Managing Tensions between Architectural Debt and Digital Innovation: The Case of a Financial Organization

Knut H. Rolland
University of Oslo
knutr@ifi.uio.no

Kalle Lyytinen
Case Western Reserve University
kjl13@case.edu

Abstract

In recent years, Information System (IS) scholars have increasingly explored the malleability and re-combinability of digital artifacts that facilitate innovation and change. In this paper, we focus on how architectural debt thwarts evolvability of complex IT architectures and systems founded on them. We conduct a case study in a major Scandinavian financial institution and explore their how they managed architectural debt during fast paced service innovation. Our analysis suggests that the firm's capability to innovate depended on software developer's ability to work across multiple syntactic, semantic, and pragmatic knowledge boundaries whilst addressing architectural debt. The paper offers two contributions. First, we add to the nascent body of socio-technical research on technical debt by illustrating how architectural debt cuts across multiple developer teams and architecture layers making it hard to identify and resolve. Second, we expand studies of digital innovation by identifying two interconnected tensions faced when innovators have to evolve complex IT architectures that lay the foundation for artefact malleability. We tie how the tensions are addressed to the firm's capability to manage architectural debt.

1. Introduction

Information System (IS) scholars have for some time investigated how flexibility and re-combinability form inherent characteristics of digital technologies and lay foundation for continuous innovation [1-5]. More often than not, digital innovation involves embarking on change projects that deploy multiple interconnected legacy systems. It is well known that legacy systems are often technically complex and costly to either extend or discontinue [6]. Moreover, although improvements in software methods and technologies such as agile methods [7], cloud based micro-service architectures (MSA) [8], and platformization [9], have been available for a while, software development as a critical foundation of digital innovation still comes with high levels of risk and complexity – especially in legacy settings. These challenges will also reduce the potential for continued innovation.

One such risk that has remained poorly understood is the treatment of technical debt and its consequences for digital innovation. Technical debt broadly refers to short-cuts, temporary solutions, and architecture anomalies introduced during software development that affect long term evolvability of software and increase its maintenance cost and risk [10-14]. The connection of technical debt comes with trade-offs. On the one hand, taking up technical debt speeds up software development and allows faster launch of novel innovations [11]. On the other hand, technical debt reduces the long term ability to maintain and evolve software with speed and quality [12]. Yet, there are few IS studies that have empirically assessed and theorized on the role of technical debt and how it shapes digital innovation. In particular we know little how prior architectural decisions to organize software influence innovation speed and quality. To address this gap we ask: *How does an organization's capability to manage architectural debt shape digital innovation?*

Extant literature tends to treat technical debt as a local, micro-level phenomenon confined to the quality of the code which has negative consequences to programming effort and quality at team level. In this paper, we posit that in the context of digital innovation a broader view is necessary as during innovation efforts teams need cater also for the effects of architectural level technical debt across applications. They need to consequently understand how the software assets are architected and coordinated at a higher level. Especially, in the context of digital innovation we ask how architectural debt becomes manifested across multiple software modules and layers within a complex legacy IT architecture. Such architectures are typical to all large firms running legacy systems. This kind of technical debt we refer to as architectural debt since it is not located within a specific code in a module. Rather it is mainly manifested in non-functional aspects of a software system that affect its evolvability, maintainability, and scalability [15].

Empirically, we draw on a case study within a major Scandinavian financial organization. This firm has legacy systems while attempting to innovate with

URI: https://hdl.handle.net/10125/71427
978-0-9981331-4-0

agility novel mobile services expected to augment its extensive legacy online services. We show that this development effort calls for significant management of architectural debt and the project's capabilities in managing such debt has significant consequences for its ability to deliver services in speed and quality. The case study indicates that architectural debt will slow down innovation speed and thwarts business development. In particular, we show that architectural debt is initially invisible and incomprehensible for teams but its level escalates and becomes visible when the development advances and new debt is added. The management of architectural debt now depends on involved actors' capability to work across multiple syntactic, semantic and pragmatic 'knowledge boundaries' [16-17] created by modules and architecture layers. For these actors it is necessary to detect, resolve, and avoid taking up more architectural debt. The analysis suggests that while most technical debt is essentially a technical phenomenon, its sources, detection, resolution, and uptake are all socio-technical processes located at multiple levels of the IT and business functions.

Theoretically, our study contributes to nascent literature on technical debt and its organizational consequences. It identifies and describes two tensions involved in managing architectural debt across applications. The first one is concerned with the tension between increasingly decentralized IS development required by agile teams and these teams' capabilities to manage architectural debt across applications. The second tension is concerned with the need for high speed innovation and the long term capabilities to manage architectural debt. We also show how these tensions depend on one another and mutually contribute to each other. We conclude by offering advice to practitioners on how to manage technical debt between applications' that invite to cut across complex knowledge boundaries.

2. Relevant literature and theory

2.1 From technical to architectural debt

The concept of "technical debt" has been used by practitioners and academics alike for some time to denote sub-optimal technical solutions expressed in code. The notion originated in software engineering in the early 1990s and therein referred to coding practices that introduced short-cuts or poor designs with the intent to speed up implementation while simultaneously increasing the life-cycle cost of the software [11, 15, 18-19]. Ignorant coding and taking short-cuts were deemed equivalent to taking up financial debt that had to be repaid with interest later. Typically, legacy software has considerable technical debt [20]. One reason is the longevity of software: "*software systems rarely die. Instead, each new version forms a platform upon which subsequent versions are built. With this approach, today's developers bear the consequences of all design decisions made in the past*" [21: p.170].

In recent years, the research on technical debt has expanded to encompass multiple aspects of software development. Alves et al. [19] developed a taxonomy of technical debt, and identified 15 categories of debt ranging from code debt to documentation debt. Although all these types of debt exercise influence on cost, a specific type of digital debt coined "architectural debt" should be of special interest for IS scholars examining interactions between software and its organizational environment. This debt has the potential to reduce the evolvability of software artifacts it refers to "*the problems encountered in product architecture, for example, violation of modularity, which can affect architectural requirements (performance, robustness, among others)*" [18: p. 106].

Besker et al. [19] link the presence of most technical debt to prior architecture decisions and sloppy maintenance effort. With a suboptimal architecture, complexity and cost of maintenance will rise and result in growing difficulties during maintenance. This applies especially to large or very software systems (over 500KLOC) where architecture related decisions can have extraordinarily important consequences. In such systems, the cycle of producing software can produce excess architectural debt causally in self-reinforcing manner (poor architecture→poor maintenance→poor architecture) with drastic consequences. The origins for the growth of architectural debt are compromises of modularity, poor reusability, analyzability, modifiability, testability, and evolvability of a software system.

Kruchten et al. [15] refer to architectural debt being as primarily "invisible" and hard to identify. It is not easily spotted by inspecting locally code ("invisible"), because it reveals itself in later stages: "when the system reveals shortcomings or complications in the maintenance or operation" [19: p. 2]. Moreover, architectural debt often is symptomatic of "technological gaps" to be crossed to evolve a software further [15]. A few empirical studies on architectural debt and its effects indicate that it reduces development speed even when using agile methods [12]. Moreover, mechanisms for resolving such debt (e.g. refactoring) are rarely prioritized until totally necessary [12]. In many settings, the accrual of debt triggers a reactive causal sequence where agile teams seek to speed up their development and hence plant architectural debt, which, in turn, reduces speed and further increases the complexity of software [12].

The picture that emerges from reviewing research on architectural technical debt is that while early studies focused on categorizing debt and studying its consequences for software quality and maintenance, recent studies focus on how architectural debt makes software less evolvable and reduces the speed of product development. Extant literature – including the literature focused on architectural debt, tends to treat technical debt as a "local" code level phenomenon confining to the quality of the code within applications and having consequences to programmers' practices, related team, and project level activities. From the view point of broader digital innovation within organizations, it would, however, be important to identify and theorize how architectural debt becomes manifested across modules, applications and layers within a complex IT architecture. Such architectures are typical to most large firms operating legacy systems.

Although the topic of architectural debt is weakly covered on research on digital innovation, its effects on reduced development speed is highly relevant for understanding conditions of and barriers to digital innovation which emphasizes evolvability, short and fast innovation cycles, and radical shifts in the product architectures and features [3]. While recent studies have examined processes of managing technical debt during platform innovation [10], its importance as an element of digital business strategy [22], and how it affects reliability in enterprise platforms [14], there are no studies that have explored how architectural debt is managed during and what are its consequences for digital innovation. Past studies, however, help contextualize the relevance of technical debt in managing digital artifacts, their organizational settings, and how they generate options. Additionally, a recent study by Rolland et al. [10] outlines a process model of how technical debt gets planted, evaluated, and resolved. Planting technical debt refers to how developers take up new debt. Uptake of technical debt is not solely a case of incompetence. It can be planted for unintentional, tactical, or strategic reasons [11]. Therefore, not all technical debt needs to be paid back [11] and timing of resolving debt is important [10]. This implies that it is of importance when and how to identify and evaluate debt and understand its consequences.

The lack of IS research on architectural debt, the "local", circumscribed focus on technical debt, and the void of connecting technical and architectural debt to socio-technical and often distributed, multi-level processes of digital innovation suggests significant gaps in the extant research.

2.2 Managing architectural debt as working across knowledge boundaries

In order to understand how architectural and technical debt is planted, evaluated, and resolved across applications and layers and what are its wider consequences, we will draw on the concept of "knowledge boundaries" [16-17]. We draw this lens as a means to conceptualize how code related knowledge is "localized, embedded, and invested within a function and how, when working across functions, consequences often arise that generate problematic knowledge boundaries." [16: p.442]. The perspective is relevant for understanding architectural debt since such debt involves identifying and applying knowledge that is localized, embedded and invested within local development teams and at the same time manifest technical interdependencies that need to be identified and understood across multiple teams, applications and stakeholders. As such, in complex IT architectures, knowledge about technical and architectural debt need to be shared, understood, and negotiated across multiple teams, units, and disciplines – and sometimes also across organizations. Especially in situations where digital innovation relies on knowledge input and development skills of geographically, functionally, professionally and/or culturally dispersed actors, the capability to work across increasingly complex knowledge boundaries becomes essential. In large incumbent firms such situations are not exceptional in that the software development is commonly outsourced and business and most IT units are organized into self-confined functions [23]. On top of this, many organizations have recently established "autonomous" agile (front line) product focused teams [24] that often introduce yet another knowledge boundary.

In validating his framework, Carlile [17] investigated traditional 'physical' product development, he recognized how multiple teams – or functions in a firm, over time, develop original knowledge that is locally embedded and situated. Similar challenges are also noted by Lyytinen et al. [25] who reviewed the heightened translation challenges created by highly heterogeneous digital product development teams. These differences establish knowledge boundaries because of highly common practices, frames, and interpretations inside the boundary. This investment works well for communicating within teams, but creates problems when working across teams since other teams or specialties do not share practices and interpretations and have not 'invested' in the specific type of knowledge [17].

With regard to architectural debt, this equals a situation where one development team plants new debt which is not accounted for, recognized or agreed up on by other teams – but at the same time may have consequences for other teams, because of extant technical interdependencies. Increased novelty, speed and innovation within teams is moreover likely to increase the 'gap' [17]. Furthermore, Carlile [16,17] narrates how working across boundaries will involve communicating across increasingly complex boundaries at multiple levels (syntactic, semantic, and pragmatic) where each layer involves more complex processes of transferring, translating and transforming knowledge. In the case of syntactic boundaries information can simply be transferred from one team to another using common digital tools and standards for representing the necessary code related information. Less complex types of technical debt, involving syntactic knowledge boundaries (call structures for example) that can be easily be identified and confined, can be document and shared across teams. Examples of this can be technical debt which is more or less visual in cases where a user interface is not updated as new back-end features become available. In cases of a semantic knowledge boundary, more complex processes of interpretation – such as different steps in perspective making and perspective taking becomes necessary [26]. Semantic boundaries are likely to cause architectural debt as misunderstandings across business units, development teams, and individuals undermine benefits of modularization and layered architectures. The particular abstract and complex nature of software architecture [27], requires in-depth specialist knowledge about the algorithms, internal details of structure, and historical background of major design decisions of the software in order to identify and evaluate architectural debt. The multitude of APIs and growing use of micro services in contemporary software often make it hard navigate and choose among alternative design choices causing developers to unintendedly plant architectural debt. Moreover, although architectural debt is identified and evaluated, its resolution requires that multiple teams and individuals coordinate their work and are willing to prioritize activities to resolve architectural debt. However, research has indicated that this is seldom done [12], and that in general, inter-team coordination in large-scale software development and complex organizational settings can be difficult [23, 28]. Furthermore, in order to innovate, invest in new technologies and standards that potentially will diverge more interests and manifest later latent conflicts. In such situations, actors must cut across pragmatic boundaries [16,17] and negotiate. With architectural debt it has been noted that organizational actors often have diverging interests when it comes to prioritizing activities to avoid debt or to resolve existing debt [12, 20]. Typically, business managers prioritize fast launching of new digital services and features, while an IT department's interest would often be in ensuring security, maintainability, and reliability. Furthermore, as many organizations outsource their software development, architectural debt can be introduced by external partners, because they want to deliver according to contract or maintain specific applications within cost. This is not necessarily in the best interest of organizations wanting to reduce and manage architectural debt.

3. Method and case description

3.1 Case study method

Because there is a lack of studies emphasizing the role of architectural technical debt in digital innovation, we conducted exploratory case study to develop emergent theory [29, 30]. We sampled the case of BankAlpha as we wanted to study this phenomenon in the context of an incumbent organization with a complex IT architecture involving numerous legacy IS as a representative case. Data collection started in spring 2019 with a focus on how IT architects were organized, their practices, and especially how they worked with technical and architectural debt. We drew on multiple sources of evidence including meetings, interviews, and documents. In early stages, four meetings with principal architects were conducted to get an overview of the complex IT architecture, legacy IS, and integration technologies utilized in BankAlpha. A meeting where an overview of the organization, business units, and core services was also conducted. In addition, we had a meeting with the IT director where different options for re-organizing the IT function was discussed. The meetings were extensively documented. During autumn 2019 and spring 2020 11 in-depth interviews were conducted using a common interview instrument including questions on the following main topics: 1. What project(s) did you work on? 2. How did your project(s) manage technical debt? 3. How did you identify architectural debt, how did you deal with it, and what where the consequences over time? 4. How did decentralizing IT affect management of technical debt? 5. How did transition towards use of agile methods affect management of technical debt? 8 interviews were recorded and all interviews were extensively documented. Informants included Business managers (3), IT architects (3), Enterprise architects (2), and Software developers (3). Interviews were also supplemented after interviews with informal discussions and email exchanges to shed light on specific issues and double-check information. Documents such as available PowerPoint presentations on IT architecture, agile organizing of software development, and organizational charts were also consulted.

Data analysis was done in three steps. First, we coded all interview data using a combination of pre-defined codes from theory; knowledge boundaries, boundary capabilities [16, 17] and notions of planting, identifying, evaluating, and resolving debt [10]. Thirdly, using the narrative approach [31], we identified three narratives describing how software development teams struggled in "managing" architectural debt in practical day-to-day situations; 1) how architectural debt was plated across teams, 2) how unknown architectural debt in legacy IS influenced software development, and 3) how architectural debt was unintendedly planted.

3.2 Case context of the mobile bank project in BankAlpha

BankAlpha is a large Scandinavian bank with several hundred years of history as a reliable and important economic and financial institution. In recent years, BankAlpha has been faced with increased competition from Fintech companies and international digital payment platforms. As a response, to improve their digital innovation capabilities, the bank re-organized its IT-function in 2019 from being relatively centralized to more decentralized and to integrate software development with its business units. BankAlpha has also invested heavily in software development to digitally transform itself. In 2018 BankAlpha adopted an agile software development method based on the Spotify model [7]. With this approach, developers and business representatives are organized in "autonomous teams" where they are responsible for not only developing the software but also for evolving and maintaining it. In terms of technology, BankAlpha embraced microservice-oriented architectures, extensive use of layering and APIs, and is currently developing its applications on the AWS cloud.

In spite of these efforts, BankAlpha was struggling with increasing uptake of technical debt related to its complex IT architecture and reliance on numerous legacy IS. An essential backbone of its digital infrastructure remains several legacy core banking systems running on mainframe computers with software written in COBOL. As a result of numerous acquisitions and mergers, the firm was endowed with multiple "archeological" layers of legacy IS often with overlapping functionality – as for example there were multiple core systems in use for handling credit cards. In this way, architectural debt has accrued abundantly over the years. A bulk of maintaining these legacy systems has been for some time outsourced and offshored to Indian consulting companies. This situation also made technical and architectural debt somewhat "invisible" for internal business and IT managers, as noted by the central IT architect: "*We are aware of technical debt, but in practice we do not have detailed insight in uptake of technical debt in applications*". Lately, however, this situation has been changing as BankAlpha has made decisions to insource most software development and maintenance of its strategic and core banking applications.

With this backdrop, in 2017, BankAlpha kicked off an ambitious project involving over 100 software developers and designers to develop new banking services on Smartphones. The goal was to launch a mobile bank APP running on the AWS cloud platform. This is a serverless and flexible IT architecture with layering and APIs, and thereby its goal is to discontinue the current online banking platform. However, the project was severely hit by an increased uptake of technical debt. Especially, the project ran into high amounts of historically planted architectural debt which was intrinsic to the many legacy IS in use. This reduced the speed of innovation and increased the complexity of the current IT architecture. However, in a couple of years after the initial start, the project finally succeeded to launch a new mobile banking app on Android and iOS platforms, and is currently evolving these applications further.

4. Results and analysis

We next scrutinize three distinct narratives and analyze how actors had to work across knowledge boundaries while identifying, evaluating, and resolving architectural debt.

4.1 Planting architectural debt across distributed teams

Initially, the mobile bank project (hereafter referred to as the project) started off in 2017 following the old waterfall approach with toll gates. The project started with developing architectural foundations and conceptual ideas developing native mobile apps in AWS cloud and APIs accessing a 'shared service layer'. The 'shared service layer' was also developed in parallel in order to serve as a common middleware layer with APIs for accessing core bank systems for all cloud-based applications. The project was organized in a distributed fashion with developers located on one site in Scandinavia and one in India, and UX designers, project and business managers at a third location in Scandinavia. However, the project turned out to have a too large scope and the distributed organizing of the project made coordination across challenging. This led to planting of technical and architectural debt where one a team's choices at one location was unaware of the technical interdependencies and consequences for teams located elsewhere. In 2018 the project was re-organized into agile teams with more in-house developers. After a while the teams became aware of the technical and architectural debt. In particular, there was considerable architectural debt in the way that the initial APIs were designed leading to considerable work to re-design and re-implement software components relying on these APIs: "*We could continue to work with some of the ideas behind the current architecture, but we had to re-implement almost everything. The entire iPhone APP had to be re-implemented and more importantly, many APIs*." (Business manager). Early in the project it was quickly decided to use the Go programming language for back-end programming. Later this choice turned out to have insufficient and undocumented features, and over time became architectural debt that slowed down the project. As the project proceeded in more co-located agile teams, the uptake of technical and architectural debt was seen as necessary to meet deadlines and business unites' expectations as explained by one developer: "*Much of architectural technical debt was consciously planted. We knew that this had to be done because of pressure to meet deadlines*" (in-house developer).

Analysis:

In the first phase the project suffered from lack of coordination and knowledge sharing across a very distributed development organization that unintendedly planted architectural debt that eventually significantly slowed down the overall innovation process. Uncoordinated choices regarding architecture, like development of APIs and selection of development platform that was done by different teams somewhat unintendedly planted architectural debt.

Drawing on Carlile's lens [16,17], effective coordination and sharing across knowledge boundaries requires processes of translating programming and architecture knowledge across the various locally distributed teams. This can be challenging as it not only requires competent developers understanding the syntax and structure of legacy programming code, but additionally that they can cognitively and socially translate its deeper implications, limitations, and meaning [25]. Working in a mostly as co-located teams later in the project, however, the local planting of architectural debt was more easily identifiable and transferrable across less complex knowledge boundaries – such as other co-located teams or within teams. At this point the developers were very well aware of the existing architectural debt as this was partly documented in Jira and in comments in the code. For example, they were aware of "local" architectural debt related to using a specific coding library for GUI on Android phones.

Architectural debt planted during the first phase of the project was not identified until years after re-organizing the project. Architectural debt was in this way different from technical debt in that it inherently is discoverable only long after it is planted. As such, it typically only becomes "visible" when trying to integrate various modules and to evolve the software further with more extensive or novel features [15]. This can make architectures path dependent, since at the point of discovery, resolution of contingently planted architectural debt may be considered too costly and risky. With regard to the Go programming platform in the project this was the case. The project just had to live with the initial decision although the involved actors realized it was a rather sub-optimal technology platform.

Our analysis indicates that architectural debt requires working across more complex knowledge boundaries as its causes and effects are stretched out in time complicating processes of sense making and sense giving. Also, although the Project involved local technical debt, architectural debt was considered far more critical due to the need for crossing knowledge boundaries: "*Architectural debt is the most demanding form of debt as it has broader consequences and can not only be resolved through prioritizing an activity in the release backlog of one application" (Chief Architect*)

4.2 Unknown architectural debt in legacy IS and outsourced IT

Like many financial institutions with a long history, BankAlpha also had extensive architectural debt related to their legacy IS. Architectural debt in legacy IS is often not documented and those developers who planted it are typically long gone. Generally, the developers and business mangers were all very well aware of that it existed, but not exactly what it was and what consequences it could have for new software development projects. Thus, as one developer explained just identifying technical and architectural debt is very difficult and sometimes near impossible: "*It is very hard to identify architectural debt during development and testing. It first comes to surface during production*" (Developer #3).

On the contrary, with a modularized and layered architecture, the expectation from most business managers was that building a mobile bank on top of existing layered IT architecture would be relatively straight forward. In contrast, the project members soon realized that this was not the case: "*Although AWS cloud is almost unlimitedly scalable, our solutions are not – since they all depend on legacy systems*" (Developer #1).

A particular expounding consequence of largely unknown or hidden architectural debt in legacy IS, was typically stumbled up on when the project wanted to add new features to the mobile APP that had slight differences compared to existing ones. At one occasion the team wanted to add a feature to deactivate credit and other bank cards. Deactivation of a card is often needed by customers when they think that they have lost their card. However, in most cases, they find their lost card and hence immediately want to activate it again. Previously when customers had lost their card, they had to call the bank and the bank manually deactivated the card and then a new card was produced and issued to the customer. However, implement this new feature surprisingly turned out to be extremely time consuming. It involved juridical expertise, re-design of internal work processes, and resolution of architectural debt in several legacy IS – typically offshored to India. In general, outsourcing of IT was especially problematic as it made such debt hidden: "*Until 2019 we outsourced much of our infrastructure, IT-operations, and application development which sometimes hides technical debt for us, because continuous management and maintenance are left to the suppliers." (Chief Architect)*

This type of challenges had huge consequences for prioritizing of features in the project, as explained by a manager: "*We think a lot about what we can do on our own without involving modifications in legacy IS. For example, if I want to have an overview of historical interest [in the mobile APP] – to develop the API will take nine to twelve months. Consequently, we need to prioritize differently…*"

Analysis:

Legacy IS has long been a profound challenge in incumbent firms like BankAlpha. The above narrative, illustrates that although modularized architectures, Cloud solutions, and APIs make it possible to build new APPs on top of legacy IS, it is still problematic due to historical uptake of architectural debt. Drawing on the concept of knowledge boundaries [16,17], we see that managing architectural debt relies on the actors' boundary capabilities as the detailed knowledge about this debt is decentralized across different teams and communities. Hence, the lack of such boundary capabilities to work across semantic and pragmatic boundaries render architectural debt invisible to application developers.

In particular, there was a huge "gap" between those who developed new applications and the largely outsourced teams involved in maintenance of legacy IS in terms semantic as well as pragmatic boundaries. First, the semantic boundaries involved the problems in translating the needs of the APP developers to the teams maintaining legacy IS. Changes that APP developers thought of as simple and easily done, took months since they did not understand the complexity of resolving the architectural debt involved. Second, there was also a pragmatic boundary involved since offshore development was contract-based it was not necessarily the IT suppliers' best interest to resolve debt. Moreover, the IT function of the bank also tended to prioritize stability, reliability and security as their number one priority, making rapid and substantial modifications in legacy IS unwanted.

4.3 Unintentionally planting new technical and architectural debt

As mentioned, the project did not only develop a new online bank on Android and iOS, there was also a team working on the 'shared service layer'. This was part of the new strategy to develop a common API not only the mobile APPs, but for all new applications developed in the Cloud. In turn, the shared service layer uses the APIs running on another layer of middleware which gives uniform access to the actual core banking systems. This layered architecture makes it relatively easy to build new apps that integrate with legacy IS in a secure way. However, as there are different distributed teams working on the different layers and modules, this also tend to hide complexity. As one of the main architects stated: "*It takes time to fully understand all the interdependencies. We reorganized the IT-function not just because it reduces complexity in itself, but because it gives ownership and the continuity required for proper understanding of how systems are implemented and why, and thereby supporting better long-term architectural decisions*."

Hence, the layered modularized architecture also hides important details of how these different core systems actually work, and sometimes make it hard for application developers what APIs to use and why. The agile teams which had a strong focus to speed-up development of new features and were also pushed by impatient business managers, and hence developers tended to unintentionally plant new architectural debt since they did not have enough knowledge about what was beyond the next layer in the architecture: "*When you work with front-end issues it is very difficult to discover bugs and how things work downwards in the architecture. Not until you discover what happens in production…" (Developer #2).* Furthermore, as much software development historically had been outsourced and offshored, some of the code appeared as chaotic and nearly incomprehensible lacking any comments in the code. To get around this situation, developers had to "bet on" and just try out which APIs to use: "*Instead of analyzing and fully understanding the complexity, we tended to use simpler and more generic APIs*" (Developer #3). In this way, existing architectural debt concerned with the interfaces and organization of APIs across modules and layers, makes it likely that individual developers introduce bugs and plant new technical and architectural debt. IS development at BankAlpha also needed to take specific compliance rules and regulations issued by the Governmental authorities into account. Whilst employees on the business side are knowledgeable about such compliance issues, this is not necessarily the case for software developers and UX experts. This situation, occasionally caused development of new software that turned out not to comply such rules and regulations, and hence planting substantial amounts of technical and architectural debt.

Analysis:

The layered architecture at BankAlpha offered a lot of initial advantages. In combination with decentralized and distributed organizing of development, and a competitive context with extensive pressure to deploy new features, however, it can cause unintended uptake of architectural debt. Efficient management of architectural debt at BankAlpha seemed to imply working across multiple semantic and pragmatic boundaries, as expressed by a developer: "*We need to learn to communicate better with the people working on the shared service layer, and they need to communicate with people maintaining the core banking systems… and they often sit in India.*". Hence, there is multiple processes of translation needed in order to bridge the semantic and pragmatic boundaries invested by the different teams. Failure to do this, as the narrative above illustrates, are likely to cause unintended planting of new technical and architectural debt. This, will over time, in turn cause more complicated code and software architecture making it even more difficult to cross knowledge boundaries across teams working on different levels in the complex architecture.

5. Discussion

Our objective has been to empirically explore the dynamics of managing architectural debt in complex IT architectures and to theorize how such debt affects digital innovation. In so doing, we have drawn upon the concepts of knowledge boundaries and boundary capabilities [16,17]. We found this lens fruitful in that although architectural debt by itself is highly technical issue, the task of identifying, evaluating, and resolving – in short managing successfully such debt, is socio-technical in the sense that managing architectural debt requires team level capabilities to work across semantic and pragmatic knowledge boundaries. Although project teams can successfully manage most local technical debt internally, they cannot similarly manage architectural debt that cuts across multiple knowledge boundaries. These insights add to existing studies of architectural debt [12,15,19] and explain why architectural debt is different and more challenging than technical debt confined within singular loosely coupled software modules. As such, our study shows that void of such capabilities, manifested in varied boundary spanning practices and objects and related cognitive and social capabilities that translate and transform knowledge across boundaries, will increase the likelihood of planting new architectural debt. Moreover, poor abilities to identify architectural debt will later escalate a string of surprises and increasingly complex problems while developing new modules and applications. This, in turn, will have broader consequences for the organization's capability to innovate under the auspices of its current IT architecture.

Grounded in the case study of BankAlpha, we posit that architectural debt influences innovation speed and quality and shapes the evolution of complex IT architectures in multiple ways. In particular, we identify two *tensions* that partially account for the dynamics of organizational capabilities necessary to manage architectural debt during digital innovation.

First, we note a *tension between decentralized IT governance and IS development, and the capability to manage architectural debt*. IS scholars have for long argued that decentralized IT governance promotes local flexibility, innovation, and an ability to quickly respond to changes [32, 33]. However, decentralized IT governance involving complex IT architectures will introduce higher and larger number of knowledge boundaries which require new boundary crossing capabilities to identify, evaluate, and resolve architectural debt. As shown in the case, decentralizing IS development using agile teams does *not* by itself improve the capability to manage architectural debt, though it improves the ability to manage internally technical debt within each module. Consequently, while it makes sense to decentralize software development so that new solutions can be co-constructed by IT developers and business experts using cross-disciplinary teams that boost innovation [35], such configuration can reduce the organization's capability to take and manage architectural debt.

We note also a *tension between desire for higher speed with agile development and the long term capability to manage architectural debt*. For some time speed has been an important facet of digital innovation [36] and strategizing [3,37]. This come from the necessity to speed up learning as to improve strategic decision making, accelerating new product development, and the sense and response cycle [37]. However, the focus on speed can conflict with the need to manage architectural debt. In general, long term continuity can be more important than immediate speed in software development when software is viewed as an asset [35]. As seen in the case, high-speed development can undermine prioritizing of resolving architectural debt, increase the amount of unobserved planting of architectural debt, and lead to taking up more architectural debt than the organization can handle. Highly productive agile teams enjoy a high degree of autonomy and are typically relatively small. This however, generates a pressing need for coordination across teams and projects [23] involving creation of boundary spanning capabilities to effectively translate knowledge necessary to identify and resolve architectural debt. Typically, it also makes it more difficult to identify accrued architectural debt in other modules and layers. Generally, organizations endowed with relatively high degree of modular and layered architectures founded on APIs and microservices are better positioned to support continuous innovation. However, as shown in the case study, decentralized autonomy in IS development and high-speed agile development lead to favor business managers immediate needs for prioritizing new features rather than spending effort to identify and resolve architectural debt. Hence, paradoxically, a too strong focus on speed, will actually slow down the speed, because cumulated architectural debt needs to be resolved at some point in order to add new product features. A critical mechanism for learning and innovation in agile development – to fast and frequently release new features, can actually undermine the long term capability to manage architectural debt.

Interestingly, per previous research on IT governance [32, 33], digital innovation [1,2], and agile development [7, 23], our results appear paradoxical in the sense that mechanisms that normally would produce flexibility, innovation, and change also will undermine these changes because of the amplifying effects of accruing architectural debt. The nature of the two tensions should be understood as a duality rather than a dualism [34]. Per Farjoun [34], duality similar to dualism describes the relationships between two entities. However, if the relationship is a duality, then the two entities are always interdependent, and at the same time both polarizing and complementary. This distinction conceptualizes stability and change as mutually enabling rather than mutually exclusive [34]. Likewise, we postulate that decentralization of IT and capability to manage architectural debt is a manifestation of duality: there is a need for effectively manage architectural debt in order to decentralize IT governance and create flexibility in IS development, and vice versa. There is always a need for a degree of decentralization in order to manage architectural debt. In this way, decentralized IT governance and capabilities to manage architectural debt are complementary. However, a too strong focus on either polarizing end will result in a contradictory scenario creating a vicious circle where absence of boundary capabilities builds more architectural debt, which, in turn, renders the IT architecture increasingly complex.

Likewise, the tension between high-speed agile development and the capability to manage architectural debt need to be understood as a duality. Effective management of architectural debt increases the speed of development, and higher speed of development – under certain conditions- imply faster and earlier identification of architectural debt. In contrast, a lack of balance triggers an escalating tension that undermines the organization's capability to digitally innovate with agility and produce value.

Per the case study we posit that a contradictory scenario (dualism) is triggered, in particular, in contexts with complex IT architectures that involve legacy IS, compliance issues, and a highly competitive environment. Compliance issues require IT developers and managers to constantly check for compliance and consult legal experts to ensure that new applications are compliant. This, however, introduces new knowledge boundaries, and in particular pragmatic boundaries where compliance experts, IT developers and business managers tend to have diverging interests. Historical uptake and unknown architectural debt are likely to hinder IT developers and managers to quickly launch new and compliant solutions. At the same time, in highly competitive markets, there is a constant pressure to innovate and launch new products and services. In such environments, attempts at high-speed development are likely to backfire in terms of uncontrolled uptake of unreasonable and expensive architectural debt. No wonder that financial services, airlines or cars do not operate like Google or Facebook which so far have faced nearly unregulated environment.

6. Concluding remarks

To this date, IS scholars studying digital innovation have shown less interest in technical conditions conducive of digital innovation such as the effects of technical and architectural debt. This is unfortunate, because the void circumscribes the explanatory power of current research. From the outset, digital artifacts provide nearly unlimited possibilities for organizations to innovate through continuous recombination and reprogramming of digital components [2]. In practice, however, technical and architectural debt provide stringent limitations for designers while innovating new products and services. Theoretically, our research suggests that these limitations play out as tensions between the mechanisms promoting innovation, and mechanisms necessary to manage architectural debt. The more an organization attempts to innovate through high-speed development and decentralizing IT development and governance, the more challenging it becomes to manage architectural debt. Over time, these tensions interact and increase the unobserved planting of architectural debt. Failing to identify debt can also escalate this debt in a self-reinforcing manner. This non-linear buildup of architectural debt will – if not rectified, in turn, produce technical implementation failures and result in skyrocketing innovation cost. The practical implications of our insights are manifold. First, there is a need for IT developers and architects to not only be aware of other teams and projects, but also to establish coordination mechanisms with the explicit purpose of translating and negotiating insights and recommendations regarding architectural debt. This can, for instance, be implemented through architectural (debt) boards. Second, there is a need for organizations like BankAlpha to take more control of maintenance of legacy IS to build deeper competence on the buildup of architectural debt in their systems. This can be done through insourcing and backsourcing of core IT platforms. Lastly, high-speed agile development needs to be balanced with stabilizing activities like clear criteria for refactoring and enhancing planning-oriented practices to identify and analyze sources of architectural debt.

7. References

[1] Henfridsson, O., & Bygstad, B. (2013). The generative mechanisms of digital infrastructure evolution. *MIS quarterly*, 907-931.

[2] Henfridsson, O., Nandhakumar, J., Scarbrough, H., & Panourgias, N. (2018). Recombination in the open-ended value landscape of digital innovation. *Information and Organization*, *28*(2), 89-100.

[3] Nambisan, S., Lyytinen, K., Majchrzak, A., & Song, M. (2017). Digital Innovation Management: Reinventing

innovation management research in a digital world. *Mis Quarterly*, *41*(1).

[4] Svahn, F., Mathiassen, L., & Lindgren, R. (2017). Embracing Digital Innovation in Incumbent Firms: How Volvo Cars Managed Competing Concerns. *Mis Quarterly*, *41*(1).

[5] Yoo, Y., Henfridsson, O., & Lyytinen, K. (2010). Research commentary—the new organizing logic of digital innovation: an agenda for information systems research. *Information systems research*, *21*(4), 724-735.

[6] Mehrizi, M. H. R., Modol, J. R., & Nezhad, M. Z. (2019). Intensifying to cease: Unpacking the process of information systems discontinuance. *MIS Quarterly*, *43*(1), 141-166.

[7] Dingsøyr, T., Falessi, D., & Power, K. (2019). Agile development at scale: the next frontier. *IEEE Software*, *36*(2), 30-38.

[8] Bozan K., Lyytinen K., Rose G. (2020): Transitioning Incrementally to Microservice Architecture (MSA): A Field Study, *Communications of the ACM*, forthcoming

[9] Vestues, Kathrine and Knut H., Rolland, "Making Digital Infrastructures More Generative Through Platformization and Platform- driven Software Development: An Explorative Case Study" (2019). *10th Scandinavian Conference on Information Systems*. 4. https://aisel.aisnet.org/scis2019/4

[10] Rolland, K. H., Mathiassen, L., & Rai, A. (2018). Managing digital platforms in user organizations: the interactions between digital options and digital debt. *Information Systems Research*, *29*(2), 419-443.

[11] Tom, E., Aurum, A., & Vidgen, R. (2013). An exploration of technical debt. *Journal of Systems and Software*, *86*(6), 1498-1516.

[12] Martini, A., Bosch, J., & Chaudron, M. (2015). Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study. *Information and Software Technology*, *67*, 237-253.

[14] Ramasubbu, N., & Kemerer, C. F. (2016). Technical debt and the reliability of enterprise software systems: A competing risks analysis. *Management Science*, *62*(5), 1487-1510.

[15] Kruchten, P., Nord, R. L., & Ozkaya, I. (2012). Technical debt: From metaphor to theory and practice. *IEEE software*, *29*(6), 18-21.

[16] Carlile, P. R. (2002). A pragmatic view of knowledge and boundaries: Boundary objects in new product development. *Organization science*, *13*(4), 442-455.

[17] Carlile, P. R. (2004). Transferring, translating, and transforming: An integrative framework for managing knowledge across boundaries. *Organization science*, *15*(5), 555-568.

[18] Alves, N. S., Mendes, T. S., de Mendonça, M. G., Spínola, R. O., Shull, F., & Seaman, C. (2016). Identification and management of technical debt: A systematic mapping study. *Information and Software Technology*, *70*, 100-121.

[19] Besker, T., Martini, A., & Bosch, J. (2018). Managing architectural technical debt: A unified model and systematic literature review. *Journal of Systems and Software*, *135*, 1-16.

[20] Holvitie, J., Licorish, S. A., Spínola, R. O., Hyrynsalmi, S., MacDonell, S. G., Mendes, T. S., ... & Leppänen, V. (2018). Technical debt and agile software development practices and processes: An industry practitioner survey. *Information and Software Technology*, *96*, 141-160.

[21] MacCormack, A., & Sturtevant, D. J. (2016). Technical debt and system architecture: The impact of coupling on defect-related activity. *Journal of Systems and Software*, *120*, 170-182.

[22] Woodard, C. J., Ramasubbu, N., Tschang, F. T., & Sambamurthy, V. (2013). Design capital and design moves: The logic of digital business strategy. *Mis Quarterly*, 537-564.

[23] Rolland, K., Dingsoyr, T., Fitzgerald, B., & Stol, K. J. (2016). Problematizing agile in the large: alternative assumptions for large-scale agile development. In *39th International Conference on Information Systems* (pp. 1-21). Association for Information Systems (AIS).

[24] Dybå, T., Dingsøyr, T., & Moe, N. B. (2014). Agile project management. In *Software project management in a changing world* (pp. 277-300). Springer, Berlin, Heidelberg.

[25] Lyytinen K., Yoo Y., Boland R. (2016) "Digital Product Innovation within Four Classes of Innovation Networks", *Information System Journal*, 26,1, pp. 47–75

[26] Boland Jr, R. J., & Tenkasi, R. V. (1995). Perspective making and perspective taking in communities of knowing. *Organization science*, *6*(4), 350-372.

[27] Baragry, J., & Reed, K. (1998, December). Why is it so hard to define software architecture? In *Proceedings 1998 Asia Pacific Software Engineering Conference (Cat. No. 98EX240)* (pp. 28-36). IEEE.

[28] Bick, S., Spohrer, K., Hoda, R., Scheerer, A., & Heinzl, A. (2017). Coordination challenges in large-scale software development: a case study of planning misalignment in hybrid settings. *IEEE Transactions on Software Engineering*, *44*(10), 932-950.

[29] Gerring, J. (2007). *Case study research: Principles and practices*. Cambridge University Press

[30] Eisenhardt, K. M. (1989). Building theories from case study research. *Academy of management review*, *14*(4), 532-550.

[31] Langley, A. (1999). Strategies for theorizing from process data. *Academy of Management review*, *24*(4), 691-710.

[32] Gregory, R. W., Kaganer, E., Henfridsson, O., & Ruch, T. J. (2018). IT Consumerization and the Transformation of IT Governance. *Mis Quarterly*, *42*(4), 1225-1253.

[33] George, J. F., and King, J. L. 1991. "Examining the Computing and Centralization Debate," Communications of the ACM (34:7), pp.62-72.

[34] Farjoun, M. (2010). Beyond dualism: Stability and change as a duality. *Academy of management review*, *35*(2), 202-225.

[35] Fitzgerald, B., & Stol, K. J. (2017). Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, *123*, 176-189.

[36] Kessler, E. H., & Chakrabarti, A. K. (1996). Innovation speed: A conceptual model of context, antecedents, and outcomes. *Academy of Management Review*, 21(4), 1143-1191.

[37] Bharadwaj, A., El Sawy, O. A., Pavlou, P. A., & Venkatraman, N. (2013). Digital business strategy: toward a next generation of insights. *MIS quarterly*, 471