Analysis of Knowledge Retention Strategies in Engineering Teams and Impact on Software Maintenance

An STS Research Paper submitted to the Department of Engineering and Society

Presented to the Faculty of the School of Engineering and Applied Science

University of Virginia

Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree

Bachelor of Science, School of Engineering

Fardeen Khan

Spring 2024

Introduction

A common idea for most people currently in the workforce is to move between jobs often to maintain a steadily increasing salary. Along with this, due to events in their personal lives, many employees take extended periods of time away from work, leading to gaps in their teams during their absence. This is especially common in engineering teams where extensive knowledge of a particular project or platform is essential to the forward progress in the teams' agenda, and with the growing role of technology and software in a business's operations. Accordingly, most engineers are not considered to add value to a company because of the additional hands they bring to the table for labor. Instead, their value lies within the extent of their knowledge of their respective work areas; in other words, they are "knowledge workers" (Lee et al., 1997). Recent research has also concluded that, combined with the shock of the COVID-19 pandemic, the frequency of employees transferring jobs has increased, and thus, its impact is further enlarged. This changing environment of engineers and the resulting rotating capacity of knowledge about a codebase, especially larger ones, can have a lasting negative impact on the software development lifecycle. As software engineering tools become more prone to obsolescence over time, developer time is spent more towards maintaining their products. Thus, it is also necessary for developers maintain their knowledge of the workings of the project.

Software engineering teams have copious amounts of standards and guidelines regarding technical aspects of the project such as coding practices and documentation. In larger businesses and in-depth applications that require knowledge of multi-layered codebases, onboarding and introducing a new member of the engineering team to the current project may consume valuable time and resources. As a result, this high employee turnover can result in significant disruptions in software engineering teams leading to project delays, technical debt, and increased costs. Therefore, implementing strategies to mitigate the effects of employee turnover is crucial for maintaining productivity while adhering to high-quality software. This paper will explore assorted studies and methods to mitigate these effects, proposing tools such as knowledge transfer documents, code documentation, assignment of tasks based on familiarity and unfamiliarity, and certain training methods to maintain cohesiveness and streamline collaboration in dynamic engineering teams.

This issue has been encountered throughout much of my time working as a software engineer in different companies of diverse sizes ranging from small startups to a mid-size insurance company and a large banking company, most notably migrating the platform for a selfservice tool from Node 14 to Node 18. In each of these scenarios, team members were absent due to team rotation or personal reasons, and steps had to be taken to minimize the gap in both potential developer work and knowledge of the project. This STS Research paper will cover a few of the mitigations engineering teams from various industries have implemented into their operations to minimize the impacts of instability in their teams, and their relative rates of efficacy in achieving this goal.

Literature Review

In the field of software engineering, a key indicator of success is the process the team uses to assign and plan project tasks among its team members, such that when faced with setbacks or changes in the project requirements, there is no critical downtime in the team's progress. In modern cases, popular team management frameworks include Agile practices such as Scrum and Kanban. Both these frameworks also have some similarities to Rational Unified Process (RUP), a cross-team collaboration method implemented at IBM in 2003.

RUP promotes a development philosophy of iterative and incremental changes happening throughout multiple cycles, and the system requirements are created with the interaction between the system itself and the end user in mind. A few of its other key aspects are its focus on defining software architecture early in the process, standardizing best practices for modeling architecture through models such as UML diagrams, and risk management – such as identifying and analyzing the cost of potential disruptors in development – at the beginning of the process. As such, aside from the similarities with Scrum and Kanban, RUP presents a more disciplined and structured approach to software engineering, whereas the former emphasizes adaptability throughout the life of the project.

In terms of resistance to instability in an engineering team, autonomous and Agile teams have been shown to cope better than synchronous teams in terms of changing environments such as different team members and requirements. Using a framework like RUP called Team-RUP, an agent-based simulation was conducted by Levent Yilmaz in Auburn Modeling & Simulation Laboratory where the simulated agents were the development team members practicing different methods of collaboration such as Agile, Autonomous, Concurrent, and Synchronous frameworks. Agents were exposed to numerous factors in the simulation such as tasks, cooperation mechanisms, reward mechanisms, and, most importantly, employee turnover. The flow of this simulation is diagrammed below in Figure 1.

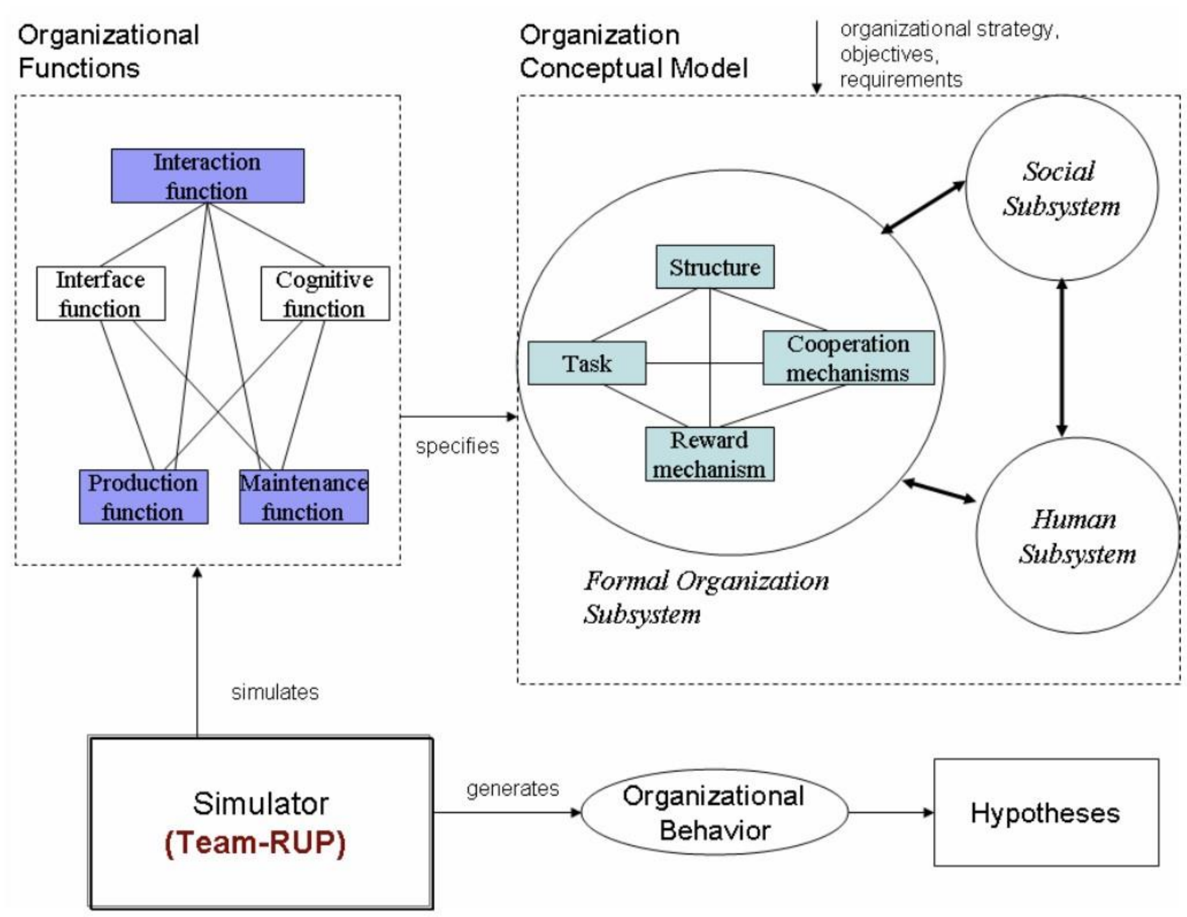

Figure 1: Conceptual Makeup of Team-RUP Process Model

Specifically, the Formal Organization Subsystem contains the modeling for the inner workings of a development team and the interactions defining how members may collaborate with one another.

For example, the Structure element defines the links and relationships between each agent or team member, and how they communicate. The Cooperation Mechanism represents the working teams and business relations among them, the Tasks model the responsibilities given to each agent, and Reward mechanism defines the motivations for each agent, and the Social subsystem represents the social interactions and relationships among them. Order of completion of tasks is represented by the following set:

$$\{(x_i, y_j) \in C \times C \mid i < j\}$$

Figure 2: Set Representing Order of Completion of Problems when Introducing Newer Problems

From a high level, this set corresponds to the placing of elements in an array of Tasks, where the order of elements represents the order of completion for those Tasks, so if there is a pair (x_i, y_k), and $i<k$, that means task i needs to be completed before task k.

The final and most pertinent portion of this framework is its modeling of Employee Rewards and Turnover. In this scenario, the turnover rate can be defined as the rate at which team members are leaving, joining, or are away from a team due to personal reasons, job change, or program rotations, as is common in many junior roles, represented by the equation in Figure 3:

$$\beta_{ti} = \frac{m}{k}$$

Figure 3: Removed Inversions as a Function of

Team Performance and Productivity

In this equation, $\square_{ti}$ represents the number of removed inversions, or tasks, *m,* completed per team, *k*, thus representing overall team performance. Naturally, this will be lower if there are disruptions in the team workflow.

After running the simulations on the Team-RUP framework, the results supported the first hypothesis, showing that asynchronous teams exhibited greater variability in productivity, staff utilization, timeliness, and quality, indicating less predictability:

> "Autonomous and Concurrent behaviors exhibit greater variability than Agile and Synchronous teams. That is, teams that coordinate their work efforts concurrently are less predictable than those that synchronize their activities." (Yilmaz, 2007)

Furthermore, Yilmaz's simulations also support the idea that bottom-up teams (Autonomous and Agile) coped better with changing requirements, referred to as "Stability," and maintained higher quality levels compared to top-down teams (Concurrent and Synchronous) as shown in Figure 4.

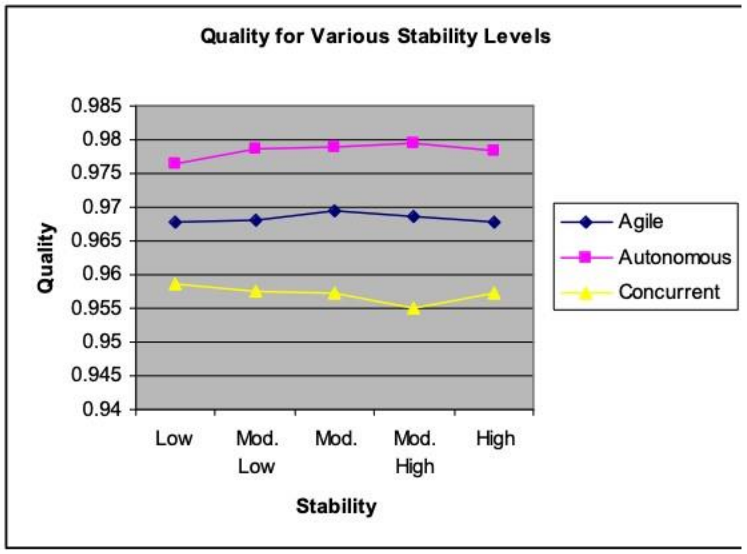

Figure 4: Performance as a Function of Different Stability Levels

> "Each of the aforementioned validation points is supported by these diagrams. The Autonomous and Agile teams cope better with requirements change than Concurrent and Synchronized teams."
> (Yilmaz, 2007)

As seen in many software engineering environments today, Autonomous and Agile development teams such as those that utilize Kanban or Scrum are widely accepted as the norm for team collaboration frameworks. In the context of software maintenance, bottom-up team architectures, such as autonomous teams, would perform well given their ability to adjust and react to the changing requirements that may be present in maintaining software after its initial release. In comparison, Agile teams can also exhibit successful patterns in maintenance, as their inherent structure is comprised of incremental and iterative allocation of tasks. (Yilmaz, 2007)

Of course, when considering the changes in a team's makeup, it is also important to consider how to retain all the knowledge from a departing engineer and transfer it to a newcomer. Perhaps an extreme analogy of this scenario is the contribution of individuals to an open-source project. Open-source projects are defined as software with a license that grants any user permission to contribute or make changes to the source code. In some instances, there may be a core team of engineers dedicated to contributing, but the ability to propose changes and add new features is open to everyone. Some examples of these projects include Linux, Android, TensorFlow machine learning and artificial intelligence library, and Kubernetes. It is possible for open-source platforms such as these to have profound impacts and sustain stability as a platform despite the changing group of engineers and a rotation of crucial knowledge such as system architecture, and design decisions implemented by engineers of varying performance levels. According to a proposal written by Rashid et al. 2020, to explore the knowledge retention practices in open-source projects, the biggest factors related to these contributions are "traditional software development, OSS knowledge loss mitigation techniques, and OSS community guides". Using these guides, it is possible to create a picture of how each project can manage this instability.

In turn, using the community guidelines for contributing to the previously mentioned open-source Linux project, there are 8 key community guidelines prefaced by the Linux Foundation for potential contributors: Code of Conduct, Licensing, Patch Submission, Code Style, Testing, Documentation, Bug Reporting, and Communication. In terms of development, the Patch Submission, Code Style, and Testing points cover the best way to streamline its development process. For example, submitting patches for inclusion into the mainline kernel should be done through the Linux development mailing list. There are also predefined guidelines for indentation, effective use of conditionals, formatting braces and spaces, and using and creating test suites that exhaustively cover recent changes. However, for developer collaboration, the points defining Documentation and Communication are most relevant. The foundation emphasizes the importance of documentation of new changes for its assistance to other contributors: "… adequate documentation will help to ease the merging of new code into the kernel, make life easier for other developers, and will be helpful for your users. In many cases, the addition of documentation has become essentially mandatory." Moreso, the importance of steady communication in a "decentralized" community is emphasized through the use of mailing lists, forums, and internet chat rooms. (docs.kernel.org, 2024)

When compared to open-source environments such as Linux, many of the issues proposed by cohesive development teams for closed-source projects and projects within companies seem to be easier to solve. However, they still enforce many of the same requirements such as Coding Style, Patch Submission, Testing strategies, Documentation, and Communication. Communication is undoubtedly easier within a company; however, documentation is the main practice observed within teams.

Although changes are documented during each Sprint iteration and GitHub pull request, a significant portion of the time leading up to a developer's planned departure is devoted to preparing knowledge transfer documents and ensuring the reasoning behind the engineer's changes is relayed to the rest of the team. Considering my experience, in each of the previous three internships, when working on a business-critical project throughout a summer, the final week was spent halting any items in progress and ensuring the project was prepared to be continued by the rest of the development team up on my departure. During this time, any problems and difficulties must also be documented, along with the pure technical definition of our additions. Such items included justifying design and architecture decisions, explaining the functionality of container services and how they interact with other platforms, and describing current progress and next steps to continue the development of the project.

When it comes to continuing the previous developer's tasks, the team can use the resources left behind. However, in the case new team members are added, there will be a significant knowledge gap, as the onboarding process will require them to acquaint themselves with the codebase. However, although the documentation provided may prove helpful, inexperienced, and junior developers may not effectively retain the information held in these resources (Forward et al. 2002), leading to losses in team productivity. An effective solution called "knowledge diffusion" (Etemadi et al. 2021) to this problem is proposed to assign the developers tasks by keeping their knowledge areas in mind during the assignment of the tasks. In other words, during the initial phases of learning the codebases, the new developer's relative areas of expertise will be accounted for, and he or she will be assigned tasks outside these areas. While this may sound counterproductive, eventually, it will lead to gains of information concerning the codebase. Assigning the developer tasks in areas in which they are not familiar will require them to problem solve either alone or consult a more experienced member to collaborate with. While possibly leading to longer times or more developers spent on completing a task in the beginning, either the hands-on problem solving or pair programming and collaboration with another, more experienced engineer, will leave the new engineer with more insightful knowledge – in other words, the information will have diffused from one engineer to another.

Conversely, assigning the most experienced developer with tasks in areas they are already most skilled in will allow for faster completion times in the short run, but will inevitably lead to him or her becoming the sole source for the specific area. Doing so will lead to the exact issue at hand – where one developer holds the most knowledge about a specific function, and upon a possible departure, will take this valuable resource with them (Etemadi et al. 2021). Upon conducting these exercises, software engineering teams were able to measure reductions in the cost of software maintenance tasks by up to 50% and resulted in knowledge successfully being equally distributed among team members, reducing the likelihood of losing crucial knowledge and therefore, creating a more "resilient project".

Methodology

A significant framework used in this research was the knowledge-management framework. This framework allows for the analysis of the efficacies presented by various forms of maintaining knowledge in software engineering teams. Using this framework, it is possible to explore the creation, sharing, and application of knowledge within software engineering teams to achieve enhanced resilience. In turn, this paper explored distinct factors such as knowledge repositories, communication channels and cross-training.

Additionally, during this literature review, select research topics were analyzed by their relevance to contributions of different engineering practices on mitigating the negative effects employee instability and turnover had on the productivity of the team. In doing so, this paper covered the contributions of different team management styles such as Agile, Asynchronous, Concurrent, and Autonomous styles on performance while experiencing instability. Furthermore, using methods described in a proposal to measure the efficiency of knowledge retention practices in software engineering contexts, it allowed for the comparison of a successful, massively opensource project such as Linux, to be analyzed for its practices, and propose implementation into a more centralized software engineering environment.

Results and Discussion

Ignoring the specific outlying skill levels of some software engineers, the effectiveness of different team management strategies can vary greatly by the strategies the teams use. Using a simulation framework called Team-RUP, Levent Yilmaz was able to quantify the quality of output by software engineering teams when faced with various levels of stability in the makeup of the engineering team. Upon completion of this simulation, two hypotheses were supported: "Asynchronous cooperation models are less predictable than synchronous models (Shamsi, Chu, and Brockmeyer, 2005)" and "A bottom-up strategy is more adept at responding to change (Pizka, M. and Bauer, A., 2004)". In a market where changing one's occupational positions every one to two years is common, the implementation of Agile and Autonomous teams was supported in order to counteract employee turnover as a source of instability among teams.

Furthermore, when considering open-source projects as ideal examples of software engineering projects with consistently high turnover, there are three data sources to include in an analysis of their success: knowledge retention practices used in traditional software development, knowledge loss mitigation techniques in projects, and community guides. Using the Linux Foundation as a model, the practices most supportive to a resilient team with inconsistent contributors are standardized documentation policies and communication channels open to public access such as mailing lists, forums, and in modern environments, Slack archives, to allow for unrestricted access to knowledge.

Forward et al. (2002) proposed the issue of a lack of effectiveness of documentation for many developers, especially younger and junior developers, thus diminishing its value. To this, Etemadi et al. posit a form of hands-on task assignment policy, whereby developers are assigned tasks according to their areas of discomfort in the codebase. At first, assigning tasks to developers in order of least experienced to most experienced may sound counterproductive, but in the long term, it will require the developers to consult other, more experienced developers or solve problems on their own, and thus gain an understanding that way.

Conclusion

Engineering teams prone to high instability in their team members should use Agile practices such as Scrum or Kanban in their projects, as these frameworks pose the most resilience to turnover due to their nature of incremental and iterative task management. However, it is also important to provide team members with the necessary resources to maintain and consult others' contributions using standardized documentation practices and open communication forms. Not only will this allow for setbacks or changes in requirements, but it will also allow the opportunity to give newcomers assignments encompassing various aspects of their project through knowledge diffusion practices in place of standard documentation review.

References

Lee, T. W., & Maurer, S. D. (1997). The retention of knowledge workers with the unfolding model of voluntary turnover. *Human Resource Management Review*, *7*(3), 247–275. https://doi.org/10.1016/s1053-4822(97)90008-5

Forward, A., & Lethbridge, T. C. (2002). The relevance of software documentation, tools and technologies. *Proceedings of the 2002 ACM Symposium on Document Engineering*. https://doi.org/10.1145/585058.585065

Fontanet Losquiño, D., & Urdell, T. (2014). *Why Do Developers Struggle with Documentation While Excelling at Programming*.

Etemadi, V., Bushehrian, O., & Robles, G. (2022). Task assignment to counter the effect of developer turnover in software maintenance: A knowledge diffusion model. *Information and Software Technology*, *143*, 106786. https://doi.org/10.1016/j.infsof.2021.106786

The linux kernel. A guide to the Kernel Development Process - The Linux Kernel documentation. (n.d.). https://docs.kernel.org/process/development-process.html

Rashid M, Clarke PM, O'Connor RV. A mechanism to explore proactive knowledge retention in open source software communities. *J Softw Evol Proc*. 2020; 32:e2198.

Shamsi, J., Chu, C., and Brockmeyer, M. 2005. Towards Partially Synchronous Overlays: Issues and Challenges. In *International Workshop on Advanced Architectures and Algorithms for Internet Delivery and Applications*, Orlando, FL.

Pizka, M. and Bauer, A. 2004. A Brief Top-Down and Bottom-Up Philosophy on Software Evolution. *7th International Workshop on Principles of Software Evolution*, Kyoto, Japan, 131—136

Levent Yilmaz. 2007. Exploring the impact of employee turnover on the effectiveness of software development team archetypes. In Proceedings of the 2007 spring simulation multiconference - Volume 2 (SpringSim '07). Society for Computer Simulation International, San Diego, CA, USA, 94–101.