Architecture Maintenance and Rapid Prototyping: A Trade-off Analysis

1st Brendan Smiley
Department of Software and Electrical Engineering
University of Calgary
Calgary, Canada
brendan.smiley@ucalgary.ca

***Abstract*—The balance between architecture maintainability and rapid prototyping is a trade-off that software development teams often face during a project's life cycle. Software engineers can struggle with this balance and knowing what is best for the current state of the project. Discussions take place about the real-world business impacts of the trade-off between architecture maintainability and rapid prototyping. The purpose of this research study is to find industry solutions for dealing with the trade-off of architecture maintainability and rapid prototyping. Industry solutions discussed include enforcing engineering acronyms, improving code quality, and selecting the correct architecture for both current and future project scope.**

***Index Terms*—Time to Market, Velocity, Code Quality, Software Engineering, Balancing Trade-Offs, Maintainability**

I. Introduction

Software engineers are constantly faced with releasing product features within a tight deadline. These deadlines are present both during the infancy and maintenance stages of the software application. However, the goals at these stages are different. During the infancy stage, the Time to Market (TTM) is the more important metric because it motivates developers to release a Minimal Viable Product (MVP) sooner. MVPs reduces the costs required to confirm market viability [1]. During the maintenance phase, the development velocity (DV) is the crucial metric to a product [2]. Furthermore, TTM and DV can be seen as short-term and long-term times required to release software features (respectively). Time to market and developer velocity are tangible business metrics, but they do not explicitly tell software engineers what they should prioritize while designing and programming. To a software engineer, a large imbalance in the time required to deliver features in short-term and long-term can be seen as technical debt [3]. Moreover, technical debt occurs when software engineers prioritize rapid prototyping over properly designed architectures [3]. The challenges of technical debt introduces the goal of this research paper. This study aims to answer the following question: ***What strategies can engineers can use to balance the trade-off between architecture maintainability and rapid prototyping***?

The rest of this paper is structured as follows: section II discusses key definitions and prior work relevant to this research, section III explains the procedure of research. In section IV, the findings of the research are revealed. Lastly, section V discusses the real world significance of the results with suggestions for future research.

II. Background

Project managers are often focused on certain aspects of a project while engineers within the team are focused on others. This section aims to describe terminology both parties will commonly use in industry.

Companies often create a MVP, which is the first version of a product which contains only essential functionality with the goal of releasing a product as quickly as possible [1], [4]. Developing an MVP in such a short time span helps the team learn what functionalities are useful to users sooner [4]. MVPs can help remove time wasted on low value features. Finally, a product's market viability is often confirmed by making a MVP [1]. Companies must be concerned with the time required to create and launch a MVP into the market. This time can be defined as the TTM [5]. Additionally, management is often focused on the Development Velocity (DV), which can be defined as the amount of work done over a period of time, specifically focusing on new features, bug fixes and other code modifications [2].

Software engineers often focus of the internal quality of code. Internal quality of code is how well designed the codebase is in terms of software architecture, design patterns, flexibility, robustness, and other coding practices and standards [6], [7]. The reason software engineers focus on internal quality is because poor quality leads to technical debt. Technical debt is the magnified work required due to a unorganized and unclean codebase [6]. Technical debt occurs because of shortcuts taken by software engineers, an potential shortcut would be a architecture that is not maintainable [3]. Architecture maintainability can be defined as the ease of modifying, adding, or deleting portions of the codebase [8]. Key components of architecture maintainability include modularity, encapsulation, loose coupling, high cohesion, documentation and standardization [8].

III. Method

This study used the Grey Literature Review methodology to conduct research, details about Grey Literature Review are provided in [9] and [10]. Additionally, the IEEE Conference Template was used as a starting point to format this paper

in Overleaf with LaTeX [11], [10], [12]. Finally, the following papers were used as references as to how an IEEE paper should be; [13] and [14].

A. Search Process

Resources were selected from industry-specific blogging sites, forum pages, and university course notes. Examples include Martin Fowler's website, GeeksForGeeks, and Medium. Keywords searched were technical debt, software maintainability, software trade-offs, code quality, etc.

B. Source Selection

References were selected using content relevance and author credibility as filters. Additionally, forums (ex: Reddit) had an additional requirement of having a high number of positive votes and engagement.

C. Data Analysis

The credibility of references was evaluated by ensuring the date modified was within 5 years and by accessing the author's internet presence and industry experience. More specifically, the author's credibility was validated by viewing they're blog portfolio and social media handles. Once the data was validated to be credible it would be analyzed by focusing on themes. The information from references was organized and grouped into common themes. The themes with the most reoccurrence were used in the findings.

IV. Results

A. Thematic Analysis

1) Industry Habits, Trade-Offs, and Best Practices: Our study indicates that software engineers often face trade-offs between architecture maintainability and rapid prototyping. Martin Fowler highlights this challenge: "Usually the pressure to deliver functionality dominates the discussion, leading many developers to complain that they don't have time to work on architecture and code quality [6]."

To maintain internal quality, engineers enforce common principles like Don't Repeat Yourself (DRY) and SOLID [15], [16]. Best practices likes design patterns, modularity, loose coupling, and high cohesion help reduce technical debt in the codebase [17], [18], [19]. Regardless of the architectural design choice, these principles remain applicable [20]. Improving the internal quality of the codebase gives the developers more flexibility in choosing the software architecture, as they will spend less time if a rewrite is required [21]. This flexibility allows developers to focus on solving the problem before making clean architecture.

Our findings also suggest that prioritizing functionality before optimizing code is a widely accepted industry principle [22]. Futhermore, industry uses the principle "You Aren't Gonna Need It" (YAGNI) to remind developers to avoid unnecessary functionality [23]. Additionally, the industry use the famous quote by Donald Knuth as a important lesson to programming:

"Premature optimization is the root of all evil, or at least most evil in programming; the true issue is that programmers have worried about efficiency much too much, in the wrong places, and at the wrong times." [24], [25]

On the other side of the trade-off, developers enable more rapid prototyping by following the Pareto principle (20/80 rule) which states "The 80/20 rule, also known as the Pareto principle, is a statistical rule that states that 80% of outcomes result from 20% of causes. [26]".

The findings also revealed that refactoring is harder the higher the level of abstraction [3]. This means that the foundation architecture decided may be challenging to change. However, developers suggest that if the codebase is modular and good quality then the workload of changing architectures is reduced [6].

2) Grouping of Themes: After conducting this research, the findings from industry can be organized into the following themes. The themes summarize the current challenges and solutions in the software development industry.

TABLE I
Overview of Industry Problems

Theme	Summary
Principles to Consider	20/80 Rule, Premature Optimization, Over-Engineering
Business Metrics	Time to Market, Developer Velocity
Architecture Selection	Complexity, Refactoring, Codebase Maintainability

TABLE II
Overview of Industry Solutions

Theme	Summary
Princple Acronyms	KISS, YAGNI, SOLID, DRY
Code Quality	Design Patterns, Modularity, Loose Coupling, High Cohesion
Practices Enforced	Refactoring, Pair Programming
Architecture Selection	Well-Balanced Architecture

B. Patterns from Industry

Architectural choices significantly impact a projects growth. Architectures like microservices provide better maintainability but they add complexity to the boundary logic between services [27]. Independent services is often too complex for the infancy stage of a application, but are easier to modify and add during the maintenance stage of a application. Choosing a monolithic architecture reduces initial programming complexities but faces massive drawbacks when modification is required.

Industry perspectives remain divided on the ideal starting architecture [28]. However, many conversations and frameworks suggest layered or plugin architectures to provide a balanced approach to the trade-off [29], [30]. Looking at currently used frameworks for development, our study shows that layered architecture is the most suitable for a web application [29], [30], whereas a plugin architecture is more suitable for a desktop application [31]. Taking a balanced architecture like layered or plugin helps mitigates over-engineering while still promoting modularity and cohesion which reduces technical debt. Furthermore, selecting a well-balanced architecture

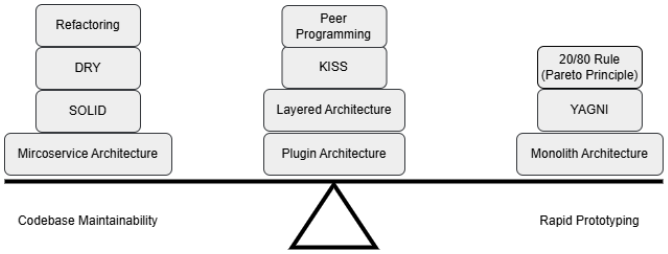

Fig. 1. Balancing the trade-off between codebase maintainability and rapid prototyping

makes future architectural refactoring easier as provides a good foundation to make good quality code.

The goal of Figure 1 is to summarized the balance between codebase maintainability and rapid development. Figure 1 shows the architectures with a heavier block which symbolizes that the architecture selected has a strong base impact for the amount of codebase maintainability and rapid development the project will have. Moreover, the other themes discovered in this research also have a impact on the balance. Thus, if a specific architecture is needed, it can be offset by following specific acronyms and principles.

C. Practical Insights

1) Current Practices: Refactoring and pair programming is used in industry to remove code smells [32], [33]. These methods reduce technical debt and thus makes the codebase more maintainable and flexible without requiring an overly engineered architecture [33]. There is practical distinction between refactoring and pair programming. Refactoring occurs after the code is complete whereas pair programming happens while the code is developed. Pair programming would be more conducive to balancing TTM and DV.

2) Actionable Recommendations: An opportunity for improvement in the software engineering industry is the development of a standardized pair programming methodology. At the time of this study, no International Organization for Standardization (ISO) standard exists specifically for effective pair programming methodologies. The closest standard discovered was ISO/IEC TR 24587:2021, which covers general agile principles but does not address specific methodologies [34]. However, ISO does recognize the benefits of pair programming, as it is included in ISO 27001 as way to create more secure code [35]. A standardized method for pair programming could include a section about enforcing the acronyms KISS, YAGNI, SOLID, and DRY. Adding this standard would provide a more consistent enforcement on the balance between architecture maintainability and rapid prototyping.

V. Conclusion

In this study, Grey Literature Review was performed on references to evaluate the challenges and solutions in balancing architectural maintainability and rapid feature development. The findings indicate that the architecture chosen has a significant impact on the codebase's maintainability and consequently the development velocity. However, the selected architecture should be suitable for the project's current scope and immediate growth needs, rather than future-proofing for all possibilities. The selected architecture lays a foundation for internal code quality enforcement, and thus has the largest impact out of the solutions discussed in this study.

To support rapid prototyping while maintaining code quality, software engineers can choose a well-balanced architecture that will reduce the risk of under or over engineering. As the project scales out of the MVP stage, additional maintainability can be achieved through architectural refactoring. Refactoring at this level of abstraction is typically labor-intensive due to accumulated technical debt. However, if the developers were already enforcing SOLID, DRY, KISS, and relevant design patterns then the codebase will be modular, reducing the effort required to transition to more a complex architecture.

Software engineers must be mindful of the project's current and future scope to avoid over-engineering and premature optimizations. Software developers in industry often ask themselves:

- Am I following KISS (Keep It Simple Stupid)?
- Am I productively applying Pareto Principle (20/80 rule)?
- Do we need this for this deadline (You Aren't Gonna Need It - YAGNI)?

Finally, the findings demonstrated that pair programming serves as an effective method to reinforce coding practices and principles that balance both architecture maintainability and rapid prototyping. Future research should investigate how pair programming approaches can provide a systematic workflow. Additionally, a quantitative study could be done to identify how different pair programming approaches impact internal quality code.

LLM Usage

This paper used ChatGPT [36] to better understand how LaTeXworks, how to use LaTeXto create a research paper, and IEEE structuring questions. Additionally ChatGPT was used to find additional search words that could be used to find references. Finally, ChatGPT was used to provide feedback about the quality of writing.

References

[1] Indeed Editoral Team. "Minimum Viable Product: Definition, Importance, and Examples". Accessed: Jan. 27, 2025. [Online]. Available: https://ca.indeed.com/career-advice/career-development/minimum-viable-product

[2] Sai Krishna Kethan. "What is Software Development Velocity and How to Improve it?" Hatica. Accessed: Feb. 03, 2025. [Online]. Available: https://www.hatica.io/blog/software-development-velocity/

[3] R. Santos. "SENG 401 - Software Architecture Technical Debt". Accessed: Feb. , 2025. [Online]. Available: https://d2l.ucalgary.ca/d2l/le/content/649709/viewContent/6889826/View

[4] M. Seibel, Y Combinator. "How to Build An MVP — Startup School". Accessed: Feb. 07, 2025. [Online]. Available: https://www.youtube.com/watch?v=QRZ_l7cVzzU

[5] Shopify. "Time to Market Explained: How To Reduce Your TTM.". Accessed: Feb 02, 2025. [Online]. Available: https://www.shopify.com/ca/blog/time-to-market.

[6] M. Fowler. "Is High Quality Software Worth the Cost?". Accessed: Feb. 06, 2025. [Online]. Available: https://martinfowler.com/articles/is-quality-worth-cost.html

[7] R. Santos. ”SENG 401 – SOFTWARE ARCHITECTURE REFACTORING”. Accessed Feb 07, 2025. [Online]. Available: https://d2l.ucalgary.ca/d2l/le/content/649709/viewContent/6889825/View

[8] S. Khan “Maintainability in System Design and Architecture,” DEV Community. Accessed: Feb. 07, 2025. [Online]. Available: https://dev.to/sardarmudassaralikhan/maintainability-in-system-design-and-architecture-1ine

[9] R. Santos. ”SENG 401 – Assignment 1”. Accessed Feb 02, 2025. [Online]. Available: https://d2l.ucalgary.ca/d2l/le/content/649709/viewContent/6844934/View

[10] Purdue University. ”IEEE Style”. Accessed: Feb 01, 2025. [Online]. Available: https://owl.purdue.edu/owl/research_and_citation/ieee_style/index.html

[11] “IEEE Conference Template.” Accessed: Feb. 03, 2025. [Online]. Available: https://www.overleaf.com/latex/templates/ieee-conference-template/grfzhhncsfqn

[12] “Overleaf, Online LaTeX Editor.” Accessed: Feb. 03, 2025. [Online]. Available: https://www.overleaf.com

[13] M. Leça and R. Santos, ”Towards User-Focused Cross-Domain Testing: Disentangling Accessibility, Usability, and Fairness”. Accessed: Jan 31, 2025. [Online]. Available: https://arxiv.org/pdf/2501.06424

[14] E.Quadros, R. Prikladnicki, and R. Lahm. ”A Grey Literature Review on the Impacts of Covid-19 in Software Development”. Accessed: Jan 31, 2025. [Online]. Available: https://pdfs.semanticscholar.org/4c06/f5fdd8d83c4003b7cc1b9ae73e9ca1cd2a8d.pdf

[15] J.-J. Levy, “Principles of Software Development: SOLID, DRY, KISS, and more,” Scalastic. Accessed: Feb. 07, 2025. [Online]. Available: https://scalastic.io/en/solid-dry-kiss/

[16] GeeksForGeeks, “SOLID Principles in Programming: Understand With Real Life Examples,” GeeksforGeeks. Accessed: Feb. 07, 2025. [Online]. Available: https://www.geeksforgeeks.org/solid-principle-in-programming-understand-with-real-life-examples/

[17] Refactoring Guru, “Why should I learn patterns?” Accessed: Feb. 07, 2025. [Online]. Available: https://refactoring.guru/design-patterns/why-learn-patterns

[18] TehBoyan, “Answer to ‘What does “low in coupling and high in cohesion” mean,’” Stack Overflow. Accessed: Feb. 07, 2025. [Online]. Available: https://stackoverflow.com/a/21281005

[19] E. Sabag, “What exactly does ‘Low coupling, High cohesion’ mean?,” Wix Engineering. Accessed: Feb. 07, 2025. [Online]. Available: https://medium.com/wix-engineering/what-exactly-does-low-coupling-high-cohesion-mean-9259e8225372

[20] GeeksForGeeks, “Difference Between Architectural Style, Architectural Patterns and Design Patterns,” GeeksforGeeks. Accessed: Feb. 07, 2025. [Online]. Available: https://www.geeksforgeeks.org/difference-between-architectural-style-architectural-patterns-and-design-patterns/

[21] A. Goyal, “Refactor, Reengineer, Rewrite: Choosing the Right Path for Software Evolution,” Medium. Accessed: Feb. 07, 2025. [Online]. Available: https://medium.com/@anil.goyal0057/refactor-reengineer-rewrite-choosing-the-right-path-for-software-evolution-f9f999940983

[22] I. Khalid, “Make It Work, Make It Right, Make It Fast: The Evolution of Software Development,” Medium. Accessed: Feb. 07, 2025. [Online]. Available: https://medium.com/@ibk9493/make-it-work-make-it-right-make-it-fast-the-evolution-of-software-development-fbbc1eddd33e

[23] GeeksForGeeks, “What is YAGNI principle (You Aren’t Gonna Need It)?,” GeeksforGeeks. Accessed: Feb. 07, 2025. [Online]. Available: https://www.geeksforgeeks.org/what-is-yagni-principle-you-arent-gonna-need-it/

[24] D. E. Knuth, The art of computer programming. 1: Fundamental algorithms, 2. ed., 7. print. Reading, Mass: Addison-Wesley, 1982. Note: This is provided as a supplementary reference for respect to the original author, it has been paired with a more current reference.

[25] GeeksForGeeks, “Why Premature Optimization is the Root of All Evil?,” GeeksforGeeks. Accessed: Feb. 07, 2025. [Online]. Available: https://www.geeksforgeeks.org/premature-optimization/

[26] H. S. Kalsi, “The 80/20 Rule: A Guide for Software Developers,” Medium. Accessed: Feb. 07, 2025. [Online]. Available: https://medium.com/@harrpreet/the-80-20-rule-a-guide-for-software-developers-68555462fd3f

[27] Kong, “Essential Guide to Understanding Microservices Architecture,” Kong Inc. Accessed: Feb. 07, 2025. [Online]. Available: https://konghq.com/blog/learning-center/what-are-microservices

[28] “You’re faced with conflicting opinions on software architecture. How can you steer towards the best approach?” Accessed: Feb. 07, 2025. [Online]. Available: https://www.linkedin.com/advice/3/youre-faced-conflicting-opinions-software-7zxsc

[29] E. Altynpara, D. Bestaieva, “Understanding the Web Application Architecture Fundamentals,” Cleveroad Inc. - Web and App development company. Accessed: Feb. 07, 2025. [Online]. Available: https://www.cleveroad.com/blog/web-application-architecture/

[30] University of Waterloo, ”Plugin Architecture,” prepared by students D. Dolan, B. Gao, D. J. F. Liu, N. Chaudhry and S. Gemnay, S. Madigan, C. Mannes, C. Souza, under the supervision of M. Nagappan, A. R. K. Ram, A. Vayiravan, and W. Zhu. [Online]. Available:https://cs.uwaterloo.ca/ m2nagapp/courses/CS446/1195/Arch_Design_Activity/Plug

[31] T. Weidinger. “Tauri 2.0 Stable Release,” Tauri. Accessed: Feb. 07, 2025. [Online]. Available: https://v2.tauri.app/blog/tauri-20/

[32] C. Klapp, “Code Smell — When to Refactor,” Better Programming. Accessed: Feb. 02, 2025. [Online]. Available: https://medium.com/better-programming/code-smell-when-to-refactor-e18f1dca2f01

[33] “Best Practices for Managing Technical Debt Effectively — Axon.” Accessed: Feb. 07, 2025. [Online]. Available: https://www.axon.dev/blog/best-practices-for-managing-technical-debt-effectively

[34] ISO. ”ISO/IEC TR 24587:2021(en) Software and systems engineering — Agile development — Agile adoption considerations”. Accessed: Feb. 07, 2025. [Online]. Available: https://www.iso.org/obp/ui/en/#iso:std:79011:en

[35] Ef Oni, “ISO 27001 and the Evolution of Secure Coding,” Security Compass. Accessed: Jan. 25, 2025. [Online]. Available: https://www.securitycompass.com/blog/iso-27001-and-the-evolution-of-secure-coding/

[36] OpenAI. ”ChatGPT: A Large Language Model”. ChatGPT. Accessed: Feb. 2, 2025. [Online]. Available: https://chatgpt.com/