Continuous practices and technical debt: a systematic literature review

Bjørn Arild Lunde, Ricardo Colomo-Palacios
Department of Computer Science
Østfold University College
1783 Halden, Norway
bjorn.a.lunde@hiof.no; ricardo.colomo-palacios@hiof.no

***Abstract*—Technical debt in software development is a common problem that is overlooked by many development teams. This debt can be generated from a variety of reasons, including time pressure and complexity in software. Technical debt in simple terms is when a simple and less optimized solution is carried out in order to gain short term benefits, which leads to refactoring and reworking code later on, costing both time and money. The issue is present in both big, established companies and small startups, and is the reason why many of these small startups never get enough economic grip before debt catch up and they go bankrupt. This paper aims to address this problem by exploring how continuous practices including DevOps could help resolve this issue by adopting the right approaches into the software development cycle and workflow. So as to collect information about these topics, a systematic literature review has been conducted, covering both positive and negative impacts these practices can have on technical debt. The findings will present the current practices used to manage and reduce the accumulation of technical debt, if and how these approaches can be used to reduce already existing technical debt and which of these practices that have the biggest impact on technical debt. The paper concludes that there's potential for continuous practices including DevOps to possibly reduce technical debt if applied appropriately**

Keywords- Technical debt, continuous integration, continuous refactoring, continuous delivery, continuous release, continuous deployment

I. INTRODUCTION

There are many activities required in a software development process, and unfortunately, many of these crucial activities don't go hand in hand for a variety of reasons like communication, outsourcing and different groups of employees handling the different operations [1]. These authors explain that, in order to streamline these activities, exploring continuous practices might be a useful approach. These practices also allow developers to provide the consumers with a continuous stream of products and updates [2]. The authors also expand on the idea that these approaches can allow teams and developers to deploy changes without depending on each other. Even though the term in itself is rather new, approaches like continuous integration are already heavily adopted in open source projects and industry settings [3]. Ståhl et al [4] conducted a study back in 2019 where they interviewed consultants, asking questions about the relationship between commits and complexity in software. Eleven out of twelve participants agreed that there was, in fact, a relationship between the two factors. Two of the participants actually claimed that a high frequency of smaller commits would result in higher complexity, and that it's the bigger changes in general that lead to reduced technical debt and more efficiency.

Technical debt as a term has been utilized for describing different kinds of issues in software development regarding everything from building a system to deployment. The term was first coined by Ward Cunningham in 1992, cited in [5]. In a nutshell, technical debt can describe the extra work you have to do as a result from choosing an easy and poorly optimized solution to achieve short-term benefits which will increase cost over time. Kruchten et al. [5] believe that technical debt is a result of scheduling pressure, and that design and testing gets a lower priority than maybe it should. In a time where software systems and development get steadily more complex, this complexity can result in technical debt, increasing development time and lowering the quality of the product [6]. While practices like Continuous Planning could resolve some of these issues [1]. The problem at hand might be too big to tackle without a complete rework of the traditional workflow. In 2010, the global technical debt was estimated to be close to 500 billion USD and was expected to double in the span of only five years [7]. Furthermore, the authors also explain that the lack of academic understanding of the term could be one of the reasons why this is such a difficult challenge to overcome. Ten years after, there is currently an increasing and fertile research community on Technical Debt and tools and solutions are reported widely in the literature.

With continuous software engineering, developers are able to deliver software at high paces [8]. The cornerstone of continuous software engineering is the use of automation by means of new practices and tools in the overall software process [9]. In other words, continuous software engineering aims at accelerating and increasing the efficiency of the whole software process by means of creating and establishing strong links among software engineering activities [10].

In order to expand its repercussions, continuous practices went beyond the conventional software development limits to touch operations too. Thus, DevOps represents a nonstop

amalgamation between development and operations. DevOps proficiently mixes software development including delivery, as well as operations in a fluid and lean way [11]. Therefore, DevOps assimilates a panoply of techniques targeting to decrease software production and delivery times; these techniques include continuous deployment or continuous monitoring [12]. This is crucial for developers and quality assurance professionals, benefiting from real data on the development of new products and features [13]. Consequently, DevOps can be see as a culture shift; and this change is based on a close collaboration to be built and maintained among operations, quality assurance and development [11].

The concept of DevOps surfaced in 2009 and describes a process in which operations and software developers work near one another in order to release software often and learn from the end users based on their experiences [14]. A study from 2019 explored how DevOps works in practice and did a case study on five companies in order to get insight as to how this affects their work. As a result of their study, the authors concluded that DevOps could speed up changes in the software, fixing of bugs and general handling of the production [15]. With practices like DevOps and the other continuous practices, a reduction in technical debt might be feasible. At the time this is written, and to the authors' best knowledge, there is currently no systematic literature reviews looking into the connection between these practices and technical debt, but the topics themselves have some research dedicated to them, mostly case studies and interviews. The lack of research calls for more work to be done in order to explore if and how continuous practices could be beneficial for businesses to adopt in order to reduce technical debt.

This paper will follow the structure of a systematic literature review, and the methodology will be described in section II. The search results will be presented in section III, before we present our findings in section IV, and finally wrapping things up with a conclusion in section V.

II. RESEARCH METHODOLOGY

This section is dedicated to the description of the methodology and the gathering of information in this study.

A. Systematic literature review

In the process of looking for resources, authors found a good amount of papers covering DevOps, specific continuous practices and technical debt, but the topics were mostly discussed independently and rarely in conjunction to each other. This is one of the reasons why authors decided to proceed with a systematic literature review (SLR). SLRs entail taking a deep dive into existing academic literature on a topic, while evaluating and interpreting said content [16]. One strength of a SLR is that it covers a wide aspect of research with different settings and methods, which means that we can back up the findings if the results are similar in the different kind of studies. There are two main phases of a SLR, planning the review and conducting the review, both with their sub-phases. The main idea of the first phase is to identify the need for the research to be conducted in the first place and the methods that will be used to fulfill this need. The second phase starts with gathering of available research, selection of the most relevant studies and the quality of these studies, before lastly analyzing the data. To the authors knowledge, there is no existing literature reviews on these topics in the same setting.

B. Research questions

So as to accomplish the objective of the study presented in this paper, three research questions have been formulated:

RQ1: What are the reported effects of continuous practices and DevOps with regards to technical debt?

RQ2: Could continuous practices and DevOps help reduce already existing technical debt?

RQ3: Which practices within the boundaries of DevOps and continuous practices have an impact on technical debt?

C. Review protocol

The review protocol contains the tasks required to best answer the research questions, which will be covered in the following sections. The first step is selecting the scientific databases, followed by the search strategy and study selection.

D. Data source

In order to find relevant and reliable sources, the author have selected the following scientific databases:

- ScienceDirect (http://www.sciencedirect.com)
- IEEE Xplore Digital Library (http://ieeexplore.ieee.org)
- Springer Link (http://link.springer.com)
- ACM Digital Library (http://dl.acm.org)

These databases were selected because that they are considered the most important databases in computing as a research field. Apart from that, they are available using institutional access schemas in authors' institution. The retrieval of information was executed by authors at the first quarter of 2020 in the above-mentioned databases. Zotero was used to support the process using its features of paper storage but also to dodge duplications.

E. E. Search strategy

Since the goal of this work is to find a response for the three research questions, the keywords used for the search are based on these questions. Authors use Boolean operators (AND & OR) to build the search string: AND for concatenation of expressions and terms and OR to include alternative spellings or words [17]. The search string used are the same in all databases, and is as follows:

("Technical Debt") AND ("Continuous Integration" OR "Continuous Deployment" OR "Continuous Delivery" OR "DevOps" OR "Continuous Practices" OR "Continuous Software Engineering")

The search string was modified and tested with different variations and numbers of "continuous activities". This string covers important results of the literature while keeping the number of results to a feasible number of articles.

F. Search process

The resources are all from the four databases stated above. The results were processed in three steps. Firstly, all the results from the databases were gathered. Secondly, all the titles, abstracts and keywords of all the articles were all read through to check relevance to the topic. Lastly, all of the most relevant articles were read in its entirety and included in the paper. The results can be found in section III.

G. Study inclusion and exclusion

To filter the results and decide which papers to keep and not to keep, a set of criteria for inclusion and exclusion of papers were applied.

- Inclusion criteria:
  - Literature presenting current practices used to reduce technical debt
  - Literature focusing on technical debt and either continuous practices, DevOps or both
  - Literature discussing the benefits of continuous practices
  - Literature discussing the direct connection between technical debt and continuous practices and/or DevOps
- Exclusion criteria
  - Papers that does not have technical debt, DevOps or continuous practices as the main focus.
  - Books will be excluded.
  - Papers not written in English.
  - Papers that are inaccessible.

III. SEARCH EXECUTION

In this section authors present the results of the aforementioned search process. The literature search ended on Friday the 13th of March 2020.

TABLE I. SEARCH RESULTS FROM THE FOUR DATABASES

Source	Initial result	Title, abstract & keywords	Full text
ScienceDirect	78	10	7
IEEE	104	12	6
Springer	194	10	4
ACM	134	13	6
Total	510	45	23

IV. FINDINGS

RQ1: What are the reported effects of Continuous practices and DevOps with regards to technical debt?

In 2011, a workshop took place at the International Conference on Software Engineering, with a goal of getting a better understanding of how technical debt could be managed [18]. In the report of the conference, a work by John Heintz was cited. Heintz is the owner of a consulting company called Gist Labs, provided a presentation on their current situation on maintenance and quality of code in regard to technical debt. He mentioned that businesses commonly spend too much time checking code manually, and that continuous integration combined with static analysis could reduce technical debt by making the process more autonomous. With continuous integration, one could be able to discover errors and duplicate code earlier by having repeatable tests [19]

A systematic approach named continuous refactoring describes a process where the goal is to always keep the code base at a satisfactory level of quality [20]. This concept was explored through an experiment conducted on several development teams. They explain that continuous refactoring is done by continuously repaying the technical debt before it builds up to a critical level. This does not only work as damage control, but the authors also explain that the team managers had enough knowledge to communicate better with the developers and optimize old solutions rather than working on new ones in situations where this was necessary.

One of the negatively reported practices in relation to technical debt is continuous release (or continuous deployment), or rapid releasing, which simply enough means releasing features and updates on weekly, daily or sometimes even hourly cycles [21]. Continuous release will often result in extremely tight schedules, less time for testing and relies on the users being willing to update frequently. Although all of the above could cause technical debt according to Mäntylä et al. [21], they also mentioned that Linux and FreeBSD projects worked around this by having regular updates, but no deadlines for certain tasks.

RQ2: Could continuous practices and DevOps help reduce already existing technical debt?

While most literature discuss continuous practices as approaches to prevent technical debt from building up, there's not a lot of discussion about whether or not it could decrease already existing technical debt. However, continuous analysis is mentioned as a tool to get a better overview of existing technical debt and how it impacts the developments process [22]. This way, the development team can work together to assess the situation and act to keep the debt at a manageable level. As mentioned in RQ1, continuous refactoring is when the technical debt is continuously repaid. One of the issues with the refactoring approach is that fixing old solutions are not always easy and requires experienced developers as well as a lot of time [20], which means that figuring out when the right moment to devote time and developers to doing this is crucial. Therefore, continuous analysis and continuous refactoring could potentially work really well together.

Even though continuous deployment could cause scheduling issues and high pressure as mentioned in RQ1, there are also benefits of adopting the hectic approach. Refactoring and improvement to the code based on errors and production incidents can be handled a lot earlier and more quickly as a result of frequent feedback from both end users and automated test procedures [23], effectively decreasing the technical debt as it builds up. Yli-Huumo et al. [20] also suggests adopting a portfolio approach in addition to continuous refactoring to handle the technical debt more systematically. This would be done by collecting all cases of technical debt into a list, making it easier to control.

RQ3: Which practices within the boundaries of DevOps and continuous practices have an impact on technical debt?

Continuous integration is mentioned in several studies [24][25] to pose a beneficial impact in technical debt management. This approach also allows developers to integrate modules of a complex system more often, which can avoid technical debt by not making integration more complex as the modules don't vary to much [26]. However, continuous integration requires automated testing in order to increase efficiency [27]. Ågren et al. also discusses how continuous deployment is a natural extension of continuous integration, and that these two approaches combined both improve the quality of whatever product is being developed, as well as decreasing the time it takes for the product to touch the customers. Even though the effects of these practices are positive if executed properly and with care, they aren't free of risks. As more completed builds of a product are being released, validation and tailoring the product for its purpose gets more difficult and increases the risk of work being wasted as the focus often gets shifted towards the technology rather than the users input [28]. It is challenging to prevent any form of technical debt during a development process, but by adopting continuous integration, the technical debt can be massively prevented if the correct automated tests are in place [29].

Another impactful approach is continuous experimentation. This is a trending practice among the industry giants such as Microsoft and allows for companies to gather test data from a fraction of their users by using traditional techniques like A/B testing earlier in the development process to help decide whether to keep working on a specific functionality or rework it before releasing it to the public [30].Technical debt stemming from the source code or poor experimentation logic will also often be avoided using traffic routing as part of the process, where you run multiple versions of the application in parallel [31]. A downside to this approach is resource consumption, as bandwidth and CPU usage among other things increase while doing experiments.

V. Conclusion

This paper reports the current situation and practices regarding management and accumulation of technical debt. The research conducted was articulated as a systematic literature review, following the guidelines of Barbara Kitchenham [16]. As mentioned by Martini et al. [22], management of technical debt is a topic without too much research dedicated to it, and such an important topic deserves more attention.

One of the most commonly mentioned approaches in the connection of continuous practices and technical debt is continuous refactoring, where the debt is repaid as it builds up. Continuous integration is also brought up by industry professionals as a tool for preventing technical debt by discovering errors and duplicate code early. These two practices might serve as steps in the right direction to prevent debt from reaching critical levels, and continuous integration is even a suggested strategy to adopt by these professionals (RQ1). There are several more practices in the realm of continuous practices that could affect the technical debt as well, but these aren't being discussed as much in the literature. While approaches like continuous release may increase efficiency, it might have a negative impact on technical debt due to high amounts of stress and little time for testing.

References

[1] B. Fitzgerald and K.-J. Stol, "Continuous software engineering: A roadmap and agenda," J. Syst. Softw., vol. 123, pp. 176–189, Jan. 2017, doi: 10.1016/j.jss.2015.06.063.

[2] S. S. de Toledo, A. Martini, A. Przybyszewska, and D. I. K. Sjøberg, "Architectural technical debt in microservices: a case study in a large company," in Proceedings of the Second International Conference on Technical Debt, Montreal, Quebec, Canada, May 2019, pp. 78–87, doi: 10.1109/TechDebt.2019.00026.

[3] F. Zampetti, C. Vassallo, S. Panichella, G. Canfora, H. Gall, and M. Di Penta, "An empirical characterization of bad practices in continuous integration," Empir. Softw. Eng., Jan. 2020, doi: 10.1007/s10664-019-09785-8.

[4] D. Ståhl, A. Martini, and T. Mårtensson, "Big Bangs and Small Pops: On Critical Cyclomatic Complexity and Developer Integration Behavior," in 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), May 2019, pp. 81–90, doi: 10.1109/ICSE-SEIP.2019.00017.

[5] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical Debt: From Metaphor to Theory and Practice," IEEE Softw., vol. 29, no. 6, pp. 18–21, Nov. 2012, doi: 10.1109/MS.2012.167.

[6] C. Ebert, J. Heidrich, S. Martinez-Fernandez, and A. Trendowicz, "Data Science: Technologies for Better Software," IEEE Softw., vol. 36, no. 6, pp. 66–72, Nov. 2019, doi: 10.1109/MS.2019.2933681.

[7] E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," J. Syst. Softw., vol. 86, no. 6, pp. 1498–1516, Jun. 2013, doi: 10.1016/j.jss.2012.12.052.

[8] R. V. O'Connor, P. Elger, and P. M. Clarke, "Continuous software engineering—A microservices architecture perspective," J. Softw. Evol. Process, vol. 29, no. 11, p. e1866, 2017, doi: 10.1002/smr.1866.

[9] R. Colomo-Palacios, E. Fernandes, P. Soto-Acosta, and X. Larrucea, "A case analysis of enabling continuous software deployment through knowledge management," Int. J. Inf. Manag., vol. 40, pp. 186–189, 2018, doi: 10.1016/j.ijinfomgt.2017.11.005.

[10] D. Ameller, C. Farré, X. Franch, D. Valerio, and A. Cassarino, "Towards continuous software release planning," in 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), Feb. 2017, pp. 402–406, doi: 10.1109/SANER.2017.7884642.

[11] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "DevOps," IEEE Softw., vol. 33, no. 3, pp. 94–100, May 2016, doi: 10.1109/MS.2016.68.

[12] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture," IEEE Softw., vol. 33, no. 3, pp. 42–52, May 2016, doi: 10.1109/MS.2016.64.

[13] J. Roche, "Adopting DevOps Practices in Quality Assurance," Commun ACM, vol. 56, no. 11, pp. 38–43, Nov. 2013, doi: 10.1145/2524713.2524721.

[14] M. Z. Toh, S. Sahibuddin, and M. N. Mahrin, "Adoption Issues in DevOps from the Perspective of Continuous Delivery Pipeline," in Proceedings of the 2019 8th International Conference on Software and Computer Applications, Penang, Malaysia, Feb. 2019, pp. 173–177, doi: 10.1145/3316615.3316619.

[15] L. E. Lwakatare et al., "DevOps in practice: A multiple case study of five companies," Inf. Softw. Technol., vol. 114, pp. 217–230, Oct. 2019, doi: 10.1016/j.infsof.2019.06.010.

[16] B. Kithenham, “Procedures for Performing Systematic Reviews,” p. 33, Jul. 2004.

[17] P. A. da Mota Silveira Neto, I. do Carmo Machado, J. D. McGregor, E. S. de Almeida, and S. R. de Lemos Meira, “A systematic mapping study of software product lines testing,” Inf. Softw. Technol., vol. 53, no. 5, pp. 407–423, May 2011, doi: 10.1016/j.infsof.2010.12.003.

[18] I. Ozkaya, P. Kruchten, R. L. Nord, and N. Brown, “Managing technical debt in software development: report on the 2nd international workshop on managing technical debt, held at ICSE 2011,” ACM SIGSOFT Softw. Eng. Notes, vol. 36, no. 5, pp. 33–35, 2011.

[19] R. J. Eisenberg, “A threshold based approach to technical debt,” ACM SIGSOFT Softw. Eng. Notes, vol. 37, no. 2, pp. 1–6, 2012.

[20] J. Yli-Huumo, A. Maglyas, and K. Smolander, “How do software development teams manage technical debt? – An empirical study,” J. Syst. Softw., vol. 120, pp. 195–218, Oct. 2016, doi: 10.1016/j.jss.2016.05.018.

[21] M. V. Mäntylä, B. Adams, F. Khomh, E. Engström, and K. Petersen, “On rapid releases and software testing: a case study and a semi-systematic literature review,” Empir. Softw. Eng., vol. 20, no. 5, pp. 1384–1425, Oct. 2015, doi: 10.1007/s10664-014-9338-4.

[22] A. Martini, J. Bosch, and M. Chaudron, “Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study,” Inf. Softw. Technol., vol. 67, pp. 237–253, Nov. 2015, doi: 10.1016/j.infsof.2015.07.005.

[23] E. Kula, A. Rastogi, H. Huijgens, A. van Deursen, and G. Gousios, “Releasing fast and slow: an exploratory case study at ING,” in Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, Aug. 2019, pp. 785–795, doi: 10.1145/3338906.3338978.

[24] J. Holvitie et al., “Technical debt and agile software development practices and processes: An industry practitioner survey,” Inf. Softw. Technol., vol. 96, pp. 141–160, Apr. 2018, doi: 10.1016/j.infsof.2017.11.015.

[25] B. Adams and S. McIntosh, “Modern Release Engineering in a Nutshell – Why Researchers Should Care,” in 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Mar. 2016, vol. 5, pp. 78–90, doi: 10.1109/SANER.2016.108.

[26] B. Vogel-Heuser, A. Fay, I. Schaefer, and M. Tichy, “Evolution of software in automated production systems: Challenges and research directions,” J. Syst. Softw., vol. 110, pp. 54–84, Dec. 2015, doi: 10.1016/j.jss.2015.08.026.

[27] S. M. Ågren, E. Knauss, R. Heldal, P. Pelliccione, G. Malmqvist, and J. Bodén, “The impact of requirements on systems development speed: a multiple-case study in automotive,” Requir. Eng., vol. 24, no. 3, pp. 315–340, Sep. 2019, doi: 10.1007/s00766-019-00319-8.

[28] E. Klotins, M. Unterkalmsteiner, and T. Gorschek, “Software engineering in start-up companies: An analysis of 88 experience reports,” Empir. Softw. Eng., vol. 24, no. 1, pp. 68–102, Feb. 2019, doi: 10.1007/s10664-018-9620-y.

[29] Z. Li, P. Avgeriou, and P. Liang, “A systematic mapping study on technical debt and its management,” J. Syst. Softw., vol. 101, pp. 193–220, Mar. 2015, doi: 10.1016/j.jss.2014.12.027.

[30] G. Schermann, “Continuous experimentation for software developers,” in Proceedings of the 18th Doctoral Symposium of the 18th International Middleware Conference, Las Vegas, Nevada, Dec. 2017, pp. 5–8, doi: 10.1145/3152688.3152691.

[31] G. Schermann, J. Cito, and P. Leitner, “Continuous Experimentation: Challenges, Implementation Techniques, and Current Research,” IEEE Softw., vol. 35, no. 2, pp. 26–31, Mar. 2018, doi: 10.1109/MS.2018.111094748.