Sammy Robens-Paradise
323 East 18th Ave.
Vancouver, BC V5V 1E7

Chair of Department,
Department of Systems Design Engineering

September 21, 2020

Dear Dr. Aultman-Hall,

I have prepared this report, "A Comparison of Software Source Code Control Systems in Modern Software Development" as my 2B Work Report. This report is the first of three that I must submit as part of my degree requirements. It was entirely written by me and has not received any previous academic credit at this or any other institution. This report was written after my 2A term of Systems Design Engineering. It investigates whether a migration from a centralized version control system to a distributed version control system qualitatively and quantitatively resulted in a more efficient software development life-cycle, from the standpoint of a software development team at an upstart Software-As-A-Service company based in Vancouver, British Columbia. This report was inspired by my immediate Supervisor, Rob Kraft, MASc. President of PUMA who asked me to discover and implement software development practices optimizing the development life cycle. This inspired me to investigate the relationship between centralized, and decentralized version control systems and their influence on the software development life-cycle. I would like to thank Rob Kraft, P. Eng., MASc. President of PUMA, Elina Poversky, Sr. Computer Scientist at PUMA and Stephen Leong, Developer at Bayleaf Software for their mentor-ship and participation in this report.

Sammy Robens-Paradise, ID: 20709541, Systems Design Engineering

A Comparison of Software Source Code Control Systems in Modern Software Development

Sammy Robens-Paradise ID: 20709541

September 21, 2020

Systems Design Engineering, 2023
Department of Systems Design Engineering, University of Waterloo

Abstract – The concept of controlling and managing different concurrent versions of a product was not invented by the software industry but was adopted from classical engineering fields which used specs and version numbers allowing engineers to work on separate components of a larger project –such as a plane [1]. Software engineers quickly co-opted the usage of these systems into Source Code Control Systems (SCCS) to store revisions between versions of source code on a centralized server, known as Centralized Version Control Systems (CVCS), and then to Distributed Version Control Systems (DVCS). DVCS allow for changes to the source code to remain locally on a developer's computer, as opposed to on a centralized server. This report compares the relative software development experience between the Centralized Version Control System, Subversion (SVN) hosted on a private server, and Distributed Version Control System Git, hosted on GitHub at PUMA Utility Monitoring (PUMA), a modern Canadian Software-As-A-Service company in Vancouver, BC.

Keywords – Git, SVN, Distributed Version Control Systems, Centralized Versions Systems

I. SITUATION OF CONCERN & PROJECT OBJECTIVES

PUMA was interested in exploring new Source Code Control Systems (SCCS), and migrating from a Centralized Version Control System (CVCS) to a Distributed Version Control System (DVCS) for three reasons: 1) Before the addition of another Software Developer, the PUMA development team consisted of two developers who focused on different application components. The introduction of a third developer caused more code version conflicts. An increase from 10% to 30% over 480 hours of development across 3 developers in revisions with version and code conflicts (merge conflicts) was observed. 2), Developers using PUMA's current CVCS prior to migration did not discuss or review code changes before they were deployed to PUMA's development and production environment. This has been shown to increase the number of post-release failures while using CVC Systems [2]. Lastly 3), the use of a distributed source control system may provide more flexibility for developers by allowing them to apply commits (revisions) locally without sending code changes to a remote server.

The comparison and migration from CVCS to DVCS affected three user classifications. Primary users, were developers who were actively developing code for PUMA and directly used the Source Code Control System. These users were required to maintain a working understanding of the SCCS so that they could efficiently deploy code and collaborate on projects. Secondary users were product managers and owners who did not directly interact with the SCCS but had ownership over the product the SCCS maintained. Tertiary users were investors and external parties who were interested in promoting efficient development practices.

The objective of the project discussed in this report was to determine the most appropriate method of migration from a CVCS to a DCVS. The Situation of concern was to determine from the standpoint of primary users whether a migration from a centralized version control system to a distributed version control system resulted in a more efficient software development life-cycle.

To make such a determination, one quantitative metric, and two qualitative metrics were evaluated. The quantitative metric was the number of development minutes the development team spent working on

source *Code Administration*. This included resolving code conflicts, setting up development environments and product deployment management. The data was collected from timetable billing records and SCCS logs before and after migration. The qualitative metrics were the amount of frustration that primary users felt during the source code migration and whether or not primary users felt that the migration from CVCS to DVCS was beneficial to their workflow. This feedback was collected by conducting virtual meetings with primary users before and after the migration.

These metrics were chosen to understand both the effect on development efficiency and the development experience. The number of minutes of code administration was chosen over a more specific metric because more specific metrics fluctuate drastically based on the feature or project the development team is working on.

A number of assumptions were made when conducting the migration analysis. The first assumption was that the development team would transition from their current development workflow, to what is commonly referred to as the *"Git workflow"* with the DVCS Git and the code management platform GitHub. The workflow prior to migration while using the CVCS Subversion *(SVN)* was what the Apache Working Group refers to as the "Never-Branch-System" [3]. The "Never-Branch-System" works on the basis that developers regularly commit their changes (save them to the remove server/repository) on a main branch, commonly referred to in SVN as */trunk* [3]. The benefits of this workflow are 1) "A low barrier to entry [for developers]" since they do not need to understand or worry about multi-branching [3], and the simplicity of having the most up-to-date code in one central branch. To deploy code, a single developer, with the role of deployment controller would either merge the */trunk* branch into a */production* branch, or manually copy source files over to a */production* branch and commit code to the remote branch. (Figure 1).

PUMA Custom Subversion (SVN) CVCS Workflow

1. Fetch latest code from */trunk* branch so local branch is up-to-date.
2. Apply code changes to working copy.
3. Commit code to remote branch */trunk*, and deploy code to development environment.
4a. If code changes are simple, perform 3-way merge from */trunk* into */production* branch.
4b. If code changes are complex, fetch files to local copy of */production* branch.
  4b.1 Manually apply code changes to local version of */production* branch.
  4b.2 Commit code changes to remote */production* branch.
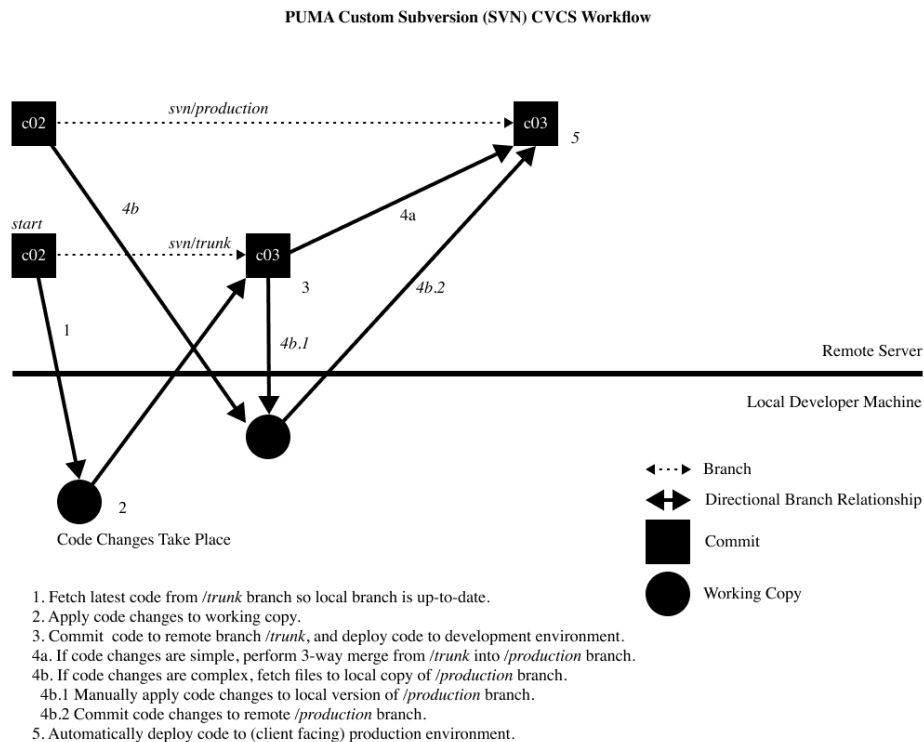5. Automatically deploy code to (client facing) production environment.

Figure 1: PUMA Custom SVN CVCS Workflow [Image source: SRP, 2020]

The drawback of this approach are that if faulty code is committed, it is will break the code base blocking ongoing development.

The Git workflow has been described by GitHub to consist of 6 components in its simplest form. [4] 1) involves the creation of a branch from the current working copy of the code. On the created branch, referred

to as a *feature-branch*, developers make appropriate code changes. 2) Developers locally commit their code changes on the *feature-branch*. 3) developers open a *pull request* against the main branch. 4) developers then collaborate and discuss code changes on the *pull request* and decide if code can be merged into the main branch in its current state. 5) GitHub provides the ability automatically deploy or run automated tests on code on a specific branch. 6) the *feature-branch* is merged into the main branch if the *pull request* is approved (Figure 2).

Proposed PUMA Git & Github DVCS Workflow

1. Fetch latest code from remote */development* branch to local */development* branch.
2. Create */feature-branch* from local */development*, then apply code changes.
3. Add comments to commit.
4. Commit code to local */feature-branch*.
5. Push code to remote */feature-branch* on GitHub server and run automations.
6. Open Pull Request, and merge code.
7. Merge Code from remote */development* branch into remote */production* branch, then build and deploy application.
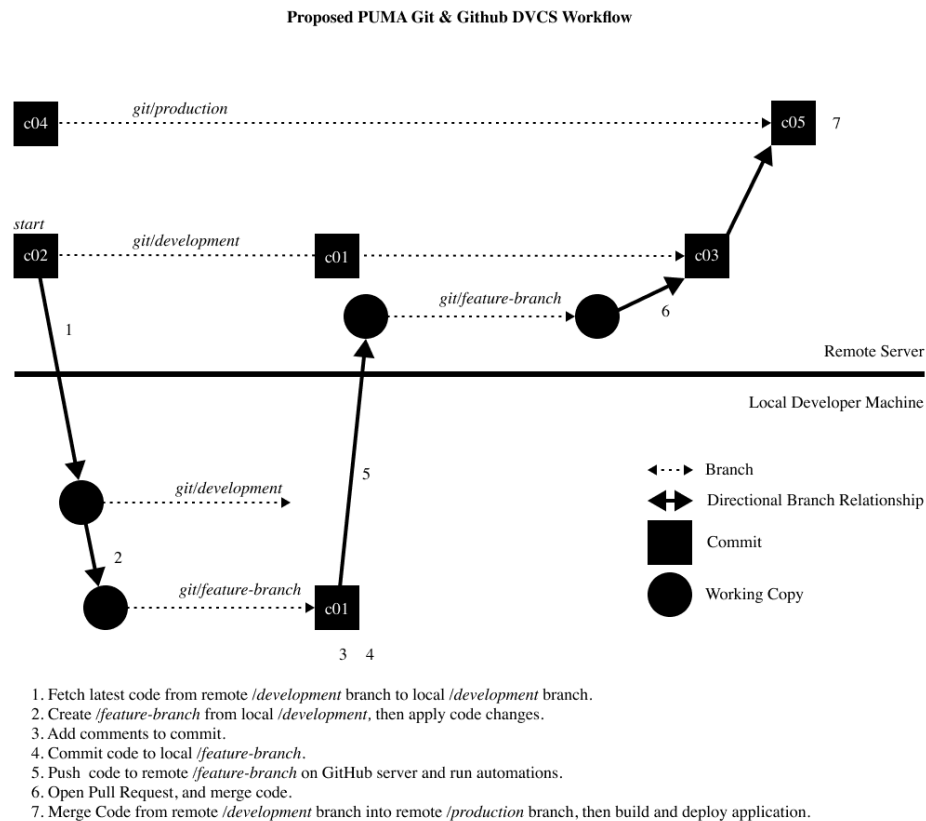
Figure 2: Proposed PUMA Git & GitHub DVCS Workflow [Image source: SRP, 2020]

The second assumption was that the three developers taking part in the project would not change their coding habits. More simply put: if a developer wrote a component of source code in a particular way, post-migration that developer tasked with constructing an identical component, would do it the same way. Without making this assumption, it would be challenging to arrive at particular conclusion about the efficiency resulting from a CVCS to DVCS migration. The third assumption is that there is at least one developer with a working understanding of the DVCS Git participating in the migration–as was the case during this migration.

II. ENGINEERING ANALYSIS

To determine the best method of migration three approaches were evaluated using a computational decision matrix. The methods varied from instantaneous, to staggered migration of the SCCS from CVCS to DVCS, where on one end, a complete migration of all of PUMA's projects would occur at once, and developers would immediately switch over from SVN to Git and GitHub. On the other end of the spectrum each project would be individually migrated and setup on GitHub. In the center, was a clustering approach, where a single—non-essential project would be migrated to GitHub and Git from SVN. Then the remaining projects would be migrated in a cluster once the first project was stable. Each option was given a ranking on a five-point scale of range [-2, +2] where 0 indicated that the impact was negligible, "+" indicated a positive relationship and "-" indicated a negative relationship. Each metric was given a weight range [1,3] where 3 was of the most importance, and 1 the least. Weights were chosen relative to the most important factor which was the risk to ongoing projects (Table 1).

Table 1: SVN to GitHub Computational Decision Matrix [Source: SRP, 2020]

Metric	Low Complexity of method execution.	Low Complexity of code administration.	Low Educational Cost for developers.	High Risk to on-going projects.	Results
Weight	1	2	2	3	
Complete, one-time migration from CVCS to DVCS	+2	+2	-2	-2	-4
Clustered migration from CVCS to DVCS	+1	-1	+2	+1	**+6**
One-at-a-time project migration from CVCS to DVCS	-2	-2	+2	+2	+4

The weighting was multiplied by the relationship and a sum of each relative indicator was taken to determine the best approach where the most desirable approach was the one with the highest score defined by:

$$result = \sum weight \cdot relationship$$

Developers were also asked which of the three approaches they would prefer. There were three developers that would be impacted by the decision. Developer 1 was reluctant to undertake a SCCS migration from CVCS to DVCS because they had become accustomed to a CVCS workflow (Figure 1) and feared they would be less capable as a software developer post migration. They preferred a one-at-a-time migration as a result. Developer 2 had previous experience using both CVCS and DVCS. While they felt more comfortable using Git and GitHub (DVCS), they expressed little preference regarding the migration process. Developer 3 participating in the migration was a proponent of Git and GitHub because their preexisting experience effectively using the DVCS and proposed a complete migration at once time.

Based on the qualitative feedback and input from the three developers, and the results gathered from the SVN to Git and GitHub Computational Decision Matrix (Table 1), the "clustering" approach was chosen as the best methodology to perform the migration from CVCS to DVCS because it allowed developers to migrate over a non-essential application before migrating the remaining projects. Following this migration approach, the application PUMA.Admin was chosen to migrate first because it is an internal-facing application and had little ongoing development. Once PUMA.Admin was successfully migrated and was shown to be stable,

the rest of the project applications were migrated together in a cluster.

This approach caused minimal disruption to ongoing software development, low risk for user-facing products, a decreased opportunity for source code corruption, and a window for developers to explore Git and GitHub before they were fully immersed in the DVCS.

Migrating a project from Subversion (SVN) to Git and GitHub had been done before but presented a number of challenges that needed to be handled during the migration. Subversion (SVN) and Git and GitHub store commit history differently, and Git stores more information with each commit. There are two significant differences in the way information is stored between Subversion (SVN) and Git (Table 2) [5, 6, 7].

Table 2: SVN, Git & GitHub Comparison [Source: SRP, 2020]

Difference	Branches	Commits
Git and GitHub	Git and GitHub enforce a strict branching structure, where each branch contains a snapshot of an entire repository. This means that a given branch, is simply just an instance of a central repository that has been pulled off of a main parent branch.[5] Changes to the current branch are tracked by Git over the entire repository, all within the same directory. This means that a developer can switch branches, effectively changing development contexts without leaving a repository and/or directory.	Git captures information such as user emails in each commit and does not need to store in a linear model. For example, If a branch with diverging code is merged into a main branch such as master then Git will reapply changes as commits to the master branch [5]. Git also only stores change-sets between code commits, giving it the ability to better determine and automatically resolve merge conflicts between two branches. Git commits also contain a hash that can be thought of as a pointer to a specific instance of the code on a branch, rather than a particular copy of the code.
Subversion (SVN)	SVN has a large degree of freedom in relation to branching. It does not enforce any particular branch structure, and in fact, the idea of branching in svn is more of an "agreed upon" concept rather than a core feature of the SCCS [7]. SVN tracks directories and files, rather than branches. meaning a particular branch is just a named directory in a central repository. Furthermore, SVN will only apply changes to the current directory.	SVN commits occur in linear order, meaning code changes cannot be re-applied to a base branch but are to be interpreted as revisions, which can be thought of as a snapshot of an entire file system and its files at a particular moment in time. Since SVN does not track code changes, merging one branch into another simply means applying a particular snapshot of a file system to another file system. This is much harder for SVN to reconcile, and often results in vast amounts of "merge conflicts". SVN also does not store author emails with each commit.

Git provides the command-line tool *git-svn* that allows "Bidirectional operation between a Subversion repository and Git" that was used to automate the migration [8]. because Git and SVN store commit history differently, *git-svn* works by recursively looping through an SVN repository's commit history then each SVN branch/directory and checking out each revision [8]. Each revision is then committed to a new Git repository and matched to a Git branch. A shell script was written to automate this process shown below in simplified form. It was executed for each software project that PUMA maintained:

```sh
#!/bin/sh
# get URL, directory path, and username as user input
# checkout svn repository
svn checkout $url
[ -d migration ] mkdir migration
cd "$name"
# make user file to match emails to usernames
```

```
touch users.txt
svn log --xml --quiet | grep author | sort -u |
perl -pe 's/.*>(.*?)<.*/$1 = /' >users.txt
read -p
# make changes to file and match username to emails
vi users.txt
# clone and recursively apply commits to Git repository
git svn clone $url --authors-file=users.txt --no-metadata --prefix "" -s ../migration/$name
cd ../migration/$name
# cleanup branch names and tags
for t in $(Git for-each-ref --format='%(refname:short)' refs/remotes/tags); do Git tag
    ${t/tags\//} $t && Git branch -D -r $t; done
for b in $(Git for-each-ref --format='%(refname:short)' refs/remotes); do Git branch $b
    refs/remotes/$b && Git branch -D -r $b; done
for p in $(Git for-each-ref --format='%(refname:short)' | grep @); do Git branch -D $p; done
Git branch -d trunk
# finish
Done
```

The process could not be fully automated because Git stores a user's email and SVN does not. Thus, a relationship between a user's name in SVN and an email had to be defined. This was done in a text file automatically generated by the Shell script, however an email associated with each user who committed to the SVN repository had to be manually entered. The complete analysis process and migration design involved four steps. The first being the collection of qualitative feedback from primary users as well as the collection of a qualitative metric: source Code Administration in tabular format by day and developer. Two, the first project was migrated over to Git and GitHub from SVN and was connected to a continuous deployment pipeline. Three, the remaining projects were migrated in a cluster from SVN to Git and GitHub and connected to a continuous deployment pipeline. Step four, qualitative and quantitative data was (once again) collected and the results were analyzed.

III. RESULTS

The following tables outline the results that were collected before and after the design change from a CVCS SVN server to a DVCS Git version control system hosted and maintained by GitHub:
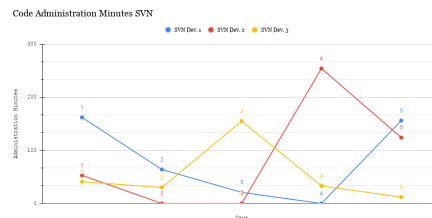

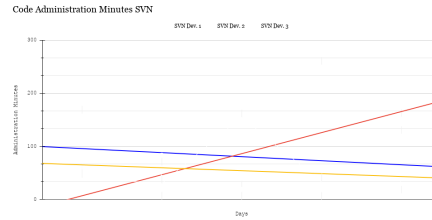Figure 3: Pre-Migration SVN Code Administration [Image source: SRP, 2020]

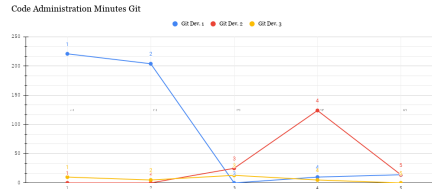Figure 4: Pre-Migration SVN Code Administration Trend [Image source: SRP, 2020]

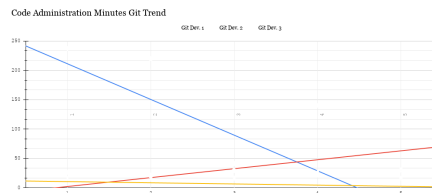Figure 5: Post-Migration Git Code Administration Minutes Chart [Image source: SRP, 2020]

Figure 6: Post-Migration Git Code Administration Minutes Trend [Image source: SRP, 2020]

In comparing the Code Administration metric it was observed that on average the number of Code Administration minutes decreased between the pre—and post-migration from SVN to Git. Previously the average time spent interacting with the SCCS was 368 minutes per 40-hour work week relative to the time spent interacting with the SCCS post migration, which was 215 minutes, therefore a decrease of 42% in time spent. Additionally the average slope of trend for the number of minutes spent conducting code administration using CVCS was -2.71, relative to the average slope of trend for the number of minutes spent conducting code administration using DVCS which was -20.3. Quantitative results above are supplemented by two qualitative metric results summarized in Table 3:

Table 3: Qualitative User Feedback Pre, and Post Migration from. CVCS to DVCS [Source: SRP, 2020]

Metric	CVCS Pre Migration Feedback	During Migration Feedback	DVCS Post Migration Feedback
Question	Is the current SSCS system beneficial to your workflow?	Do you feel frustrated with the migration from a CVCS to a DVCS?	Is the current SSCS system beneficial to your workflow?
Developer 1: No Experience with Git	No Opinion. [They] feel as though it is beneficial because it is what is comfortable	No. But overwhelmed by transition and [the] need to learn new technology.	Speculative Yes. Stated that they continue to feel more comfortable with each system interaction.
Developer 2: Experience with Git	No. Recognized that when working with multiple developers probability of merge conflicts increases.	No. Worried about the ability of other developers to maintain their workflow post migration to DVCS.	Yes. User believes that DVCS brings added benefit to a collaborative code workflow.
Developer 3: Experience with Git	No. Feels as though CVCS is not beneficial to workflow	No. Worried about the ability of other developers to maintain their workflow post migration to DVCS.	Yes. Satisfied with the ability that [the] Git Flow provides to collaborate with other developers.

IV. DISCUSSION

Data collected before, and after migration from a CVCS to a DVCS showed a decrease in the average number of minutes developers spent conducting code administration. Furthermore, the average trend of code administration post-migration was about approximately 7.5 times smaller. The post-migration data was collected in a 40-hour work week immediately following the migration. Using multiple trace theory which implies that humans will improve at a task the more they perform it, it may be induced that as developers continue to use Git their task recall will improve and so will their efficiency [9]. As a result, the data collected indicate that a migration from CVCS using Subversion (SVN) to a DVCS based on Git and GitHub using the standard GitHub Workflow (Figure 2) may have a causal link to a decrease in time spent conducting code administration tasks for the PUMA team resulting in a more effective development experience.

This observation could be explained for two reasons: The first pertains to the development environment. Under the CVCS SVN model, directories are treated as "branches". This means that for each branch, a developer needed to instantiate a development environment. This includes adjusting operating system permissions, adding configuration files and configuring Internet Information Services (IIS) for local development. This can take a significant amount of time. It was also the reason why the development team at PUMA adhered to the "Never-Branch-System" which had its own drawbacks [3]. Under the DVCS Git model, developers could change branches from within one directory and development environment, completely eliminating the need to setup a development environment for each branch. This saved a substantial amount of time. The second possible explanation relates to the difference in the way SVN and Git handle merge conflicts. SVN attempts to reconcile an entire line-by-line file change, causing often "erroneous" merge conflicts such as a change in white-space. This meant that developers spent valuable time resolving mundane merge conflicts. Git on the other hand, tracks change-sets (Table 2). This meant that developers needed only intervene when there were significant or substantiated merge conflicts such as the change of a code path. This greatly reduced the amount of time developers spent on code administration tasks.

Another interesting observation is that Developer 1 stated they were still not completely comfortable with the DVCS model post-migration. This can be observed in Figure 6, where on the first day of the data collection week Developer 1 spent 221 minutes (49%) of their 450 minute workday handling code administration tasks. As they got more comfortable however, this daily number quickly decreased later in the data collection period (Figure 5 and Figure 6).

V. LIMITATIONS OF METHODS USED

Because the developers participating in the migration took on multiple roles during the software development life-cycle, it was hard to get a precise picture of the amount of time that developers truly spent conducing code administration in the average week, since each week was somewhat different. The consequence of this is that it was challenging to conduct a true comparative analysis from before and after the migration from a CVCS to a DVCS because no development week is exactly the same. This could cause skewed results meaning that while the conclusion may be accurate for a particular development week, it cannot be generalized to the entire software development life-cycle. While qualitative assessment of primary users provided some mitigation, it is not a substitute for quantitative results. Another limitation on the methods used is the presence of implicit bias in the analysis. It must be noted that the author proposed a CVCS (SVN) to DVCS (Git) migration, participated in the migration (Developer 3) as well as analyzed and collected qualitative and quantitative data concerning the migration. Because analysis was conducted by a participant, implicit bias must be considered in the report.

VI. CONCLUSIONS

Based on qualitative and quantitative data collected, the average time developers 1,2 and 3, spent conducting code administration decreased by 42% pre, versus post migration from a CVCS to a DVCS. This indicated that the usage of the DVCS Git over the CVCS SVN may result in a more efficient software development life cycle across PUMA's projects because developers were able to spend more time writing code, than managing it. It can also be concluded that while there was a negative trend in time spent conducting code administration, the efficiency of developers with an SCCS is influenced heavily by their conceptual understanding of the logic and methodology behind the given version control system, as well as the amount of time developers have spent interacting with the SCCS.

VII. RECOMMENDATIONS

Recommendation: Software development teams with more than two developers working on Unicode-based languages and currently using SVN as their SCCS should consider use the clustering approach discussed in this report to migrate to Git and GitHub.

Rationale: Qualitative and quantitative analysis conducted as part of this report indicates that Git provides resources linked to an increase in effectiveness of the software development life-cycle.

Costs: With the assumption that there is at least one developer with a working understanding of DVCS and the skill set to perform migrations, it is estimated, based on the timelines observed during this migration that it would take a single developer approximately 60 developer-hours to conduct project migrations using the "clustering migration" approach using an automated script. Ideally, it would take two capable developers 30 hours each to conduct the same migration. Using the median hourly rate of software developers in Canada of $25.01 CAD/Hour in 2020 [10], and the GitHub's current price-per-user/month of $ 4.00 (September 2020), an estimated cost would be calculated by:

$$\$cost \approx \$25.01 \cdot (60 + hours_{additional}) + \$4 \cdot number_{developers} \cdot n_{months}$$

Benefits: There are two reasons why teams should undergo a migration from SVN to Git. First, Git has been shown to produce less merge conflicts, and the ability to locally commit code as well as collaborate on "pull requests" was qualitatively linked to a more desirable development experience. Secondly, assuming the average hourly rate for a software developer in Canada to be $25.01 CAD/Hour [10], developers would save a median of 51 minutes of development time per week based on the data collected from PUMA's migration. This translates to annual ideal savings of $1065 CAD/year, compared to the relative average cost of Git, GitHub and the migration at $508 CAD/year. This can be thought of as additional 22 hours of development time per developer per year. For developers earning more, savings would increase since the cost of GitHub

is independent of a developer's salary.

ACKNOWLEDGMENTS

References

[1] W. Carstensen and W. Carstensen, "A brief history of version control - Redgate Software", *Redgate Software*, 2016. [Online]. Available: https://www.red-gate.com/blog/database-devops/history-of-version-control. [Accessed: 13- Sep- 2020].

[2] Albert Einstein. C. Bird, N. Nagappan, P. Devanbu, H. Gall and B. Murphy, "Does distributed development affect software quality?", *Communications of the ACM*, vol. 52, no. 8, pp. 85-93, 2009. Available: 10.1145/1536616.1536639.

[3] "Subversion Best Practices", *Svn.apache.org*, 2017. [Online]. Available: https://svn.apache.org/repos/asf/subversion/trunk/doc/user/svn-best-practices.html. [Accessed: 13- Sep- 2020].

[4] "Understanding the GitHub flow · GitHub Guides", *Guides.github.com*, 2020. [Online]. Available: https://guides.Github.com/introduction/flow/. [Accessed: 13- Sep- 2020].

[5] "Git vs. SVN – What Is The Difference? — Perforce Software", *Perforce Software*, 2020. [Online]. Available: https://www.perforce.com/blog/vcs/Git-vs-svn-what-difference. [Accessed: 13- Sep- 2020].

[6] B. Collins-Sussman, B. Fitzpatrick and M. Pilato, "Examining History", *Svn.gnu.org.ua*, 2006. [Online]. Available: http://svn.gnu.org.ua/svnbook/svn.tour.history.html. [Accessed: 13- Sep- 2020].

[7] B. O'Sullivan, "Making Sense of Revision-control Systems", *Dl.acm.org*, 2020. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/1594204.1595636. [Accessed: 13- Sep- 2020].

[8] S. Chacon and J. Long, "Git - Reference", *Git-scm.com*, 2020. [Online]. Available: https://Git-scm.com/docs/Git-svn. [Accessed: 13- Sep- 2020].

[9] M. Moscovitch et al., "Functional neuroanatomy of remote episodic, semantic and spatial memory: a unified account based on multiple trace theory", *Journal of Anatomy*, vol. 207, no. 1, pp. 35-66, 2005. Available: 10.1111/j.1469-7580.2005.00421.x.

[10] "Software Developer Salary in Canada — PayScale", *Payscale.com*, 2020. [Online]. Available: https://www.payscale.com/research/CA/Job=Software_Developer/Salary. [Accessed: 13- Sep- 2020].

Images, tables and graphs presented in this document created by the authors are indicated by author initials and the year of creation.