analyzing time-series data.
   - Users can define thresholds on certain metrics to receive notifications or alerts.
   - There are a lot of plugins available for easy integration including CI/CD data source plugins like GitHub.

   Disadvantages:
   - Because of many possible customizations, there is a lot of opportunity to explore, and hence more time needed for learning and implementation.
   - Can consume more resources if the data-stream is of high frequency.
   - Generally not the preferred method for analyzing textual data or log-based data source.

[6] G. Labs. *Grafana Introduction*. [Online; accessed 14 August 2023]. URL: https://grafana.com/docs/grafana/latest/introduction/.

[7] G. Labs. *Grafana Provisioning*. [Online; accessed 28 August 2023]. URL: https://grafana.com/docs/grafana/latest/administration/provisioning/.

6 Solution

The solution was designed based on the pros and cons of every software explained in the previous chapter. There was more inclination towards Kibana and Grafana because of their open-source nature and hence Instana and Datadog were no longer considered [4]. Out of Kibana and Grafana the differentiating factor is that, to use Kibana there is an increased overhead of performing the data extraction step. Furthermore, all the gathered data has to be kept in elastic storage which will add up to the infrastructure and maintainability aspects of the overall solution. On the other hand, Grafana has the capability to collect data on its own from GitHub using a GitHub data source plugin. And if there is any gap in the collected data then it can be compensated by using REST APIs as an endpoint to feed data to the Grafana dashboard. This reduces the need to maintain an elastic storage and provides the flexibility to use manageable scripts in any programming language of choice. Out of all the other advantages, the one that stands out and coincides well with the use cases is the ability to use a GitHub data source plugin. Since it is an open-source project, these plugins can also be attempted to be modified in future if needed. Naturally, Grafana was the choice made at the end of this analysis.

In the initial surveys, it can be observed that the developers would very much like to have an overview of dependencies and current state of their repositories along with other metrics such as pull requests, issues etc. However, the GitHub data source plugin does not provide functionality to view such a dependency graph or the current state of any repository. This is the gap that will be filled by using custom scripts which will feed data to the Grafana dashboard using REST APIs. Now, to keep these scripts small and maintainable it is important to choose a programming language which needs no special training and is light weight. Hence, Python was considered to be the best fit as it is easier to understand, maintain and provides the ability to develop REST APIs at very minimal effort using its Flask library. There was also a slight difference in the design and implementation of both the solutions because of their underlying difference in data source. For the IBM project, the data source for creating a dependency graph was Jenkins as their dependency management tool i.e., Renovate has a periodic job on Jenkins and that job identifies the dependency between repositories. This information was found in GitHub for the preCICE project via the GitHub dependency graphs.

Grafana allows the users to be able to create roles and accordingly grant permissions for example, only admin can edit the dashboard. These permissions can be changed by configuration files hosted on the grafana server. The admin can also create teams and control the visibility of various dashboard elements. A dashboard consists of a set of variables and visualization panels that contain the data retrieved from the data sources. Detailed explanation about both the solutions have been made in the following subsections.

6.1 IBM

In this section, the main focus will be on the solution design for the HPVS project. The solution starts with analysis of what all information is already provided by the grafana GitHub data source. This data source is a plugin which makes calls to the GitHub GraphQL APIs and returns the result to Grafana visualization panels[1]. During this analysis it was found that pull requests, issues etc. are all part of this plugin and hence there is no need to extract them externally. Dependency graph and status of the repositories were two major things to be considered while implementing the custom scripts. The idea was to keep these two aspects separate from the beginning so that there is better modularity in the design. If there is any need to change something in the Python scripts, the GitHub dashboard will remain untouched and vise versa. The structure of the application can be inferred from Figure 6.1.

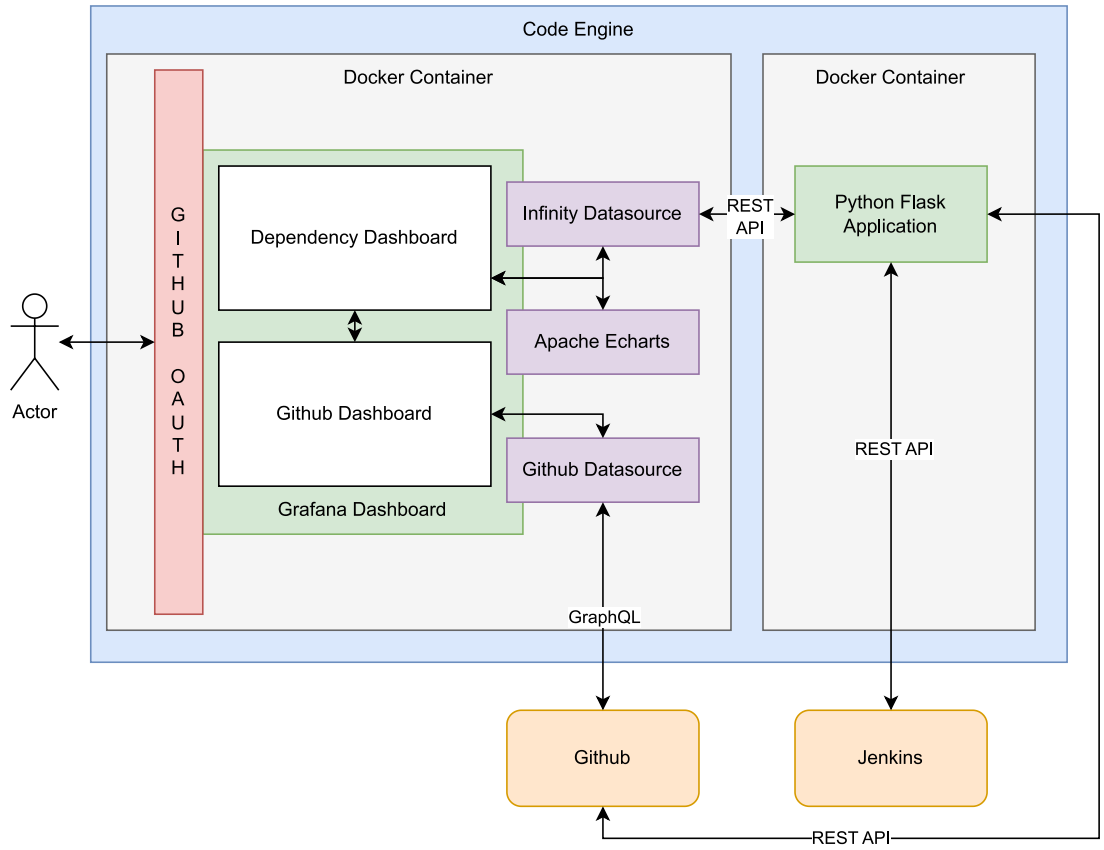


Figure 6.1: Solution for Use Case 1: HPVS on VPC (IBM)

[1] G. Labs. *Grafana GitHub datasource*. [Online; accessed 15 September 2023]. URL: https://grafana.com/grafana/plugins/grafana-github-datasource/.

6.1.1 Development Process

The solution is developed by following an agile approach with the help of GitHub Project Board (Kanban-like board). GitHub Enterprise is used for source code management along with a monolithic repository structure. The branch strategy is fairly simple, the new changes are a part of the test branch and once they are verified, they can be merged to the main branch. The development tools involved in the solution are Visual Studio Code and podman (docker and docker-compose alternative). The planning of the project was done using milestones and issues on GitHub which then gets pulled into the project board columns (to-do, review, done). The goal of using the project board is to make sure all the tasks from the 'to-do' column are moved to the 'done' column by the end of this study. Special attention is given to tag all the commits with proper issue numbers to maintain traceability if needed. Following some of the mentioned software development practice helped in creating a more maintainable software with the agility to incorporate changes when needed.

6.1.2 Backend

The initial approach to get the repository status was to use Jenkins REST APIs and present the status of pipeline in a table on the dashboard. But the response time of Jenkins API was not acceptable since it made the request call to often timeout. Hence, a new data source was needed to replace Jenkins. After consulting with a few developers, it was realized that the checks that run over a branch during any commit will be more useful since it is a list of checks and not just a check on the build state. Following that suggestion, the Jenkins API was then replaced with the GitHub REST API which was performing better and was even more improved with the multiprocessing pool used in the Python script. Furthermore, to get the dependency information about the repositories, there is a tool called Renovate which identifies dependencies and manages them. But it does not provide any visualization for the developers. Also, there are no APIs offered by Renovate to extract the dependency information. This barrier was overcome by creating a parser in Python which scrapes the logs from the renovate job and creates a JSON file representing a uniform tree structure for the repositories. This JSON file also has supporting informations such as source Uniform Resource Locator (URL)s, updates, artifactory URL etc. This rich set of information can prove to be handy and hence was not discarded during the parsing process. This JSON file then acts as an input to another Python script which converts it to a set of nodes and edges and then feeds this information to the grafana dashboard. The dashboard uses the Infinity data source plugin[2] to communicate with the Python server. This JSON data is then read by the Apache Echarts plugin[3] and converted to a network graph using a small code snippet written in typescript. This script is embedded in the dashboard configuration itself. The flow of operations from parsing the logs to generating the graph is shown by Figure 6.2.

[2]Sriram. *Grafana Infinity Datasource*. [Online; accessed 10 September 2023]. URL: https://sriramajeyam.com/grafana-infinity-datasource/docs/installation.

[3]V. Labs. *Apache ECharts Panel*. [Online; accessed 10 September 2023]. URL: https://volkovlabs.io/plugins/volkovlabs-echarts-panel/.

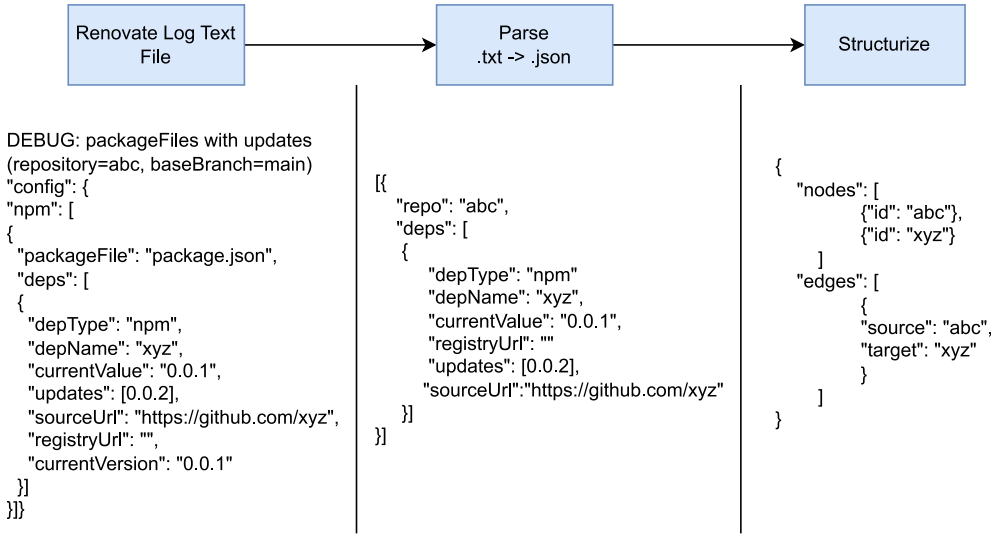


Figure 6.2: Dependency Graph Script Flow for Use Case 1: HPVS on VPC (IBM)

The main motivation of creating a JSON file was to provide a generic structure to the data collected from renovate. So that, if this data source changes in the future only the first step needs to be modified and once it matches with the JSON structure, no change is needed in the further steps. Also, the data stored in the JSON file acts as a local database to get information for all the information on the dashboard, unless refreshed. This not only promotes re-usability but also makes the code more maintainable in terms of separation of concerns and root cause analysis.

6.1.3 Frontend

Now that a basic understanding of the backend process is established, the frontend side can be looked at with more details. While using grafana as a visualization tool, one can provision the entire dashboard by using a few configuration files and embedding it into the grafana docker image. These configuration files are either written in JSON or in YAML ain't markup language (YAML) which facilitates the user to make changes quickly and keep the code base light weight. The data sources used in this project were infinity (yesoreyeram-infinity-datasource) and GitHub (grafana-GitHub-datasource), these data sources have to be configured in a YAML file and placed in a specific location so that grafana server could find it on startup. Similarly, dashboard configurations also have to be configured in a similar YAML file. All these files are present in `/etc/grafana/provisioning/` directory, which is a generic path of all systems. When the grafana server starts, it scans through these provisioning folders to check any custom implementations otherwise it continues with its default values. This folder has sub-folders pointing to a specific purpose like dashboards for dashboard configurations, data sources, plugins etc. Once the dashboard was created through the grafana user interface, it was exported as a JSON file and placed inside the `/etc/grafana/provisioning/dashboards` so that it can be automatically recognised on the next startup.

The snapshot of the **GitHub Dashboard** is shown in Figure 6.3. The fields chosen for the dashboard are based on the survey inputs received in the initial stage. The top bar has a time picker which lets the user select time based on which, rest of the dashboard elements are loaded. Dashboard can also

be manually refreshed using the refresh button on the top right. The variables that Grafana uses to query the data are placed at the top of the visual elements. In this particular dashboard, the list of repository is one such variable. Furthermore, there is a button in-line with the variable labeled as 'Dependency Dashboard', it is a toggle switch to transfer the user to the connected dashboard. Major components of this dashboard include statistics for active PRs, open issues, total PRs and issues, commit and average PR open time. These statistics display the gist of activities happening in a repository. They are followed by four tables giving some details about the statistics shown above. These tables have information regarding author, state, issue number, URL of pull request, description etc. The user also has the option to navigate to GitHub using the URLs given in these tables. This dashboard has more fields which can be added as per the use case such as milestone, tags, release, commits etc.

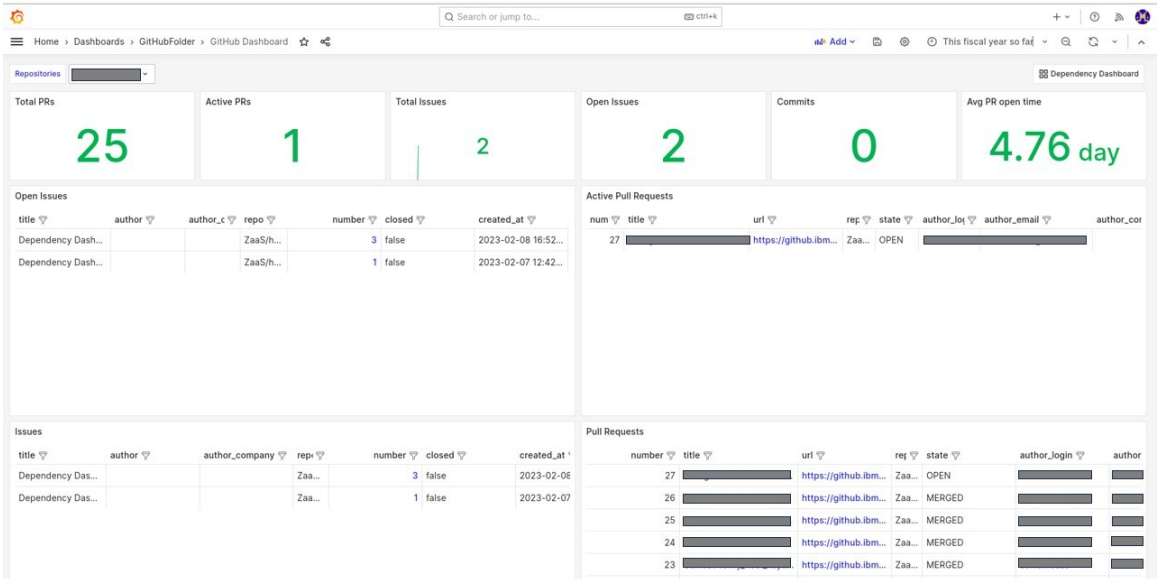


Figure 6.3: GitHub Dashboard: HPVS on VPC (IBM)

The **Dependency Dashboard** has some similar elements such as a refresh button, variable list, and a dashboard toggle button. However, the main components differ a lot compared to the GitHub Dashboard. It contains two visualizations, one being the dependency graph and the other being a table showing status of all the repositories. The IBM use case has one extra table showing the information captured by renovate for the dependencies. The information includes the current version, updates, artifactory URL, dependency name etc. The snapshot of the dependency dashboard for IBM use case is shown in Figure 6.4.

6.1.4 Authentication

Other than the dashboard and data source properties, there are also some application-level properties that need to be defined. These properties are mentioned in an initialization file that is read during the initialization of the grafana server. The file is called `grafana.ini` and is placed in `/etc/grafana/`

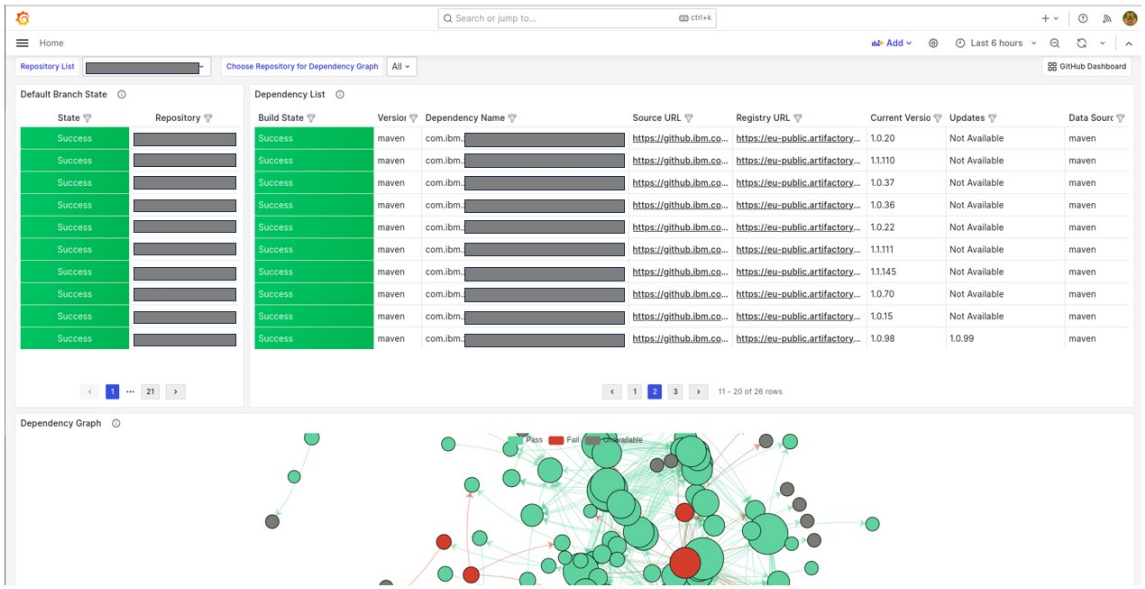


Figure 6.4: Dependency Dashboard: HPVS on VPC (IBM)

The dependency graph has not been shown fully as it is already shown in Figure 6.8

folder. This file contains information related to security aspects such as login, oauth, proxy, analytics and many more. For the IBM use case, it was used to configure the authentication via GitHub OAuth application. This enables the users to simply use their existing GitHub accounts as a sign in option and eliminated the need to remember new credentials. It can be further restricted to users of a particular GitHub organization etc. It is important to keep the file structure as it is, to avoid exceptions such as dashboard not found when user logs in to the grafana interface. All the possible configuration options can further be explored at Grafana's official provisioning documentation[4]. For enabling the GitHub OAuth functionality, one needs to create a GitHub OAuth application by following a few simple steps given on the official GitHub documentation[5]:

This will generate a Client ID and Secret which can be used in the `grafana.ini` file to enable the GitHub authentication system. There are also options to map roles as per the login usernames to enable the authorization functionality. Currently, there are two roles in the solution namely, admin and viewer wherein admin can also edit the dashboard and create or delete users. However, the viewers can only view the dashboard to which they have been granted access. This layer of security was critical in this use case because unlike an open-source project, all the details displayed on the grafana dashboard are confidential and it has to go through proper channels before accessing that information. Although, this may not be so critical for any open-source project as all the dashboard information is anyway present on GitHub but in a rather scattered way.

[4] G. Labs. *Grafana Provisioning*. [Online; accessed 28 August 2023]. URL: https://grafana.com/docs/grafana/latest/administration/provisioning/.

[5] GitHub. *Creating an OAuth app*. [Online; accessed 10 September 2023]. URL: https://docs.github.com/en/apps/oauth-apps/building-oauth-apps/creating-an-oauth-app.

6.1.5 Infrastructure

Once the backend and frontend are working as individual applications, it is important to make them as services to follow with the micro-service architecture approach. For ease of deployment, both the applications have been containerized using Dockerfiles. This file contains simple instructions which tell the system what all steps are needed to run the service on a specified port number. For the development environment, it is sufficient to use a docker compose file and run both the applications as services using a common network connection. This network allows the frontend to communicate with the backend by using just the service name in the URL instead of the entire IP address of the server. For deploying this application and making it available to its users, Code Engine was chosen as the preferred cloud platform. It being an IBM product, was easily accessible and the configurations are also very intuitive. The major benefit of using Code Engine is that it is serverless, which means the container instances are up only when someone is accessing the application. It is the ultimate pay per use model, and it comes with inbuilt environment variable management where secrets can be stored as variables and injected to the containers at run-time. In this project, GitHub access token, jenkins access token, jenkins username and a few more environment variables are of major concern for running the application successfully. Both the frontend and backend communicate with each other user the exposed ports in the private network. Only the grafana interface is exposed on the public network but since it is protected by the GitHub OAuth and an ingress proxy by Code Engine it is still considered to be safe. While this infrastructure was used for the mentioned reasons, it is totally possible to use a completely different cloud provider or technology if preferred. This is a huge advantage obtained by containerizing the application, it becomes portable and any environment supporting docker images can host it seamlessly.

6.2 preCICE

A major aspect of the solution is to be reusable for other complex software development projects. Since most of the components of the solution are true for preCICE also, in this section more focus is given on the differences in the implementation. The solution is tweaked at a few places and is displayed in Figure 6.5.

One major difference between both the use cases is the change of data source. It can be observed from the above figure that Jenkins is no longer a part of the system, which means the dependency graph is now inferred using GitHub itself. When a user wants to see the dependencies in their project, they can navigate to the insights section of every repository and view this information through the GitHub interface. However, this information is present only inside each of the repositories and there is no organizational level graph showing this information as a bigger picture. This is achieved by combining all these dependencies into a single graph where nodes represent repository and edges represent the relation. This information is obtained using the custom Python script which makes graphQL API calls to the GitHub end point and converts the collected data into a tree data structure. The flow of this script is similar to Figure 6.2 with a minor difference in stage 1. The modified flow of script can be observed in Figure 6.6

The first stage collects dependency information for each repository and combines this information into a generic JSON file. The logic from there on-wards is same for both the solutions, this is a key benefit one can utilize while using the solution for another software development project in the

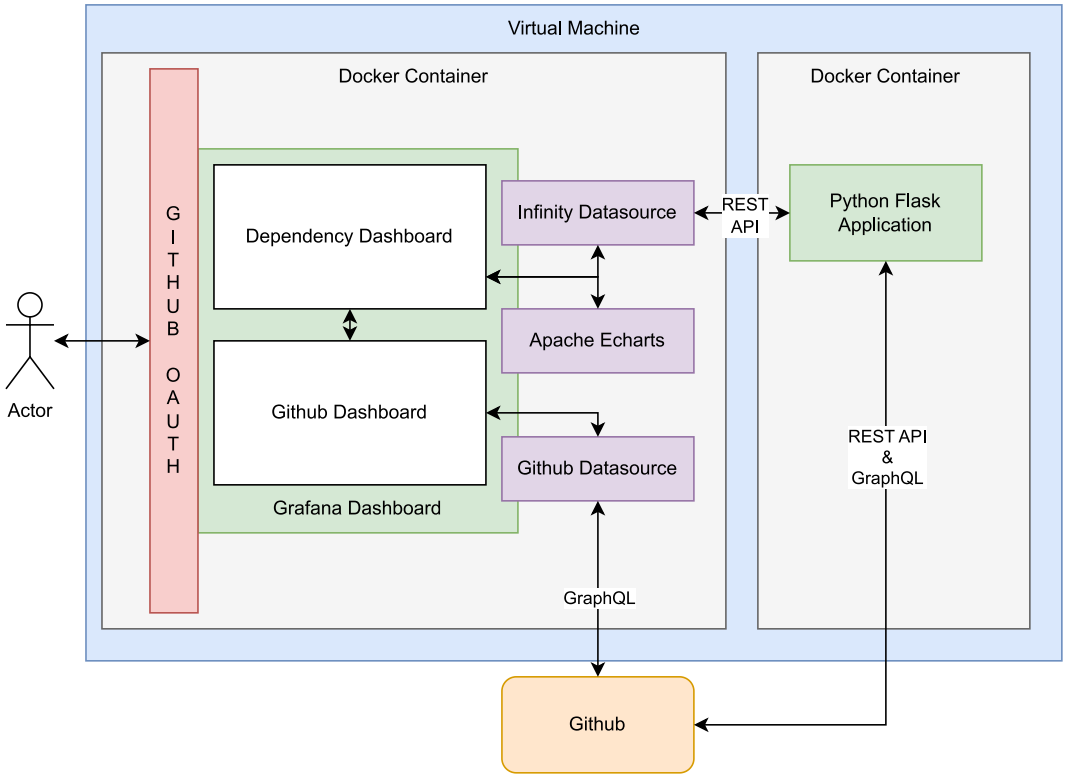


Figure 6.5: Solution for Use Case 2: preCICE

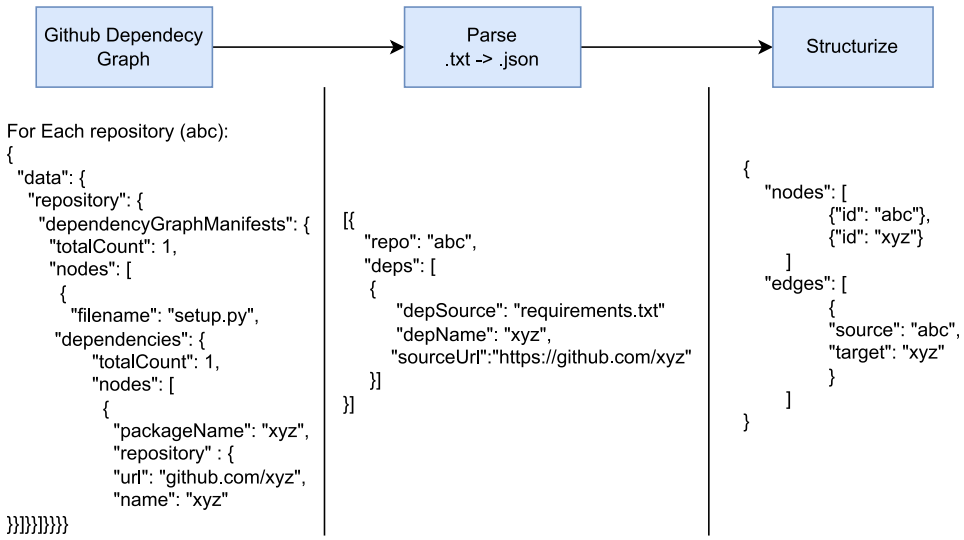


Figure 6.6: Dependency Graph Script Flow for Use Case 2: preCICE

future. Moving on to more differences for preCICE, the status of repositories is also obtained by using GitHub REST APIs. But these APIs are not returning proper response and seem to have a bug

which was properly reported during the process[6]. Since there was no resolution provided during the course of this research, this API currently is under consideration for alternative options. Another significant difference is the infrastructure used for deploying the applications. For the preCICE use case, Baden-Württemberg Cloud is preferred as hosting infrastructure for this solution due to its ease of availability and tie-ups with the University student accounts. Towards the end, this infrastructure had some challenges which are explained in Section 7.3. Due to these challenges, IBM Code Engine was used as a temporary (until the end of this research) deployment environment. Even though the cloud provider initially was different but the concept of containerizing the components remained in place. It is simply because containerization provides easy maintainability options and can be considered as portable if it needs to be shifted to a different cloud provider in future. The images for these docker containers are published on GitHub container registry which works similar to as of Docker Hub and provides automation options while making any new changes in the repository. Furthermore, the GitHub APIs used during the entire process have some usage limits which have to be taken into consideration. Since the GitHub instance being used at IBM is GitHub Enterprise, their limit is up to 15000 API calls per hour[7]. This is considerably higher as compared to the public GitHub instance which offers only 5000 requests per hour.

On the frontend side, there are two major differences. First one being, the dependency graph for IBM shows only internal dependencies as they are large in number but for preCICE, even external dependencies are shown in the graph. The second major difference is that there is one extra table on the dependency dashboard of IBM as their renovate tool provides additional information about each dependency and that could be of interest for the developers. However, GitHub dependency graph does not provide such surplus information such as updates, artifactory url, warnings etc. In future, if such details are available, it is fairly easy to add a visualization to the grafana dashboard similar to the other tables already provided in the current solution. Since, the dependency dashboard for both the solutions differ in design, a snapshot of it is provided in Figure 6.7.

A summary of differences between the two solutions is outlined in the table below:

Criteria	IBM: HPVS on VPC	preCICE
Data source for dependency graph	Renovate and Jenkins	Github Dependency Graph
Dependency Graph	Only internal dependencies	Internal plus external dependencies
API Limit (per hour) of the data source	15000 (GitHub Enterprise)	5000 (Public GitHub)
Docker Image Registry	IBM cloud registry	GitHub container registry
Cloud Infrastructure	IBM Code Engine	Baden-Württemberg Cloud Virtual Machine
Size of the project (repositories)	more than 250	more than 45

Table 6.1: Differences in the proposed solution

[6] supercobra. *GitHub API "combined statusïs always Pending*. [Online; accessed 7 September 2023]. URL: https://github.com/orgs/community/discussions/58407#discussioncomment-6679878.

[7] GitHub. *Rate Limit*. [Online; accessed 31 August 2023]. URL: https://docs.github.com/en/rest/overview/resources-in-the-rest-api?apiVersion=2022-11-28#rate-limiting.

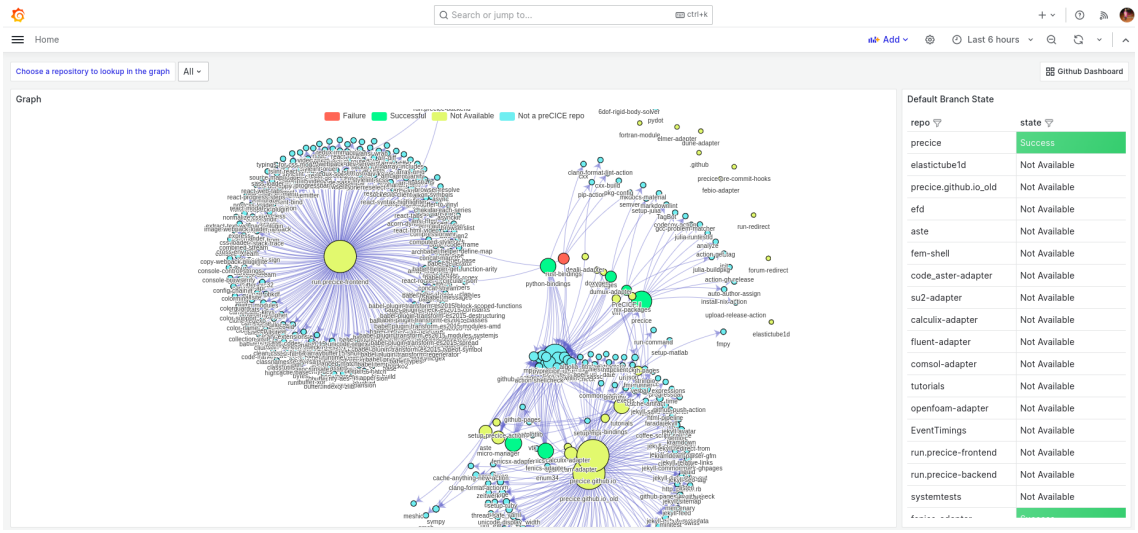


Figure 6.7: Dependency Dashboard: preCICE

During the course of planning and development, there have been several improvements made in the process to make sure it is an incremental solution. Some of those improvements have been highlighted in this section.

1. **Authentication**: The authentication earlier was based on default grafana admin and user credentials. In the later stages, GitHub OAuth was integrated to make the application more secure.

2. **API Response Time and Limit**: The APIs being used are cached to provide faster responses when the same query is made multiple times. This not only takes care of response time but also keeps a check on number of requests that are sent to GitHub as there is an upper limit of 15000 and 5000 calls per hour (mentioned in Section 6.2). Furthermore, caching has also been implemented for Python functions which always return the same result. It reduced the response time for the Flask API calls made from the Grafana server.

3. **Reusability**: The software development was improved to make it reusable for other projects as well. All the configurations were pulled into separate configuration and environment variable files. This gives a very good scope to make the solution as reusable as possible. While on-boarding this solution for another project, one needs to simply change the configuration parameters and use proper environment variables while running the docker container. The solution which took around 2-3 months of development time initially can be configured for a new project in just 2-3 days.

4. **Dependency Graph**: The dependency graph started with a very basic look, few nodes and edges with no customization. In the later stages it got colored nodes with their size representing the degree of connections from each node. It not only made it to look more intuitive but also made it possible to interpret things more clearly. The stages of dependency graph was recorded at random intervals during the development process (The graph nodes originally have repository names as label but that is kept hidden due to confidentiality purposes):

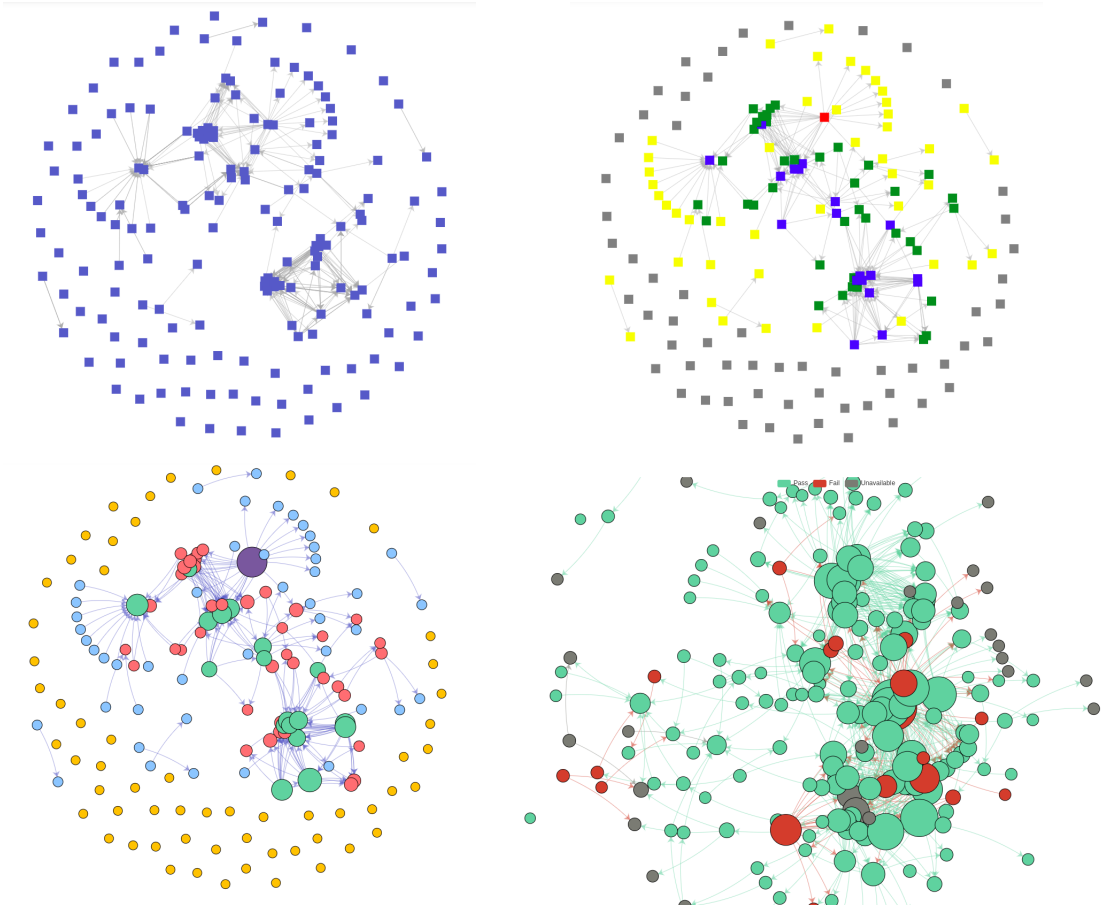


Figure 6.8: Evolution of the Dependency Graph

The evolution in the graph is in the form of node properties such as colour, shape, size, and connectivity. From the top left to the bottom right, the graph evolved to be more connected and intuitive for the users.

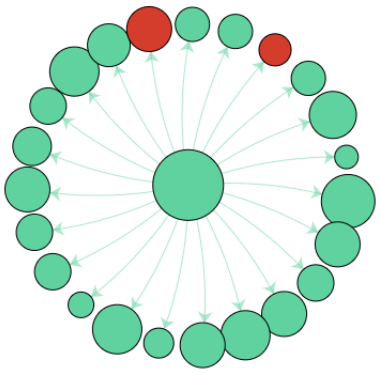


Figure 6.9: Filter Capability of the Dependency Graph

Graph can also be filtered over a repository to view only the associated connections.

5. **Repository Status**: Initially, the status of repositories was corresponding to their build states. However, this was later changed to their default branch status as it was more useful to the developers according to their feedback which was recorded during various demo sessions during the development process.
6. **Generalization**: The URLs used for making the API calls were embedded in the grafana dashboard configurations. Which means, if the backend URL changes then there is a need to modify the dashboard configuration also. This was later replaced as a variable which can be resolved during the start of application. For the IBM use case, it is stored as a environment variable and gets replaced with its value when the grafana server starts. For the preCICE use case, it is simply the name of the service mentioned in the docker-compose file. Docker can resolve the server using the common network properties that both the services share. This gives the possibility to migrate between environments more easily than with earlier arrangement.

7 Results of the Final Survey

The outcome of this research was also recorded using surveys. These surveys follow similar design and rules as described in Chapter 4. Furthermore, there are a couple of challenges which were faced during the implementation part of the solution and some of them are also explained in this chapter.

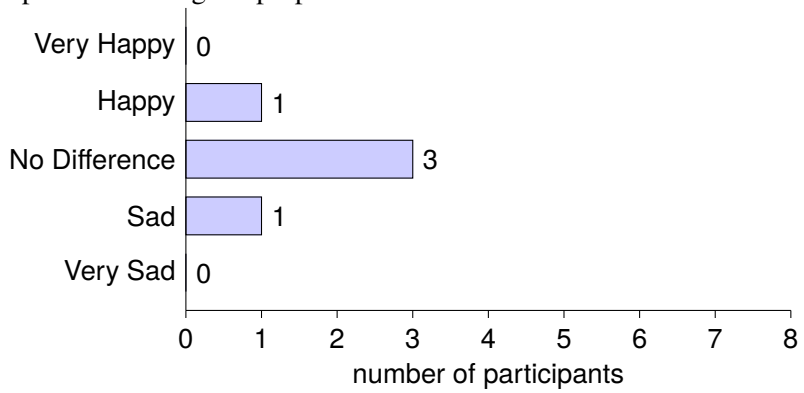
7.1 Final Survey: preCICE

Participants: 5

Q1: Do think the provided solution solves the proposed problem to some extent?
All the participants think that the provided solution will help them solving the problem at hand to some extent.

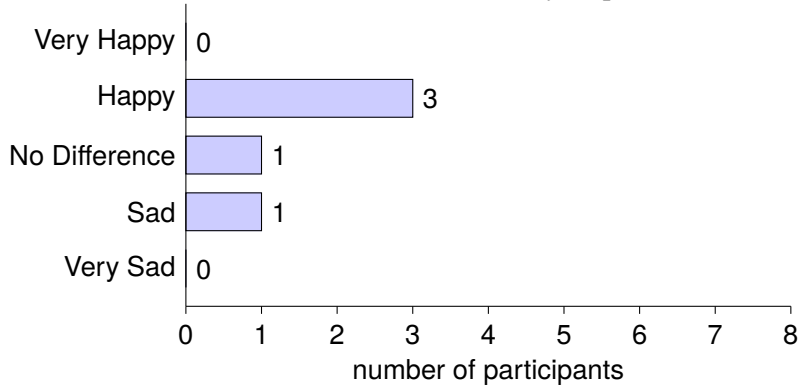
Q2: Did you know of any similar (existing) solution which includes all the elements shown in the demo?
All 5 participants answered as 'No'.

Q3: On what scale will it reduce the mental stress or effort to discover changes within/across repositories using the proposed solution?



Average Score: 3

Q4: On what scale will this centralized observability help to ease the development effort

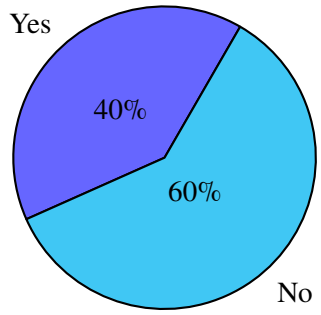


Average Score: 3.4

Q5: What kind of elements do you still miss having on that dashboard?

- More relevant dependencies
- Select 'All' option for GitHub dashboard
- Additional field to display involved people and discussions
- Information about build status of PRs.
- Relation between issues and PRs.
- Links to associated projects and milestones

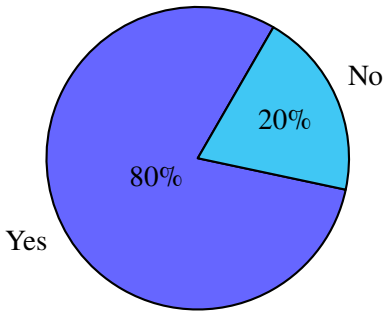
Q6: Is this solution better than any previous tool you were using to keep track of repository activities such as github insights, travis view etc?



Q7: If you answered the previous question as 'no' please express in what way was the other tool better else enter 'NA'.

- GitHub Insights: Less cognitive load.
- The overview of active PRs is a good for catching up with ongoing work.
- Great improvement over manual documentation.
- State of non-preCICE is not available.

Q8: Do you see yourself using this solution(if made available) over traditional systems like GitHub insights for change tracking and observability purpose?



Q9: Any other remarks which could help in improving this solution or anything you found most useful in this dashboard?

- Option to select 'All' repositories on GitHub Dashboard
- Having C++ dependencies
- More relevant columns for GitHub dashboard to reduce cognitive load.
- Clear classification of 'Active' and 'All' fields on the user interface.
- More relevant connections for the dependency graph.
- Advanced filtering options for the dependency graph based on state, activities, scope (internal or external) etc.
- Contributors dashboard to track their activities or identify fading contributions.

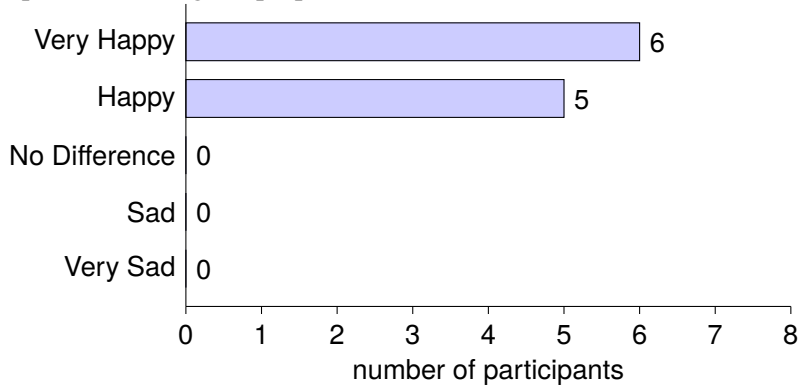
7.2 Final Survey: IBM

Participants: 11

Q1: Do think the provided solution solves the proposed problem to some extent?
All 11 participants think that the provided solution will help them solving the problem at hand to some extent.

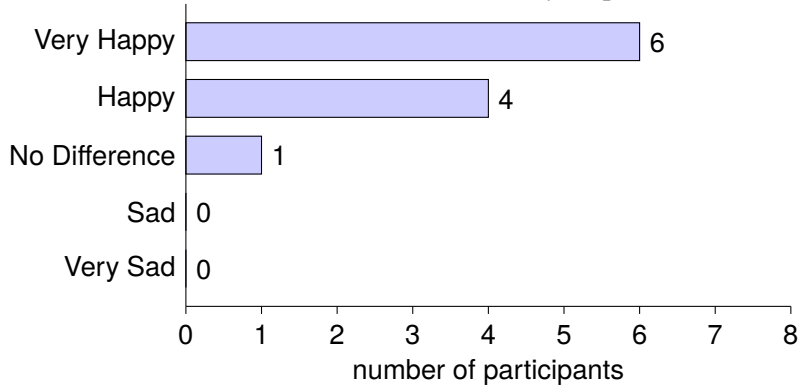
Q2: Did you know of any similar (existing) solution which includes all the elements shown in the demo?
All 11 participants answered as 'No'.

Q3: On what scale will it reduce the mental stress or effort to discover changes within/across repositories using the proposed solution?



Average Score: 4.5

Q4: On what scale will this centralized observability help to ease the development effort

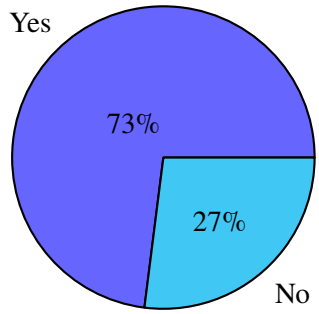


Average Score: 4.5

Q5: What kind of elements do you still miss having on that dashboard?

- Tags/Versions of selected branch.
- Differentiate dependencies by their order.
- Some dependencies are still missing.
- Information about build status of PRs.
- Legend for the dependency graph.

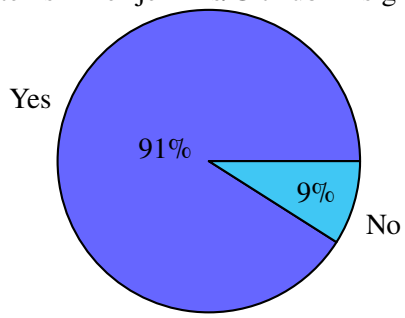
Q6: Is this solution better than any previous tool you were using to keep track of repository activities such as GitHub insights, jenkins view etc?



Q7: If you answered the previous question as 'no' please express in what way was the other tool better else enter 'NA'.

- The dependency graph is extremely helpful to grasp what all is pulled in.
- The overview of active PRs is a good for catching up with ongoing work.
- Great improvement over manual documentation.

Q8: Do you see yourself using this solution(if made available) over traditional systems like jenkins/GitHub insights for change tracking and observability purpose?



Q9: Any other remarks which could help in improving this solution or anything you found most useful in this dashboard?

- Some information about the build versioning.
- Build State of PRs and indication if the same PR failed multiple number of times.
- Some dependencies are still missing.
- This tool is quite good as it is.
- Good observability tool along with other existing SCMs.
- Automatic refresh of dashboard instead of a manual trigger.

7.3 Challenges

The challenges are divided into two parts corresponding to the two use cases of this research topic.

HPVS on VPC

1. **Build state data source**: The data source for gathering the build state in the first iteration was Jenkins REST API. However, they were often slow resulting in timeouts since the query involved more than 250 repositories. This was overcome by implementing two things, changing the data source from Jenkins to GitHub and by using multi-processing to process the requests in parallel.
2. **Missing dependencies**: After the first round of development, it was observed in the feedback that some of the dependencies are still missing from the graph. With some analysis it was clear that some of the dependencies which did not have any proper source URL in their data structure were discarded while creating the tree. It was then fixed using more accurate logic in the Python backend. This logic involved better interpretation of the JSON file while iterating through the dependencies.
3. **Color of legend in graph**: Legends are the meta-data reflecting the data shown in any graph. For creating the dependency graph, Apache ECharts plugin was used in the solution Figure 6.1. This graph had a color mismatch with the associated legend when the theme of the interface is changed from light to dark. This error was reported on the GitHub repository of Apache ECharts and a workaround was then implemented as suggested by the maintainers of this repository[1].

preCICE

1. **Build state data source**: The data source for the build states of preCICE repositories was always GitHub but the APIs on the public GitHub instance was not behaving as expected. It was not returning the proper state of the repository and due to the same there is an ongoing discussion about it on their official community platform[2]. To circumvent this issue, another API was used which also belongs to GitHub. Many repositories have a GitHub action file associated with them, which is referred to as workflow and they also have a status associated with it. These statuses are often used by developers to show as a badge in their README.md files. The status of such workflows are currently being used to indicate the state of a repository.
2. **Grafana Variables**: Grafana variables were used to present a list of repositories on the user interface. Initially, these variables were planned to be multi-select so that the users can pick multiple repositories. This feature is provided through a macro implementation on the GitHub data source plugin. But for the GitHub Dashboard, the multi-select list did not work.

[1] N. Bhawsinka. *Legend doesn't work in dark Mode*. [Online; accessed 7 September 2023]. URL: https://github.com/VolkovLabs/volkovlabs-echarts-panel/issues/203.

[2] supercobra. *GitHub API "combined statusïs always Pending*. [Online; accessed 7 September 2023]. URL: https://github.com/orgs/community/discussions/58407#discussioncomment-6679878.

It queries for incorrect values and hence no data is returned on the interface. Later on, only a single select variable list was used. This issue was also reported on the official GitHub account of GitHub data source plugin[3].

3. **Dependency Graph**: For preCICE, GitHub dependency graph was used as a knowledge base for visualization. GitHub supports certain package ecosystems[4] for identifying the dependencies within a project. This ecosystem also supports C++ but only with NuGet package manager which is a conflict with the C++ repositories in preCICE. Hence, an issue was created on the core library of preCICE to experiment with the configurations of NuGet ecosystem[5].

4. **Deployment Infrastructure**: There were a couple of challenges while dealing with bwCloud. First, the instances were not accessible via Secure Shell (SSH), and it was then observed that it could only be used through the IPV6 address of the instance. Secondly, it was also a challenge to connect the instance with internet. There was some DNS(Domain Name System) lookup configuration which was preventing the instance to ping the IPV6 addresses instead of the IPV4 ones. The learning from that challenge was that the instance supports only IPV6 look-ups for both incoming and outgoing requests[6]. To overcome this issue, there was a modification made in the `/etc/resolv.conf` file[7]. This modification tells the instance to use the given IPV6 address as the nameserver. Lastly, due to the restriction on IPV6 look-ups, once the application was deployed on the instance it could not query the GitHub APIs[8].

[3] yesoreyeram. *Support for multi value variable queries*. [Online; accessed 7 September 2023]. URL: https://github.com/grafana/github-datasource/pull/162#issuecomment-1612999519.

[4] GitHub. *supported package ecosystems*. [Online; accessed 7 September 2023]. URL: https://docs.github.com/en/code-security/supply-chain-security/understanding-your-software-supply-chain/about-the-dependency-graph#supported-package-ecosystems.

[5] N. Bhawsinka. *Dependency Resolution for c++ projects (NuGet)*. [Online; accessed 7 September 2023]. URL: https://github.com/precice/precice/issues/1793.

[6] L. P. (Matthias Leander-Knoll (KIT). *INFO: Situation mit Versorgung von IPv4-Adressen*. [Online; accessed 10 September 2023]. URL: https://www.bw-cloud.org/de/news/2023/11-03-info_ipv4_ka.

[7] AskUbuntu. *systemd-resolved not resolving any domains*. [Online; accessed 1 September 2023]. URL: https://askubuntu.com/questions/1370794/systemd-resolved-not-resolving-any-domains.

[8] P. Spooren. *IPv6 support for cloning Git repositories*. [Online; accessed 10 September 2023]. URL: https://github.com/orgs/community/discussions/10539.

8 Conclusion and Outlook

Multiple versions of a change tracking and observability dashboard has been implemented to ease the development effort in a complex software project. It consists of a Grafana observability dashboard supported by additional information from a Python backend. The final solution was implemented for two use cases which were different in many aspects but united by similar problems. Right from the design process to the solution, there were slight modifications made to adapt the software as per the differences in the use cases. Initially, the solution was designed, implemented, and circulated for the IBM use case and was later tweaked to fit with the preCICE project. These tweaks were minor in comparison to the initial effort which was spent in designing the solution. Throughout the development process, there were many instances where developers of both the teams expressed similar concerns or appreciated similar elements of the dashboard. For instance, they both believed that dependency graph will be extremely useful in analyzing the components of a complex software project. Since the overall solution is open-source, there is an opportunity to enhance it in future with the new features of Grafana dashboard.

To keep the work close to the expectations, surveys were conducted with the developers of the involved projects. The results of these surveys express how different set of developers working for different projects can utilize the solution and even improvise it if needed. Participants of both the surveys were very proactive in trying out the solution and communicating their point of views clearly. The main intention behind keeping some of the survey questions as free text was to let the developers present their ideas. For the first use case of HPVS on VPC (IBM), the developers particularly expressed enthusiasm for the dependency dashboard. They believed that the dependency graph is extremely helpful for them in given scenarios and that it is a huge improvement over traditional approaches such as manual documentation. More than 90 percent of the developers prefer to use this solution over the existing tools as it decreases their development and mental effort. On the other side, for the second use case of preCICE, most of the developers believe that it could be useful to them provided more relevant dependencies are available in the dependency graph. Since a lot of important work in preCICE is close to C++, it will be a good improvement to have them discovered by the GitHub dependency graph[1]. Some further filtering on the GitHub dashboard could possibly reduce the cognitive load and make the content more to the point for the developers. Overall, more than 70 percent of the developers see themselves using this solution.

The slight differences in the opinion of both the use cases may have been due to the difference in the complexity of projects. Furthermore, the dependency management tool used by them makes a huge difference in the final solution because it was the primary data source for the dependency graph. The more accurate the tool is, the more accurate the visualization would be on the dashboard. According to all the developers surveyed, this is a unique tool, and no similar tool is readily available having all

[1] N. Bhawsinka. *Dependency Resolution for c++ projects (NuGet)*. [Online; accessed 15 September 2023]. URL: https://github.com/precice/precice/issues/1793.

the elements included in the solution. And they also agree that the solution will help them in solving their problems to some extent. Grafana also provides an option to further improve the GitHub dashboard just by using the user interface (no coding involved). However, for the dependency dashboard developers might have to make minor changes in the Python script depending on their requirements. In terms of functional requirements, one major observation was that for the preCICE project, a certain type of dependency (C++) was more important than others such as GitHub Actions. Whereas for the other use case, most of the dependencies detected by Renovate are considered equally important.

Finally, the solution aims at re-usability and hence it has been implemented in a modular fashion. To use it for a completely new project, one needs to modify a few configuration files which contain project specific elements such as GitHub URL, organization name etc. The re-usability aspect was once tested while using the same codebase as the IBM use case for preCICE. To make it more resilient, it will further be tested on another IBM project in the future. The end product is distributed in the form of docker images. This facilitates portability and ease of deployment to the maintainers of the solution. From the final survey results in Chapter 7 it can be observed that the implemented solution is useful for the developers and has some scope for improvements. Many of these improvements were already implemented, but a few of them were library bugs and they were reported via GitHub issues to their respective projects. These bugs will be followed-up in future and incorporated in the solution.

The solution has immense potential to adapt to different needs and incorporate new elements in the form of visualizations. Some of the areas of exploration may include incorporating intelligence in the graph based on repository states or addition of alerts based on thresholds defined on the GitHub dashboard. There also have been suggestions to improve the filtering capability of the dependency graph and to incorporate change roll out analysis. This could enhance the overall experience for the developers and allow them to derive relevant information while preparing for a release. Furthermore, the colors of nodes in the dependency graph can be programmed to reflect the age of last change. This was a common suggestion received in both the use cases and will be helpful to determine active or stale repositories. Having the dependency graph on the dashboard opened doors to a lot of innovative ideas which could help teams in managing their software life cycle with more efficiency. The implemented solution offers flexibility, re-usability, and portability along with the functional requirements to the developers working in a complex software development process.

Bibliography

[1] G. Chourdakis, K. Davis, B. Rodenberg, M. Schulte, F. Simonis, B. Uekermann, G. Abrams, H. Bungartz, L. Cheung Yau, I. Desai, K. Eder, R. Hertrich, F. Lindner, A. Rusch, D. Sashko, D. Schneider, A. Totounferoush, D. Volland, P. Vollmer, O. Koseomur. "preCICE v2: A sustainable and user-friendly coupling library [version 2; peer review: 2 approved]". In: *Open Research Europe* 2.51 (2022). DOI: 10.12688/openreseurope.14445.2. URL: https://doi.org/10.12688/openreseurope.14445.2 (cit. on pp. 11, 13).

[2] E. González. *Data visualization with Grafana*. Version 1.0. Oct. 2020. DOI: 10.5281/zenodo.4068095. URL: https://doi.org/10.5281/zenodo.4068095 (cit. on p. 26).

[3] R. Majumdar, R. Jain, S. Barthwal, C. Choudhary. "Source code management using version control system". In: *2017 6th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*. 2017, pp. 278–281. DOI: 10.1109/ICRITO.2017.8342438. URL: https://doi.org/10.1109/ICRITO.2017.8342438 (cit. on pp. 3, 6).

[4] I. Nurgaliev, E. Karavakis, A. Aimar. *Kibana, Grafana and Zeppelin on Monitoring data*. Aug. 2016. DOI: 10.5281/zenodo.61079. URL: https://doi.org/10.5281/zenodo.61079 (cit. on p. 27).

[5] L. Parziale, C. A. D. Leon, J. Y. Girard, C. G. Gomes, F. Schwanzara. *Securing Your Critical Workloads with IBM Hyper Protect Services*. IBM Redbooks, 2022. ISBN: 9780738460338. URL: https://www.redbooks.ibm.com/abstracts/sg248469.html (cit. on p. 12).

[6] T. Saravanan, S. Jha, G. Sabharwal, S. Narayan. "Comparative Analysis of Software Life Cycle Models". In: *2020 2nd International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*. 2020, pp. 906–909. DOI: 10.1109/ICACCCN51052.2020.9362931. URL: https://doi.org/10.1109/ICACCCN51052.2020.9362931 (cit. on p. 4).

[7] M. Stoica, M. Mircea, B. Ghilic-Micu. "Software Development: Agile vs. Traditional". In: *Informatica Economica* 17 (Dec. 2013), pp. 64–76. DOI: 10.12948/issn14531305/17.4.2013.06. URL: https://doi.org/10.12948/issn14531305/17.4.2013.06 (cit. on p. 1).

[8] S. M. Syed-Mohamad, T. McBride. "A Comparison of the Reliability Growth of Open Source and In-House Software". In: *2008 15th Asia-Pacific Software Engineering Conference*. 2008, pp. 229–236. DOI: 10.1109/APSEC.2008.20. URL: https://doi.org/10.1109/APSEC.2008.20 (cit. on p. 11).

[9] E. Yang. "Fuzz testing and software composition analysis in software engineering". In: *2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. 2018, pp. 1–3. DOI: 10.1109/VLSI-DAT.2018.8373240. URL: https://doi.org/10.1109/VLSI-DAT.2018.8373240 (cit. on p. 8).

Declaration

I hereby declare that the work presented in this thesis is entirely my own. I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted hard copies.

---

place, date, signature