Embracing Technical Debt, from a Startup Company Perspective

Embracing Technical Debt, from a Startup Company Perspective

Terese Besker[1], Antonio Martini[2a,b], Rumesh Edirisooriya Lokuge[3], Kelly Blincoe[3], Jan Bosch[1]

[1]*Computer Science and Engineering, Software Engineering, Chalmers University of Technology Göteborg, Sweden*
besker@chalmers.se, jan.bosch@chalmers.se

[2a]*CA Technologies Strategic Research Team, Barcelona, Spain*
[2b]*Programming and Software Engineering, University of Oslo Oslo, Norway*
antonima@ifi.uio.no

[3]*Dept. of Electrical and Computer Engineering, The University of Auckland Auckland, New Zealand*
kblincoe@acm.org, redi099@aucklanduni.ac.nz

***Abstract*— Software startups are typically under extreme pressure to get to market quickly with limited resources and high uncertainty. This pressure and uncertainty is likely to cause startups to accumulate technical debt as they make decisions that are more focused on the short-term than the long-term health of the codebase. However, most research on technical debt has been focused on more mature software teams, who may have less pressure and, therefore, reason about technical debt very differently than software startups. In this study, we seek to understand the organizational factors that lead to and the benefits and challenges associated with the intentional accumulation of technical debt in software startups. We interviewed 16 professionals involved in seven different software startups. We find that the startup phase, the experience of the developers, software knowledge of the founders, and level of employee growth are some of the organizational factors that influence the intentional accumulation of technical debt. In addition, we find the software startups are typically driven to achieve a "good enough level," and this guides the amount of technical debt that they intentionally accumulate to balance the benefits of speed to market and reduced resources with the challenges of later addressing technical debt.**

Keywords— Technical Debt, Startup, Software development

I. INTRODUCTION

Software startups are freshly created companies with no operating history and mainly oriented towards developing high-tech and innovative products, aiming to grow their business in highly scalable markets [18], [10]. Startups often operate with limited resources and under extreme time pressure as they strive to produce their product and avoid being beaten to market by a competitor or running out of capital [19]. Thus, startups typically develop early software versions to test and validate emerging ideas to avoid wasteful implementation of complicated software which may be unsuccessful in the markets [26]. Under these conditions, often the extra effort required to design and implement software with an optimal design is considered an unaffordable luxury and a potential waste of time and effort.

Software companies often make sub-optimal design decisions to allow them to get to market quickly [19]. For instance, the product might be built with an inflexible architecture that cannot be easily changed to speed up time-to-market and let the startup put their product in users' hands earlier, get feedback, and evolve it [3]. If and when the developed software becomes successful on the market, then the pressure turns modifying the software to meet the user needs (i.e., adding new features). This can cause startups to build upon the original inflexible architecture that was not designed to last for the long term and is not easily extendable.

The result of this situation is the accrual of what is described as Technical Debt (TD). The TD metaphor was first coined at OOPSLA '92 by Ward Cunningham [8], to describe the need to recognize the potential long-term negative effects of immature code that is made during the software development lifecycle. A recent definition was provided by Avgeriou et al. [4] who define TD as *"In software-intensive systems, technical debt is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible. Technical debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability".*

TD has been the focus of much recent research, but this research has been mostly focused on mature software companies, where large amount of TD is considered to be detrimental to the long-term success of software development [24]. However, deliberately accumulating TD could be much more beneficial since it can considerably speed up time-to-market, allowing them to release their product to end-users faster, get feedback, evolve the software, and preserve capital [14]. However, TD must be managed to ensure it is addressed at an appropriate time; unmanaged TD can have negative consequences, such as the death of the startup itself [7].

There is a current paucity of empirical research focusing specifically on TD and startups [25]. This paper reports on a qualitative study that examines the organizational factors that influence the introduction of TD and the benefits and challenges of deliberate taking on TD. Through interviews with 16 professionals at seven different startups, we identified six organization factors that lead to TD. In addition, we present a list of benefits and challenges of TD in startups, which can be considered by practitioners to aid them in the TD decisions.

The remainder of this paper is structured as follows: In Section II we describe the background and related work. Our research methods are described in Section III. We describe the cases in Section IV. The results are presented in Section V. Finally, we discuss the implications and limitations of our work in Section VI, and offer a brief conclusion in Section VII.

II. Background and Related Work

In this section, we provide a complete description of a software startup, provide some background on the startup lifecycle, and review related work on TD in startups.

A. Software Startups: A Definition

Giardino et al. [10] define software startups as those "*organizations focused on the creation of high-tech and innovative products, with little or no operating history, aiming to aggressively grow their business in highly scalable markets*". Sutton [23] presents different characteristics that reflect both engineering and business concerns, which software startup companies must operate within. Software startups are relatively young and inexperienced compared to more established and mature development organizations, and they commonly have very little accumulated experience or history. Typically, their resources are limited, and they primarily focus on getting the product out, promoting the product, and building up strategic alliances. Their business is dependent on influences from various sources, such as investors, customers, partners, and competitors. The software these startup companies are developing are commonly technologically innovative products, and their developing often involves cutting-edge development tools and techniques [23].

B. Software Startups Life Cycle

Crowne [9] identified four distinct stages for a software startup: startup, stabilization, growth, and maturity. Each stage has different types of critical product development issues that potentially can lead to company failure. The first "*Startup*" phase refers to the period between product idea and the first sale. This stage is characterized by a product where the product doesn't meet the customer's requirements and is unreliable and fails frequently. Rectifying defects takes longer than expected and often creates additional defects [9]. The second "*stabilization*" phase begins when the first customer takes delivery of the product and ends when the product is stable enough to be commissioned without any overhead on product development. During this stage, a divide between developers can be spotted, where the developers who join the company early, and those who are recruited later differ in terms of that the early developers mount significant resistance to organizational change. During this stage, the non-functional requirements such as security, reliability, scalability, and performance gain additional attention, and the result of the previously introduced sub-optimal solutions becomes evident [9]. The third "*growth*" phase takes place when the product can be commissioned for new customers without creating any overhead on the development team. This phase ends when market size, share, and growth rate have been established, and all business processes necessary to support product development and sales are in place. In this stage, new features implementation requires a coordinated program of activities across functional areas including product development, professional services, support, and sales and marketing, which stresses the importance of having a repeatable process for software development implementation. The last "*maturity*" stage occurs when the company has evolved from a startup into a mature organization, where, e.g., market size, share, and growth rate have been established In this stage also all processes necessary to support product development and sales are in place [9].

C. Startups and Technical debt

There is a lack of research studies on TD management in software startups [25]. Giardino et al. [10], conducted an empirical study addressing how startups employ software development strategies, using a Greenfield Startup Model (GSM), which also covers startups and TD to some extent. Giardino et al. describe that to be faster, startups may introduce TD as an investment, whose repayment may never come due, with the long-term negative effects on morale, productivity, and product quality. Further, in their study they state that "*Startups achieve high development speed by radically ignoring aspects related to documentation, structures, and processes*", and that "*instead of traditional requirement engineering activities, startups make use of informal specification of functionalities through ticket-based tools to manage low-precision lists of features to implement, written in the form of self-explanatory user stories*".

Gralha et al. [21] investigated the evolution of requirements practices of software startups. They found that TD is one of the six factors that influence the requirements practices of a startup. They identified three phases regarding the accumulation of TD in startups. They also identified trigger points that cause startups to transition from one phase to the next. An increase in the number of employees and software features causes startups to transition from simply knowing and accepting TD to tracking and recording it. Then, when their client retention rate goes down, or they begin to see an increase in negative feedback, they begin to manage and control TD.

Another study which to some extent covers TD in startups is presented by Yli-Huumo et al. [28]. In that study, they investigate the relationship between business model experimentation and TD, with the goal of understanding if conducting these types of experimentations have any effect on the amount of TD occurring during the software life cycle. The concept of a business model experimentation in their study refers to when a company uses the technique to validate assumptions made on a product from real customers before the actual product is created. An example of this can be illustrated when a Minimum Viable Product (MVP) is used to test the business model by collecting and measuring customer feedback [28]. Since adopting the technique of business model experimentation is a conventional approach in both startups and larger companies [28], this study is somewhat related to ours. The result of their research showed that there is a relationship between business model experimentation and the occurrence of TD and also that focusing too much on business model experimentation and not on remediation of TD can have consequences to the product quality.

In a recent study by Klotins et al. [13], where the authors explore how startups estimate TD, the precedents for accumulating TD, and to what extent startups experience outcomes associated with TD, it was found that TD peaks at the growth stage and that the number of people in a team amplifies precedents for TD and finally that there is an association between a startup outcome and their TD management strategy.

Unterkalmsteiner et al.'s [25] research agenda for software startups states that researchers must build a more comprehensive, empirical knowledge base to support forthcoming software startups. They list several research question related to TD, and by answering these questions, they state that it could help clarify the role of design decisions in software development in the context of a software product roadmap, similarly to what happens in other engineering disciplines. The overall goal of the research questions listed by Unterkalmsteiner et al. [25] address in what way practitioners will be able to make better decisions considering the characteristics of the current software product implementation.

III. Research Methodology

The goal of this study is to understand how software startups reason about TD. In particular, we are interested in the organizational factors that impact TD together with the potential benefits and challenges of TD. We, therefore, aim at answering the following research questions:

RQ1: What organizational factors influence the accumulation of TD in software startups?

RQ2: What are the challenges and benefits of Technical Debt for software startups?

In order to answer these research questions, we investigated the strategy of software development in different software startup companies by interviewing 16 practitioners in seven different startup companies, working in seven different areas.

A. Participants

We collected data from software professionals active in seven different software startup companies, shown in TABLE I. The sample population was selected using a non-probability sampling technique [27], where the selection of participant companies was obtained using convenience sampling. The startup companies were located in two different countries. The companies are described in more detail in Section IV.

B. Data Collection

Initially, we ran two workshops (one in each country) with participants from four different startups (A, B, C, and D). The workshops included both a presentation made by one of the authors about TD, followed by a group discussion where the participants explored their own experiences with TD within their startup companies. Each workshop lasted about 120 minutes and in total 12 practitioners from the investigated startup companies participated.

TABLE I. Study Participants

Role	Company	Country	Segment
Developer	A	Sweden	Sport
Developer			
Developer	B	Sweden	Energy
Developer			
Developer	C	New Zealand	Retail
CEO / Developer			
Co-founder / Developer			
Co-founder / Developer			
Co-founder / Developer			
CEO	D	New Zealand	Medical
CFO			
COO			
Senior architect			
Advisor (Business and Technology)	E	Sweden	Media
CEO	F	Sweden	Software Development
Chairman of the board	G	Sweden	Mental Health

The goal of these workshops was to introduce the participants to the study, to align and equip them with relevant knowledge about the concept of TD and to gather background and contextual information on each participating startup company in preparation for the following interviews.

We conducted semi-structured (as suggested in [20]), face-to-face interviews with 16 professionals from seven different companies in two different countries. To improve the reliability of collected data at least two of the authors participated in each interview session. Each interview lasted between 60 and 120 minutes and was digitally recorded and transcribed verbatim. The questions were prepared by three of the authors together.

The aim of the interviews was to understand the accumulation and refactoring of TD and what contextual aspects (related to the startup's environment) influenced such accumulation. We started by asking participants to describe their startup company and product and a. We asked follow-ups to learn about the contextual aspects of the startups (inspired by [18]). Next, we asked about TD. Specifically, we asked:

- Describe some critical TD issues.
- Which TD issues were refactored (and when)?
- Which TD issues are planned to be refactored (and when)?
- If TD issues are not planned to be refactored, why not?
- What value did the accumulated TD give the company?
- What cost was (or will be) paid to remove the TD?
- What extra costs were (or will be) paid because of the TD?
- What led to the accumulation of TD?
- What roles, processes, guidelines, and strategies were used for TD?

Finally, to get more insight into the existing TD, we also jointly ran the software SonarQube [2], and AnaConDebt [1] during the interviews. None of the companies previously used these tools, and they were not familiar with the output from the tools in advance. We asked questions on:

- What issues were revealed and were they already known?
- Would it have helped to use the tool (and when)?
- Will you use the tool in the next iterations?

C. Data Analysis

We used thematic analysis [5] to identify, analyze, and report patterns and themes within the interview data. Thematic analysis involves searching across a dataset to find repeated patterns of meaning. The thematic analysis provides a flexible and useful research tool, which offers a detailed, and yet complex account of the collected data.

The thematic analysis was conducted using a six-phase guide. First, the audio-recorded qualitative data collected from interviews were transcribed, and we familiarized ourselves with the data through careful reading of the transcripts. The second step involved the production of initial codes from the data, where we organized the data into meaningful groups. The third phase focused on searching for themes by sorting the different codes into potential themes and collating all the relevant coded data extracts within each identified theme. Each extract of data was assigned to at least one theme and, in many cases, to multiple themes. For example, the citation "*if it [the software from a third-party application] lifts and take off, we can build our own solution*" was coded as "*Third party*" in the theme "Software development Process." To ensure that the coding was performed in a consistent and reliable fashion and in order to triangulate the interpretation of the data and to avoid bias as much as possible, two authors synchronized some of the output of the coding, following guidelines provided by Campbell et al. [6]. The fourth phase focused on the revised set of candidate themes, involving the refinement of those themes. When needed, we revised the themes or created a new theme. The fifth phase focused on identifying the essence of each theme and determining what aspect of the data is captured by each theme. The final phase of the thematic analysis took place when we had a set of fully developed themes, and involved the final analysis and write-up of the publication. We have made a figure illustrating how the codes and the corresponding themes were assigned during the thematic analysis available at https://figshare.com/articles/Thematical_Analysis/6115172.

IV. Description of Cases

In this Section, to provide more context for our study, we describe the companies in more detail. TABLE II. summarizes the seven companies that participated in this study. As can be seen, there is diversity across all aspects. We also indicate the startup stage for each company (using the stages in Crown's [9] classification of startups, which we described in Section II.B). Across the seven cases, all stages are represented by at least one of the cases in this study.

Figure 1 shows how TD was accumulated or addressed in each stage. All companies reported accumulating significant TD in the startup phase. Surprisingly, two companies reported undertaking either a major refactoring or a complete redesign during the startup phase prior to securing their first customer. Both of these cases were due to unintentional issues with the code or the design. During the stabilization phase, most companies reported addressing the TD that accumulated in the previous stage either by taking on formal refactoring initiatives or by informally removing TD as needed. The two companies in the growth and maturity stages indicated that most of the TD had been addressed before entering these stages. Only two of the companies, C and F, had not yet performed a large refactoring or redesign, but both planned this for the future.

TABLE II. Description of Cases

Company	Product	Domain	Years since founding	Founders SW Knowledge	Software developed	Current Employees	Experience of Software Developers	Development Practices
A	Mobile app	Sport	2.5	None	Initially external then in-house	Founder, CTO, CMO, 3 developers, one salesperson	2 junior, 1 senior + senior CTO	Some agile practices (e.g. sprint planning)
B	Mobile and web apps	Energy	6	High	In-house	CEO, 5 developers, two sale reps	4 senior, 1 junior	Scrum
C	Web app	Retail	2	High	In-house	4 Founders	All junior	No formal process
D	Web app	Medical	2	None	In-house	3 Founders, 2 Technical staff	All senior	Some agile practices (e.g. Kanban, CI)
E	SaaS app	Media	9*	Low	In-house	35 employees (Two-thirds are developers)	All junior	Some agile practices
F	Web app	Software	2	High	Combination in-house and consultant	Founder + consultant as needed	Senior	Scrum
G	Mobile app	Mental Health	6	None	Initially external then in-house	Founder, CTO, 3 developers, 1 salesperson	3 junior + senior CTO	Scrum

* Today this startup is 9 years old, but the data collected for this startup reflects a time period of 3-5 years after they were founded

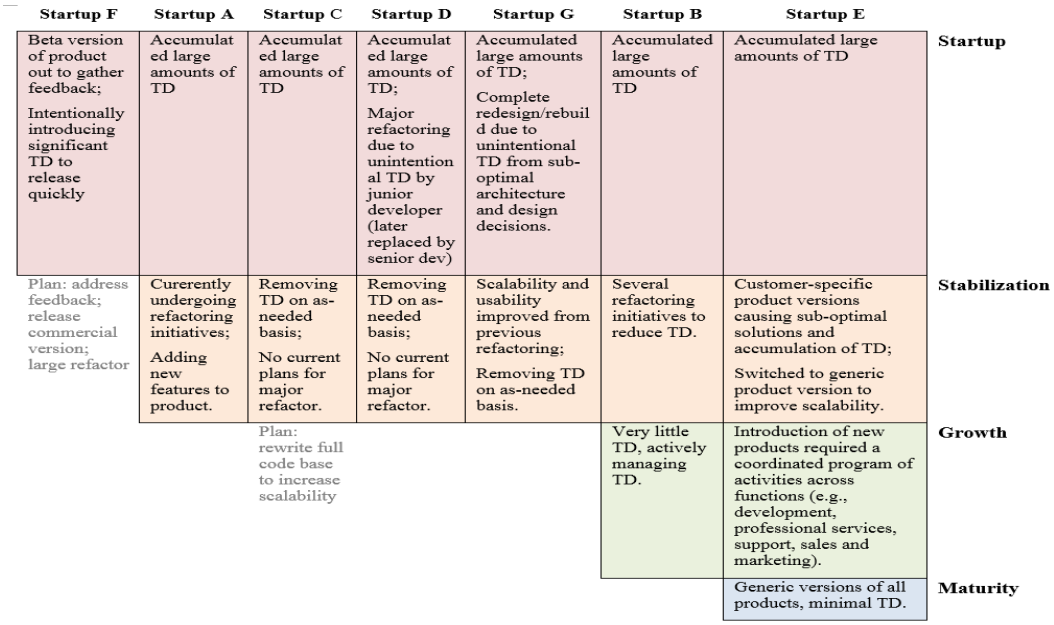

Fig. 1. Overview of TD strategies across Crowne's [9] stages for each Startup.

V. Results

The following subsections present results for the research questions presented in Section III and the results are grouped according to each research question.

A. What organizational factors influence the accumulation of TD in software startups? (RQ1)

Our analysis has identified many factors that influenced the amount of TD that the startups accumulated.

1) Experience of software developers

Our results indicate that the experience level of the software developers can have both positive and negative influence on the accumulation of TD. As startups are typically very small in terms of number of developers initially, the experience level of individual developers can be impactful.

Less experienced (junior) developers often unintentionally accumulate TD due to their lack of experience. As one interviewee from Company A stated, "*It's really good to have at least one guy that is more experience in the team*." Another interviewee from Company E explained this as: *"Junior developer are less able to project outcome to the future about how the system is likely to evolve, which means that they have a tendency to focus on the 'here and now', and solve the today's requirement whereas people that are experienced can often predict a little bit more easily what is likely to come in the future and already start to prepare the system for that."* Thus, junior developers are more likely to introduce unintentional TD due to their lack of experience.

More experienced (senior) software developers are more aware of and have accumulated more experience about the effect of introducing TD, compared to junior developers. Thus, having senior developers to guide the development is very beneficial. However, senior developers are more expensive, and startups typically cannot afford to have many senior developers. "*I think that it would be very expensive to get another very experienced person. And maybe it's not worth it."*

In addition to high salary costs, senior developers may be less likely to intentionally accumulate TD if they have experience working on more mature software products that are not under such extreme time pressures to get to market. A participant from Company D stated, "*If we had had the knowledge or the insight, we probably would have taken on board technical debt earlier on, but I think because we ended up hiring senior developers that were used to working in certain ways with testing and re-testing everything. They ended up building, a fairly robust, as far as we can tell, but for our purposes, there might have been something over-engineered perhaps.*" Senior developers may be less willing to operate in an unstructured and less quality oriented approach. For example, one interviewee from Company A said: "*So, you need to be more flexible, and if you are senior maybe you aren't ready to cope with that."* This could cause startups delays in getting to market if TD is always avoided in favor of producing high quality software.

2) Software knowledge of startup founders

We found that the knowledge of the founders, related to software development, has an impact on how TD is accumulated. Founders with limited software development knowledge are less likely to accumulate TD intentionally. Since they are unable to implement the product themselves, they are likely to employ an external consultancy company or hire in-house developers to implement the first software solution, which involves a significant investment prior to being able to receive revenue from the software. The founders typically expect a high-quality implementation in return for this investment since they tend to have no knowledge about the benefits of TD.

On the other hand, when the startup founders are experienced software developers, they are more likely to implement the product on their own. They often accumulate a large amount of TD because they focus on producing the first release quickly. They view the initial release as more expendable since they have not invested money towards its development (despite having invested their time).

3) Employee growth

We found that when startup teams were remaining stable in terms of the number of developers, they did not feel a need to reduce their TD since the issues related to the TD affected only the developers, not the customers. The participants did not believe their TD impacted product performance or usability. While the TD did make the code more difficult to extend or modify, the existing developers were already familiar with the TD in the code, so it was not necessary to reduce the TD.

However, the addition of new developers caused the TD to decrease for several reasons. First, the *existing developers reduce the technical debt* prior to hiring new developers. The developers want the code to be easier to understand so that new developers can be onboarded more quickly. They also do not want new developers to unintentionally introduce additional technical debt because they are modeling their own code on existing TD. For example, an interviewee at company B stated: *"But as time goes on, the quality of real code, or its readability and how easy it is to work with, becomes more and more important. It is very easy when you as a developer comes into a project that you start writing code in the way of the existing code base. You kind of go 'oh, this is how they do it here,' and that is not always a positive thing. A lot of time that is quite a negative thing, because, you slip into those habits and before you know it, all the things that you personally hold true about what good code is, you are not doing that anymore".* This fear of duplicating TD was also described by one interviewee from Company A stating: *"And if you come in as a new developer, you might copy-paste some code, and you copy-paste that old thing of doing it, and we get the more messy code. And that is what we don't want."*

In addition to the existing developers purposely reducing TD, *new developers also remove TD* as it is difficult to extend. The existing developers may be so familiar with the code, that they no longer notice the problems, while they will be more obvious to the new developers. For example, a developer from Company D said *"I mean there's a big refactor when they brought me on. …[we] ended up throwing a lot of code out and rewriting it. And that was probably because of the technical debt side of things in there, using constants throughout and the like."* Our results corroborates to some extent the results found by Klotins et al. [13] stating that "increase in team size is also associated with outcomes of technical debt".

4) Uncertainty

In general, uncertainty about the future of the organization and product is very common characteristic in the startup companies. Our results suggest that, not surprisingly, the uncertainty plays a major role when making decisions about TD. One of the interviewees from Company C put this as "*with these sorts of projects, you need to build a business case, and you'd be silly to like build something with no technical debt in it until you've at least proven that it's something you have to pay for. As soon as we confirm that there will be [revenue], and see the money starting to come in, that's when you probably start to look at the repaying the technical debt*". Another participant from Company D stated "*there was a point where basically we said, okay, now we just need to stop spending money because we don't know if this is even going to be a viable project and if it's going to generate any money or anybody's going to want to buy it*". This uncertainty causes startups to accumulate significant TD so they can release a proof-of-concept as quickly as possible. Once their idea is validated and they have a number of paying clients, they can worry about paying off their TD – possibly be rewriting the entire codebase from scratch.

5) Lack of development process

None of the interviewed startup companies adopted a systematic software development process, and the need of having such a process was not considered by the interviewees to be important during the first phases in the startups' life-cycle. However, this topic was brought up as a challenge, especially when the startup grows and hires more developers. A lack of processes for the management, identification, and prioritization of TD means that TD decisions are often made ad hoc, and there are no consistent decisions being made across the team. This is especially important as the team grows to ensure there is conformity. As one interviewee in company A said: *"Multiple ways of doing things, are spreading at the same time… I mean, it is quite important for me, when we start to grow, that we have the same way of writing code."*

6) Autonomy of developers (related to TD)

Related to the lack of development process, developers often have full autonomy to decide when to take on TD and plan when to refactor the TD. Developers typically do not discuss TD-related decisions with others. While this allows for flexible work and short decision paths, it means developers, who are often not financially invested in the project, are making very important decisions without possibly considering the financial repercussions of these decisions.

This can be especially problematic when employing external software consultancies since decisions tend to made based on the benefits to the consultancy company, rather than making the best decision for the software product under development. The consultancy could decide to minimize TD because they want to maintain a high-quality reputation for their company and do not want to deliver software that is not maintainable. If the development is not on a fixed price contract, this desire for perfection could cost the startup significant time and money. On the other hand, they may be driven to take on significant TD since they know they do not need to maintain the software and they are driven by the desire to save money during the development. For example, the interviewee from Company G stated: "*the externally hired consultants, they just did what was asked of them in their contract, with the lowest possible development effort. That is commonly how it works with externally hired developers, they do not really care about Technical Debt, they care about delivering the software according to the given specification*

they are paid for." We saw only one case where developers were not given full autonomy regarding TD decisions. The founders of this company found being involved in even trivial implementation decisions very useful. One of the founders of Company D said "*I think that they got used to basically involving us in their decision-making even though on a relatively trivial scale so that they'd ask about everything… And then we could understand and be involved in making those decisions about, how much debt and things will take on, even though we didn't call it debt. And there was a point probably about two-thirds of the way through the project where 'cause we'd often get updates on estimates of hours required to complete certain tasks so we'd keep an eye on how much money we were spending."*

TABLE III. ORGANIZATIONAL FACTORS INFLUENCING TD IN STARTUPS

Factor	Level	TD	Reason
Experience of developers	low (junior)	increases	poor design decisions (unintentional)
	high (senior)	increases	developers aware of benefits of TD (intentional)
		decreases	developers accustomed to producing high quality software
Software knowledge of founders	low	decreases	founders unaware of TD benefits; large investment for developers causes desire for high-quality
	high	increases	founders develop product themselves; code seen as expendable
Employee growth	stable	stable	devs already familiar with code (and its TD); no impact to customer
	increasing	decreases	existing devs refactor to make onboarding easier
			existing devs refactor to prevent a culture of "bad" code
			new devs refactor because code not readable
Uncertainty	high	increases	goal: reduce dev time and cost
	decreasing	decreases	TD repaid after market validation
Lack of dev. process	---	varies	ad hoc decisions
Autonomy of developers	high	varies	developers make decisions without any guidance (possible poor business decisions)
	low	varies	strategic decisions made

Answer to RQ1: We identified six organizational factors that influence the accumulation of technical debt: experience of developers, software knowledge of startup founders, employee growth, uncertainty, lack of development process, and the autonomy of developers regarding technical debt decisions. The results are summarized in TABLE III.

B. *What are the challenges and benefits of deliberately introducing Technical Debt for software startups? (RQ2)*

In this section, we explore how software startups determine and reason about both the challenges and benefits of intentionally introducing TD. In general, startup companies deliberately introducing TD, have a positive attitude of doing that. They are also relatively aware of the harmful effects these decisions can have on the future software in terms of impeding innovation and expansion of their software systems.

1) *Benefits of intentional technical debt*

We identified many benefits of intentionally introducing TD in software startups.

Cutting development time in order to be able to release the product as quickly as possible is seen as a large benefit for startups. Getting to market quickly can:

- enable ***fast feedback*** from the customers. An interviewee in Company A said: "*We prefer to cut some corners to improve the speed, and get something out instead of making it more mature directly" …."It is more important to get to the market fast and get feedback from the users, then to focus on avoiding TD, taking on TD is ok*."
- ***increase revenue***. One of Company C's founders said, "*Yeah, we probably wouldn't have got the contract earlier, right… Then we wouldn't have the capital*". Another participant from Company A said "*we are a startup, and we need to make money. We need to get things working, but they don't need to be perfect".*

Another benefit is the **preservation of startup capital** since commonly startup companies have less money in the early stages. A participant from Company D stated, *"it's just that we had to get the code out the door. And we had to get it so that we could afford it."* Another participant from Company F said "*by taking the first technical debt, we spent 10% of what we would have spent if we would have done the whole product without TD."*

Related to saving money and time, another benefit is the **decreased risk**. Since the startups involve uncertainty, it is sometimes wise to invest as little money and time as possible prior to validating the idea through evaluation of the product. A participant from Company F said: "*In case the product would turn out to be a failure, we would have saved 90% of the money…we avoided a big risk, and we reduced uncertainty thanks to technical debt. It was a great decision, I think."*

Intentional TD also allows startups to stay flexible. When they do not spend large amounts of money or time developing new features, they are more willing to discard them and alter the product significantly when needed. Thus, the TD allows them to **make more objective decisions**. *"If you put too much time and effort in there, it could be harder to throw it away in the next version. So, I think it's not always bad that you don't do the best".*

2) *Challenges of intentional technical debt*

Despite the benefits of intentional TD, we also identified challenges since the sub-optimal solutions would eventually need to be fixed. The two companies who initially hired an external consultancy company to implement the first software solution failed in doing so. Most of the initial implementation was later removed and replaced by in-house developers, causing significant delays and additional expenditures. In such extreme cases, TD can cause the **product failure** or a **business disruption**. Another challenge of TD is the **reduced scalability** it often introduces. *"If you validated it and it's*

looking good, you wanna be able to put your foot on the gas and go quickly and scale. And if the architecture's not ready…" A first, light and sub-optimal solution may only work in a specific setting but will need to be refactored in order to scale the software. One developer from Company A put it "*growing is not just like taking what we have and do the exact same thing because that will only scale to a specific limit…There was no segmentation of the code in any part. We started to split the code up, we started to segment and to separate the code, so that we also can scale different part of the code."*

The interviewees mentioned different TD types such as architectural, infrastructural and source code related TD as having a substantial negative impact on the system growth. Another challenge is that the harmful effects of TD increases in severity as the software grows and when more developers were involved in the development process. Thus, the introduction of TD can have **compounding effects** on the development time and resources, since it will take more time to develop code on top of existing TD. Then, if the TD is removed later, it all of the code built on top of the TD will also potentially be impacted. As one interviewee at Company B put it: "*In a greenfield project, I think there is an argument hacking together something that works quickly. But as time goes on, the quality of real code, or its readability and then how easy it is to work with, it becomes more and more important."* Another challenge is that fixing TD could **increase risk**. When fixing TD, it might create new bugs in the code, adding to the amount of future work that needs to be done. *"The bugs will probably grow, especially if we try and fix it, spend time trying to fix it."*

Finally, the introduction of TD requires the loss **of productivity to be managed later**. We found that during the early phases, startups rarely manage their TD and decisions are often made on an ad hoc basis and none of the interviewed startups used any software tools assisting their TD management strategy. In order to understand if the startups would consider using tools as beneficial, we jointly run both SonarQube and AnaConDebt on four of the startups' software (A, B, C, and D). After running the tools we went through the output and assessed whether the result was perceived as useful or not. All the startups using SonarQube found it specifically valuable identifying specific areas within their codebase that could further be improved in terms of refactoring initiatives of TD. As the founder from Company D said, "*I think this is very useful in terms of prioritizing the back end of what we have and what we need to sort of like work on*."

The result of running AnaConDebt provided the startups with estimates on the TD principal and interest and also the growth of them with respect to different future scenarios, was also unanimous perceived as valuable to the startups' TD management strategy. However, using these kind of tools was not a considered as a good choice during the first startup phase since it would have distracted the developers from being fast with the first product release. The output from running the tools cannot be reported due to confidentiality reasons.

3) Good Enough Level

When startup companies deliberately introduce TD, they implicitly decide what a Good Enough Level (GEL) of the software quality is and what amount of TD is acceptable to take on. They weigh the benefits and challenges of the TD when making their decisions (illustrated in 1). However, it is not usually an easy decision. A founder of Company D said "*It's difficult to balance where you're constantly making decisions how do we balance what we're spending on this, versus the likelihood of producing these results*."

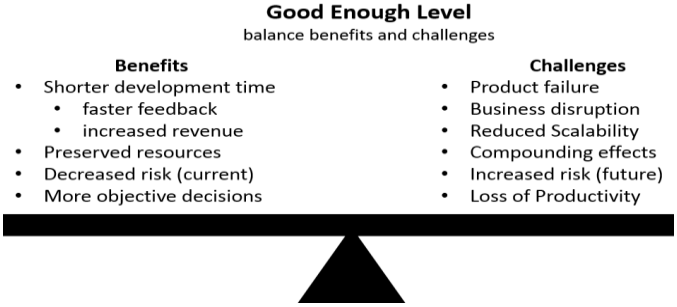

Fig. 2. Good Enough Level is achieved by considering the ideal balance between the benefits and challenges associated with intentional TD.

Answer to RQ2: Intentionally introducing technical debt allows startups to cut development time, enabling faster feedback and increased revenue, preserve their resources, decrease risk, and make more objective decisions. However, the technical debt causes reduced scalability, becomes more severe as the product grows, and introduces future development risks. Thus, deliberately introducing technical debt brings both benefits and challenges and startups must weigh these to determine a "*Good Enough Level".*

VI. DISCUSSION

In this section, we discuss recommendations for startups, compare our results to existing knowledge on accumulation and refactoring of TD in other contexts, and describe the limitations of this study.

A. Recommendations for software startups

Based on the finding related to the organizational factors that influence TD in startups and the benefits and challenges associated with TD, we have the following recommendations for startups.

Balanced experience levels (of developers) needed. We found that junior developers often introduce unintentional TD. Senior developers are often more calculated in their TD decisions. However, senior developers may be less risk adverse if they have more experience working on more structured, mature products where quality is paramount. A mix of both senior and junior developers seems ideal to find the right balance between TD and quality. These results are in line with the ideas of Crown [9], who states that "*The principal developer for the company must be highly experienced, and familiar with all aspects of software engineering practice. This person must also be an accomplished technical leader, as they will need to influence their less experienced colleagues".* Though, we advocate that junior developers are equally important.

Unbiased technical advisors needed. When the startup founders do not have software development knowledge, those implementing the software are likely to make decisions that benefit their own needs, rather than the startup company. For example, they may cut corners to save their own time, or they may gold plate the software to build up their own reputation (and to increase their own revenue). Thus, startup founders who lack software development expertise should consider seeking technical guidance from someone other than the company or developers they hire to implement the solution so they can obtain unbiased advice related to TD decisions. Depending on the stage of the startup (and the available capital), this advice could be obtained by the introduction of a CTO or from an external consultant.

Consider "contagiousness" of TD in prioritization. We found that TD is often removed as the number of developers increases. This is in line with the results of Gralha et al. [17]. We found there are various reasons for this decrease in TD. One of which is the removal of TD that could be "contagious" – new developers may model their code off existing TD or may directly duplicate poorly written code. Thus, in addition to prioritizing TD that might block key features planned in the upcoming iterations [8], contagious TD [16] should also be prioritized, especially during times of growth in the development team. If such TD is not removed, it can generate new TD in a vicious spiral, reducing the growth time and compromising the software quality [22], [12] and culture of the startup in the future.

Encourage autonomy with high-level guidance. We found that in most startups, developers make TD-related decisions with full autonomy. Thus, they could possibly be making poor business decisions without considering the strategic repercussions of their decisions. Providing overall guidance to the developers, so they know what level and types of TD are appropriate can mitigate this risk, while still maintaining developer autonomy.

B. Strategy to balance TD over time

Startups need to balance several factors affecting the accumulation of TD, to reach a Good Enough Level. However, how do startups do this over time? We report, in Fig. 32, a first interpretation that helps to understand the strategy adopted by the studied cases in different phases.

Fig. 2 shows the accumulation of TD with respect to each startup phase and key events. The black line suggests the accumulation of Technical Debt that has been preferred by the studied startups. We also show GELs ("Good Enough Level"), or else thresholds under which TD needs to be kept via strategic refactorings, otherwise causing possible disruptive events (red lines and crosses). Finally, in the bottom of the picture, we outline which mechanisms have been reported by the participants to be necessary and effective to keep a GEL of TD in a specific phase. In the startup phase, startups recklessly accumulate TD. This has been reported to be not only necessary, but very valuable to quickly satisfy the first customers, to reduce risks and costs. However, too much TD can still be disruptive in the first phase, leading to product failure and business disruption, if the acquired TD prevents the successful delivery of the MVP itself. In particular, the cases report that the domain specific technology needs to be well understood and that the usability of the product should not be overlooked (GEL1). In the stabilization phase, a partial refactoring (Stabilization refactoring) is recommended to reach GEL2. In this case, the TD to be prioritized is the one blocking key features planned in the upcoming iterations for the delivery of the product to key customers. In addition, TD that is judged to be especially contagious (likely to spread to the new features and to be picked up by new developers) should be at least considered. The challenges if the startup fails to keep this level of TD is the difficulty (if not the halt) of evolving the system with new features, with the consequent loss of key customers. Additionally, while entering the growth phase, TD that is accessed by new developers can generate new TD in a vicious spiral, reducing the growth time and compromising the code and culture of the startup in the future. Here the high-level guidance and the experience of the developers are key to keep the right level of TD, but a budget needs to be allocated for the refactoring to reach GEL2. During the growth phase, there is a need to remove some more TD (Growth refactoring) to reach a GEL3. If the contagious debt is not removed in the previous phase, it needs to be removed here before hiring new developers. In addition, the code is optimized to be scalable and to be delivered to several customers in the market: the architecture of the system should be refactored to allow the productive management of customer variability, to reduce the cost of maintenance and operations for the developers, to avoid a loss of productivity. In the growth phase, several other mechanisms can be introduced to not only reduce the current TD, but also to prevent the accumulation of future TD (e.g. tools, processes). TD needs to be well communicated in order to make business decisions. In their maturity phase, startups seem to start behaving like mature companies. However, in this study, we do not have enough cases to report common practices related to this phase.

C. Comparison of TD Management with non-Startups

Looking at the current literature, we can see some differences with how startups accumulate and refactor TD, compared to large and more mature organizations. Some large and mature organizations might have internal innovation projects that have a more similar context to startups or might have high turnover of junior developers. Since we did not find studies on such context and TD, such cases are excluded from the following analysis and will require additional studies. In both startups and mature organizations, there is often a peak of accumulated TD at the beginning of feature development [17]. However, in mature organizations, there is usually a defined quality threshold, in the form of the desired software architecture or other quality models. In such cases, TD is referred to the divergence from such desired thresholds. Such reference points do not seem to exist in startups. Consequently, they tend to accumulate more TD, which is also considered a benefit. There is, naturally, some level of uncertainty in both startups and mature organizations at the start of a new project. However, the uncertainty in young startup companies is greater than in a mature company [11]. Thus, taking on a right amount of TD seems to be a well-established strategy to deal with the high levels of uncertainty. Another difference can be found on

how inexperienced developers are considered in startups and mature companies. Inexperienced developers seems to be considered as less aware of the long-term effects of TD, which consequently leads them to be keener to accumulate it.

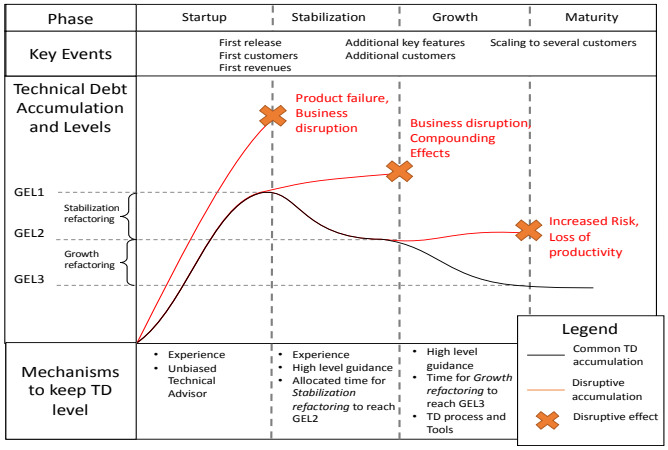

Fig. 3. TD balanced differently in different startup phases

This choice seems to fit with the importance to accrue TD in startups. However, as we have seen in all the analyzed cases, an experienced developer (technical lead or CTO) is crucial in the startup team to keep the TD level to desired thresholds. In contrast, in mature organizations, it is preferred to have team members that have a higher understanding of TD and to make sure that TD is not accumulated [15]. One of the main reasons is that code developed by mature organizations, especially in large projects, is continuously integrated with a large codebase and needs to be available and reliable for other teams' work. In other words, TD has a bigger impact. Such impact is not present in the *startup* and *stabilization* phase of startup companies, but comes into play when the startup enters the *growing* phase.

A similar difference can be seen with respect to processes and tools: a recent survey in the large organization [15] highlights how a third of the participants, answering the survey, use tools to track TD. In startups, we could see the complete lack and conscious avoidance of such processes and tools until the company reaches the growing phase. On the other hand, both in startups and partially (2/3 of the participants) in large organizations [15], we notice the lack of knowledge on how to implement such processes and what tools to use to keep TD at bay. Learning how to manage TD seems to be equally important for large companies and for startups entering the growth phase.

In summary, despite some similarities exist regarding TD management between large, mature organizations and startups, the first three startup phases seem to stand out with respect to managing TD. This is due to the level of uncertainty, the environment, and the business context being different. Although this analysis includes a small sample of both startups and large companies, and more studies are needed to corroborate this analysis, we have some initial evidence suggesting that the strategic management of TD in startups might differ from the best practices related to large organizations.

D. *Limitations and Threats to Validity*

The main limitations of this study are related to the limited sample of startups investigated and to the qualitative nature of the investigation. However, these are limitations that can be considered acceptable in light of the exploratory purpose of this study. We preferred to gain a deep and rich understanding of the context of a few cases to build a holistic first theory rather than surveying the topic on a high level only.

Specific threats to validity include construct validity related to the concept of TD, external validity with respect to the limited contexts analyzed, and reliability of the results affected by the high level of interpretation that both interviewees and researchers might have been injected in the study [20].

To mitigate construct validity, we held a workshop with several of the participants in the startups to clearly define and align on what TD was. We gave concrete examples, we used the up to date definition of TD reported in the Dagstuhl seminar [4], and we asked the participants to share examples in order to test if their understanding matched the community's definition. Additionally, when asking questions, we have always asked and probed the claims by inquiring for additional concrete examples.

To mitigate the external validity threat, we collected information from two different countries in different geographical areas. In addition, the case companies represent different segments, and we interviewed different roles, from developers to CTOs to CEOs, to external advisors.

Although we do not claim to provide fully generalizable results in this exploratory study, we have aimed at maximizing the coverage of our cases. Furthermore, we plan to expand our sample in the future, to reach a higher degree of validation of our results. Reliability threats were mitigated by assuring that two researchers were always present when conducting interviews, that one of the researchers was always attending all workshops and interviews for consistency purposes, and that the analysis was organized in two groups where researchers analyzed the codes separately and then merged the findings. In other words, we made sure that different observers were contributing in different phases of the data collection and analysis, reducing the bias of single researchers.

VII. Conclusion

This exploratory study set out to provide a first understanding of how software startups reason about TD. Through interviews with 16 software professionals in seven different startup companies, we identified six organizational factors that influence the accumulation of TD in software startups (experience of developers, software knowledge of startup founders, employee growth, uncertainty, lack of development process, and the autonomy of developers regarding TD decisions). We also found that startups must strive towards a *Good Enough Level*, over time, for their product, while weighing the benefits and challenges associated with taking on TD. This study provides a set of recommendations and a first strategy which can be used by software startups to support their decisions related to the accumulation and refactoring of TD.

REFERENCES

[1] https://anacondebt.com/.
[2] https://www.sonarqube.org/.
[3] P. Avgeriou, P. Kruchten, R. L. Nord, I. Ozkaya, and C. Seaman, "Reducing friction in software development," IEEE Software, vol. 33, no. 1, 2016, pp. 66-72.
[4] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162)," Dagstuhl Reports, vol. 6, no. 4, 2016, pp. 110-138.
[5] V. Braun and V. Clarke, "Using thematic analysis in psychology, Qualitative research in psychology, 3(2)," 2006, pp. 77-101.
[6] J. L. Campbell, C. Quincy, J. Osserman, and O. K. Pedersen, "Coding In-depth Semistructured Interviews Problems of Unitization and Intercoder Reliability and Agreement," Sociological Methods & Research, 2013.
[7] M. Chicote, "Startups and Technical Debt: Managing Technical Debt with Visual Thinking," in 2017 IEEE/ACM 1st International Workshop on Software Engineering for Startups (SoftStart), 2017, pp. 10-11.
[8] Z. Codabux and B. Williams, "Managing technical debt: an industrial case study," presented at the Proceedings of the 4th International Workshop on Managing Technical Debt, San Francisco, California, 2013.
[9] M. Crowne, "Why software product startups fail and what to do about it. Evolution of software product development in startup companies," in IEEE International Engineering Management Conference, 2002, pp. 338-343 vol.1.
[10] C. Giardino, N. Paternoster, M. Unterkalmsteiner, T. Gorschek, and P. Abrahamsson, "Software Development in Startup Companies: The Greenfield Startup Model," IEEE Transactions on Software Engineering, vol. 42, no. 6, 2016, pp. 585-604.
[11] P. A. Gompers, "Grandstanding in the venture capital industry," Journal of Financial Economics, vol. 42, no. 1, 1996/09/01/, 1996, pp. 133-156.
[12] Y. Guo, R. Spínola, and C. Seaman, "Exploring the costs of technical debt management – a case study," Empirical Software Engineering, 2014/11/30, 2014, pp. 1-24.
[13] E. Klotins, M. Unterkalmsteiner, P. Chatzipetrou, T. Gorschek, R. Prikladnicki, N. Tripathi, *et al.*, "Exploration of Technical Debt in Start-ups," in ACM/IEEE 40th International Conference on Software Engineering: Software Engineering in Practice, Gothenburg, Sweden, 2018.
[14] P. Kruchten, R. L. Nord, I. Ozkaya, and D. Falessi, "Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt," SIGSOFT Softw. Eng. Notes, vol. 38, no. 5, 2013, pp. 51-54.
[15] A. Martini, T. Besker, and J. Bosch, "Technical debt tracking: Current state of practice a survey and multiple case study in 15 large organizations," Science of Computer Programming, 2018.
[16] A. Martini and J. Bosch, "On the interest of architectural technical debt: Uncovering the contagious debt phenomenon," Journal of Software: Evolution and Process, 2017.
[17] A. Martini, J. Bosch, and M. Chaudron, "Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study," Information and Software Technology, vol. 67, 2015, pp. 237-253.
[18] N. Paternoster, C. Giardino, M. Unterkalmsteiner, T. Gorschek, and P. Abrahamsson, "Software development in startup companies: A systematic mapping study," Information and Software Technology, vol. 56, no. 10, 2014/10/01/, 2014, pp. 1200-1218.
[19] M. Reddy, "Chapter 4 - Design," in API Design for C++, M. Reddy, Ed., ed Boston: Morgan Kaufmann, 2011, pp. 105-150.
[20] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," Empirical Software Engineering, vol. 14, no. 2, 2009, pp. 131-164.
[21] S. Gralha, D. Damian, A. Wasserman, M. Goulao, and J. Araujo, "The Evolution of Requirements Practices in Software Startups," in International Conference on Software Engineering (ICSE), to appear, 2018.
[22] C. Seaman, Y. Guo, N. Zazworka, F. Shull, C. Izurieta, Y. Cai, *et al.*, "Using technical debt data in decision making: Potential decision approaches," in 2012 Third International Workshop on Managing Technical Debt (MTD), 2012, pp. 45-48.
[23] S. M. Sutton, "The role of process in software start-up," IEEE Software, vol. 17, no. 4, 2000, pp. 33-39.
[24] E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," Journal of Systems and Software, vol. 86, no. 6, 2013, pp. 1498-1516.
[25] M. Unterkalmsteiner, P. Abrahamsson, X. Wang, A. Nguyen-Duc, S. Shah, S. S. Bajwa, *et al.*, "Software Startups – A Research Agenda," e-Informatica Software Engineering Journal, vol. 10, no. 1, 2016, pp. 89–123.
[26] M. Waseem and N. Ikram, "Architecting activities evolution and emergence in agile software development: An empirical investigation initial research proposal," in Lecture Notes in Business Information Processing vol. 251, ed, 2016.
[27] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslen, Experimentation in software engineering: an introduction: Kluwer Academic Publishers, 2000.
[28] J. Yli-Huumo, T. Rissanen, A. Maglyas, K. Smolander, and L.-M. Sainio, "The Relationship Between Business Model Experimentation and Technical Debt," in Software Business, Cham, 2015, pp. 17-29.