Shane McIntosh
Research Statement

17 Van Order Dr, Unit 9-301, Kingston, ON, Canada : (613) 876-1243
✉: shanemcintosh@acm.org : http://shanemcintosh.org/

Modern software is developed at a breakneck pace. While the software releases of the past would take several months or even years to prepare, modern software organizations like Google, LinkedIn, and Facebook release several times daily [1]. Furthermore, it is not uncommon for large software systems like Mozilla to receive hundreds of change requests (e.g., defect reports, feature requests) on a daily basis [2]. After these change requests have been triaged to the appropriate team members, developers update the codebase to implement the change requests. In August 2014, the codebase of Mozilla was updated 13,090 times – an average of 422 times per day.[1]

To cope with the rapid rate at which modern software changes, software organizations dedicate personnel to the task of developing and maintaining tools that automate the software release process. For example, if any of the hundreds of daily code changes cause an error in the *build process*, i.e., the automated steps performed to translate source code into deliverables like executables, team members should be notified immediately so that reactionary measures can be taken. To provide this rapid feedback loop, software organizations adopt techniques like *continuous integration* that routinely download the latest source code changes onto dedicated servers to ensure that they are free of compilation and test failures. Moreover, techniques like *continuous delivery* hasten the tempo at which official releases can be delivered to users by automatically packaging and deploying software changes that satisfy automated testing criteria.

The rapid release cycle of modern software generates new challenges for software organizations. My research focuses on how such challenges can be addressed. In my estimation, my largest contributions in this respect have had a lasting impact on industrial release processes and academic SE research:

- **Modernization of an industrial software build system** – At the heart of the rapid release cycle of modern software is the *build system*, i.e., the system that specifies how source code is translated into deliverables. An efficient build system that quickly produces updated versions of a software system has become critical infrastructure that organizations require in order to keep up with market competitors. At the EMC^2 Corporation (one of the world's leading data storage providers), I spearheaded an effort to modernize an aging software build system comprised of more than 200,000 lines of build logic. The newly designed build system is now used by a globally-distributed development team, producing hundreds of builds daily. Since the new build system implements a more reliable version of the old build process, it can leverage multi-core processors, and thus complete builds in as little as one-tenth of the time required by the old build system.
- **Assessing the quality of software build systems** – During the build modernization project at EMC^2, I noticed that many of the problems were encountered due to duplication of build logic (i.e., cloning). Industrial researchers at CQSE in Germany had also noticed a similar trend during numerous software quality assessments. Hence, together with Avaya Labs Research, we collected a large benchmark of build logic clones to extend CQSE's suite of software quality assessment tools to include measures of build system quality. Detailed analysis of the benchmark has also been used to guide organizations like Munich Re (one of the world's leading reinsurers) in improving the quality of their build systems.

Broadly speaking, I perform empirical studies that mine historical data generated during the development of modern software systems. Specifically, my dissertation work, which was supported by the only *Vanier Canada Graduate Scholarship* (Canada's top PhD scholarship) to be awarded in the field of software engineering, focuses on release engineering with an emphasis on software build systems. My more recent work, in collaboration with academics in Japan, focuses on software quality with an emphasis on peer code review practices. The following sections discuss my findings and my vision for future research in each area.

Release Engineering

Source code changes must be integrated into an official release in order to be made available to users. The process of integrating code changes into official software releases and automating the build process

[1] http://relengofthenerds.blogspot.ca/2014/09/mozilla-pushes-august-2014.html

is referred to as *release engineering*. Despite the importance of release engineering in practice, it has garnered little research attention. In my dissertation, I focus on one aspect of release engineering – the software development overhead generated by the maintenance and execution of build systems.

Current Research: **Studying the software development overhead introduced by the build system.** While an efficient build system is critical infrastructure that software organizations rely upon, the benefits that they provide come at a cost – build systems introduce overhead on the software development process.

Maintenance overhead is generated by the need to keep the build system in sync with other software artifacts, such as the source code and testing infrastructure. Indeed, my research has shown that, from release to release, source code and build system tend to *co-evolve* [8, 9], i.e., changes to the source code can induce changes in the build system, and vice versa. This work was nominated for best paper at the 7th Working Conference on Mining Software Repositories. Furthermore, in work published at the International Conference on Software Engineering (ICSE, the flagship conference of the SE community), we have shown that up to 27% of code changes and 44% of test changes are accompanied by build changes [12].

These were the first studies to systematically explore the impact that build maintenance has on software organizations in an empirical fashion. Prior to that, little research had focused on the build system. Since the publication of these studies, these papers have generated 71 citations.[2] Moreover, in recent years, top SE conferences like ICSE have received enough high quality papers on the topic of build systems to dedicate sessions to the problem. In addition to inspiring other recent studies, these early papers lay the groundwork for my continued work on build maintenance:

- **Automatically identifying code changes that require accompanying build changes**. The maintenance overhead introduced by the build system is exacerbated by the difficulty of identifying the code changes that require accompanying build changes. Seo *et al.* show that 30%-37% of the builds triggered by Google developers on their local copies of the source code are broken, with neglected build maintenance being the most commonly detected root cause [7]. If those build breakages are not fixed before the changes are committed to upstream repositories, then the breakage will impact the entire team. Broken builds prevent quality assurance teams from reproducing and testing actively developed versions of a system in a timely fashion, slowing development progress and the release process. Hence, we set out to better understand when build changes are necessary [10]. We trained classifiers capable of identifying the code changes that require accompanying build changes. These classifiers accurately identify up to 63% of the rarely occurring source-build co-changes, more than doubling the performance of naive classifiers like ZeroR.
- **Using a large benchmark of build systems to compare build technologies**. There are dozens of build technologies available for developers to select from, each with its own nuances. However, the impact that technology choice has on build maintenance is not clear. Hence, we set out to study the impact that technology choice can have on a variety of build maintenance factors, such as: (1) the rate of build change and its coupling with the source code [15], and (2) duplication of build logic [16]. We find that although modern technologies like Maven provide additional features that older technologies do not provide, they tend to change more often, inducing more churn than lower level technologies like Ant. Furthermore, the modern technologies tend to be more tightly coupled to source code changes and contain more duplication of build logic than lower level technologies.

Execution overhead is introduced by the slow nature of using the build system to (re)generate system deliverables. Since large software systems are made up of thousands of files and millions of lines of code, the execution of the build system can be prohibitively expensive, often taking hours, if not days to complete. For example, Windows builds of the Firefox web browser take more than 2.5 hours on dedicated build machines.[3] Certification builds of a large IBM system take more than 24 hours to complete [5]. Indeed, while developers wait for build tools to synchronize source code with deliverables, they are effectively idle.

- **Automatically identifying build hotspots**. While files that rebuild slowly can slow developers down, the impact is limited if those files only rarely require change. This suggests that traditional build profiling techniques may miss the files that would really make a difference in day-to-day development. Instead, I assert that build optimization effort should be focused on *build hotspots*, i.e., files that not only take a substantial amount of time to rebuild, but also require frequent maintenance, and thus

[2] http://scholar.google.ca/citations?user=FxUqGoUAAAAJ&hl=en

[3] http://tbpl.mozilla.org/

generate considerable overhead on the build process. In recent work, we design an approach to detect build hotspots by analyzing the build system and extracting the historical change tendencies. Through simulation, we show that focusing optimization effort on build hotspots will deliver more rebuild cost savings than focusing on files with the highest rebuild cost, rate of change, or impact on other files [11].

- **Understanding the characteristics of build hotspots**. While the rate of change of a file is required to detect build hotspots, it cannot be avoided in order to improve build performance. Instead, build optimization effort must focus on controllable properties that influence the likelihood of a file becoming a build hotspot. Hence, we build regression models capable of identifying build hotspots using code properties. Our models can explain 32%-57% of the identified hotspots. Furthermore, code layout properties provide more of the explanatory power in the larger studied systems than code content properties do. This suggests that as systems grow, build hotspots have more to do with how code is organized than with the code contained in a file itself.

This work has piqued the interest of a variety of open source communities. Indeed, I was invited to present my findings to the PostgreSQL open source community at the Free and Open Source Developers European Meeting (FOSDEM 2014). Moreover, after delivering my FOSDEM 2014 presentation, I was contacted by the Qt open source community, who have used our technique to improve their build performance.

Research vision: **Enhancing software analyses using build data**. While my prior work has focused on the various ways that the build system introduces overhead on the software development process, the build system contains plenty of useful information that researchers can leverage to complement analyses of software systems. For example, in recent work, we mine traces of build executions to identify potential software licensing violations [17]. The potential violations identified by our approach are of immense practical value, generating rapid reactionary measures in several open source systems.

I believe that the data contained in the build system can also be used to enhance other types of software analysis. For example, impact analysis of defects is of growing importance in the field of software defect prediction. The build system can be used to aid in impact analysis by not only identifying the deliverables that are impacted by a defect, but also by identifying the impacted configurations of the software (e.g., Windows vs. Mac OS X). The intuition behind this approach would be that a defect that impacts many deliverables in many configurations has a higher impact than one that impacts few deliverables and configurations.

Further improvements to build speeds. As the speed of computer processors stagnate, developers rely more and more on parallel processing in order to optimize software system performance. The same is true for build processes, which increasingly rely on distributed and parallel build architectures in order to improve build performance. Yet the speedup achieved by parallel build processes is limited by the dependency structure of the system being constructed. Thus, files in the build process may act like bottlenecks that slow the build process down. In future work, I plan to analyze the graph of dependencies described by the build system in order to identify bottlenecks that prevent parallel builds from achieving larger speedups.

Software Quality

Current Research: **Understanding the factors that impact software quality**. In the modern software market with shorter release cycles, ensuring that software systems are of sufficient quality for public release has a become a more time- and resource-constrained phase of the software development process. Hence, it is of critical importance that the limited quality assurance resources of a software organization are focused on the factors that truly impact software quality. In my recent research, I have focused on gleaning information from machine learning classifiers and regression models of software quality:

- **Cross-project models of the risk of software changes**. In recent work, we use machine learning techniques to train classifiers that identify high-risk code changes, and test them in a cross-project contexts [4, 6]. We find that the within-project performance of a classifier is not a strong indicator of its performance in a cross-project context. Instead, classifiers trained using data from several projects or ensembles of classifiers trained on individual projects tend to perform well, and can even outperform within-project classifiers. This work was nominated for best paper at the 11^{th} Working Conference on Mining Software Repositories. In future work, we plan to analyze the impact that code change metrics have on the risk of introducing defects in these cross-project models.
- **The impact of modern code review practices on software quality**. Code review, i.e., the practice of having other team members critique changes to a software system, has long been a well-established

best practice in software development [3]. In the past, a rigourous formal code inspection process was proposed [3], where a base level of review quality was ensured by encouraging developers to follow checklists and conduct script-based inspection meetings. However, in the modern, globally distributed, and rapidly releasing development environment, such a heavyweight code review process is impractical. Instead, modern software organizations adopt a lightweight, tool-supported code review process. In recent work, we apply rigourous statistical analysis to identify properties of the modern code review process that have a measurable impact on long-term software quality [13, 14]. We find that while the coverage of the review process occasionally shares a link with software quality, estimates of review participation (e.g., discussion length, reviewing timeframe) and reviewer expertise share a more consistent link with software quality. This implies that organization policies that focus on ensuring that a review has happened will not yield optimal results. Instead, organizations should consider review participation and expertise when making integration decisions. This work won the distinguished paper award at the 11th Working Conference on Mining Software Repositories.

Research Vision: **Revisiting approximations of software quality in rapidly releasing software systems**. Software quality is typically modelled using *post-release defects*, i.e., defects that escape the development process and are only noticed in official software releases. The intuition behind this approximation of software quality is natural – any post-release defect has an implicit negative impact on the customer-perceived quality of a software system. I conjecture that such an approximation of software quality is insufficient for rapidly releasing software systems. While the post-release defects of the past would linger for months or even years as new releases were slowly prepared, in a rapid release cycle, such defects may be addressed almost immediately, and an update can be issued within minutes.

I believe that a shift in mindset is required to better understand how software quality can be approximated in rapidly releasing systems. First, qualitative research is required to better understand the software quality challenges encountered by organizations that rapidly release software. Such research would yield an industry-grounded definition of software quality in rapidly releasing software systems. Based on this definition, more appropriate approximations of software quality can be derived, which will likely lead to software quality models that yield more practical value for organizations that operate under a rapid release cycle.

References

[1] B. Adams, C. Bird, F. Khomh, and K. Moir. 1st International Workshop on Release Engineering. In *Proc. of the 35th Int'l Conf. on Software Engineering (ICSE)*, pages 1545–1546, 2013.

[2] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an Open Bug Repository. In *Proc. of the OOPSLA workshop on Eclipse technology eXchange*, pages 35–39, 2005.

[3] M. E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 15(3):182–211, 1976.

[4] T. Fukushima, Y. Kamei, Shane McIntosh, K. Yamashita, and N. Ubayashi. An Empirical Study of Just-in-Time Defect Prediction using Cross-Project Models. In *Proc. of the 11th Working Conf. on Mining Software Repositories (MSR)*, pages 172–181, 2014. **Nominated for best paper**.

[5] A. E. Hassan and K. Zhang. Using Decision Trees to Predict the Certification Result of a Build. In *Proc. of the 21st Int'l Conf. on Automated Software Engineering (ASE)*, pages 189–198, 2006.

[6] Y. Kamei, T. Fukushima, Shane McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan. Studying Just-In-Time Defect Prediction using Cross-Project Models. *Empirical Software Engineering*, Under review, 28 pages.

[7] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge. Programmers' Build Errors: A Case Study (at Google). In *Proc. of the 36th Int'l Conf. on Software Engineering (ICSE)*, pages 724–734, 2014.

[8] Shane McIntosh, B. Adams, and A. E. Hassan. The Evolution of ANT Build Systems. In *Proc. of the 7th Working Conf. on Mining Software Repositories (MSR)*, pages 42–51, 2010. **Nominated for best paper**.

[9] Shane McIntosh, B. Adams, and A. E. Hassan. The evolution of Java build systems. *Empirical Software Engineering*, 17(4-5):578–608, August 2012.

[10] Shane McIntosh, B. Adams, M. Nagappan, and A. E. Hassan. Mining Co-Change Information to Understand when Build Changes are Necessary. In *Proc. of the 30th Int'l Conf. on Software Maintenance and Evolution (ICSME)*, pages 241–250, 2014.

[11] Shane McIntosh, B. Adams, M. Nagappan, and A. E. Hassan. Identifying and Understanding Header File Hotspots in C/C++ Build Processes. *Automated Software Engineering*, Under review, 27 pages.

[12] Shane McIntosh, B. Adams, T. H. D. Nguyen, Y. Kamei, and A. E. Hassan. An Empirical Study of Build Maintenance Effort. In *Proc. of the 33rd Int'l Conf. on Software Engineering (ICSE)*, pages 141–150, 2011.

[13] Shane McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the QT, VTK, and ITK Projects. In *Proc. of the 11th Working Conf. on Mining Software Repositories (MSR)*, pages 192–201, 2014. **Distinguished paper award**.

[14] Shane McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. An Empirical Study of the Impact of Modern Code Review Practices on Software Quality. *Empirical Software Engineering*, Under review, 40 pages.

[15] Shane McIntosh, M. Nagappan, B. Adams, A. Mockus, and A. E. Hassan. A Large-Scale Empirical Study of the Relationship between Build Technology and Build Maintenance. *Empirical Software Engineering*, To appear, 2014.

[16] Shane McIntosh, M. Poehlmann, E. Juergens, A. Mockus, B. Adams, A. E. Hassan, B. Haupt, and C. Wagner. Collecting and Leveraging a Benchmark of Build System Clones to Aid in Quality Assessments. In *Proc. of the 36th Int'l Conf. on Software Engineering (ICSE)*, pages 115–124, 2014.

[17] S. van der Burg, E. Dolstra, Shane McIntosh, J. Davies, D. M. German, and A. Hemel. Tracing Software Build Processes to Uncover License Compliance Inconsistencies. In *Proc. of the 29th Int'l Conf. on Automated Software Engineering (ASE)*, pages 731–741, 2014.