A FLEXIBLE METHOD, A RIGID ARCHITECTURE

Étienne Asselin[1] ***1773922*** and Vincent Rodier[2] ***1744784***

***Abstract*— A very important part of any software is without a doubt the architecture. Every software has an architecture, whether it was planned or not. The preferred solution is always to have a plan for the architecture that is able to fulfill every requirement of the software - reality is not always as bright. Meanwhile, agile software development techniques are gaining in popularity and discouraging people from planing up-front architecture. A project without any plan of architecture is a project set-up for failure - or is it? Is it possible to apply the Agile Manifesto philosophy all the way and not plan anything up-front? And if not, exactly how much planning is required? How to know exactly how much up-front architecture is needed?**

I. INTRODUCTION

Initially software development was only sequential following the cascade model. The steps were to list requirements, model a system architecture, implement, verify and ultimately maintain the software. This approach brought some benefits such as a complete documentation of the system, a project cost estimate from the start, the software was generally easier to maintain and the design followed a very specific standard. Despite the advantages of cascading development, it does not follow the reality of software development due to its lack of flexibility. Thus, in the early 2000s, agile development methods emerged.
Agile development follow four fundamental values:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

Here, the dilemma is finding the middle ground between the initial development of a software architecture and the ability to respond to customer changes. The problem is the duality between the rigidity of the initial development of architecture and the flexibility of agile development. Indeed, it is very difficult to derogate the architecture of a program when a possible change occurs. In such a case, the risk of failure must be reduced.
This paper aims at helping people to make decisions regarding how much up-front architecture (if any) is needed to mitigate the risks failure associated with the lack of planning philosophy of agile development. In order to achieve this goal, these questions must be answered:

- Is a project without or very little plan of architecture a project set-up for failure?
- Is it possible to apply the Agile Manifesto philosophy all the way and not plan anything up-front?
- How to know exactly how much up-front architecture is needed if any?

These questions will be answered using previously performed research. In this paper, agile development and software architecture will first be explained. Then similar work and the method of research of theses will be exposed. Finally the results will be presented and detailed.

II. BACKGROUND

This section will introduce some crucial knowledge to understand the whole paper.

A. Software architecture

The software architecture is a model for the system and the project. First, the architecture design is an artifact that allows early analysis of the system to ensure a design approach that will produce a system acceptable to the customer and the task it was design to do. Architecture is the primary support for system qualities such as scalability, performance, modifiability, security and cost reduction. So building an effective architecture can identify design risks and mitigate them early in the development process.

B. Agile development

The term Agile Development was consolidated in 2001 following the writing of the Agile Manifesto. Agile development methods rely on an iterative and incremental development cycle. Agile development encourages several principles including prioritizing customer satisfaction, supporting change requests and often delivering operational versions. To achieve these principle Agile Manifesto encourages the reduction of the amount of up-front planning and design. The agile development methods seems to promote the emergence of an architecture and refactoring at the expense of follow a plan [1].

C. Where the two meet

At first glance, it looks like A and B cannot work together and contradicts one another. The thing is that only sticking to the plan has the disadvantage of behing too stiff and cannot allow changes in the requirements down the line potentially leading to a software that doesn't work with the

[1]É. Asselin is a student in Software Engineering at Polytechnique Montreal, Qc H3T 1J4, Canada
[2]V. Rodier is a student in student Software Engineering at Polytechnique Montreal, Qc H3T 1J4, Canada

clients need. But if the requirements are changed during the course of the project, refactoring the entire process of planning an architecture is very time consuming and costly.

On the other hand however, giving yourself up to a series of ad hoc and unplanned decisions can have some disastrous effect too [1]. You may end up dealing with major refactoring problems that are impossible to deal with or straight up faillure of the project.

The answer lies somewhere inbetween these two senarios. Even if sometimes both methods of planning and not planning may lead to success, it is best to just enough of both in order to set you up for success. This reaserch

III. RELEATED WORK

To be able to write this paper, we search the *ACM Digital Library* and *IEEE Xplore* for work or research done in relation to this topic. We queried the library using words like "agile development" and "software architecture". The follow section will review one by one and in no particular order the chosen publication that was taken into consideration to write this paper.

Waterman, Noble and Allan (2012) investigated how much up-front architecture was important and how much value agile practitioners valued it. They did a number of interviews with agile practitioners that have knowledge of software architecture to highlight practices that the industry employs in hopes of finding the right amount of up-front architecture needed in a project. After 11 interviews and one focus group the researchers found out that using template architectures is useful because it reduces the amount of up-front architecture needed while being easier to adapt and change than a custom made architecture. They also stated that the experience of the team working on a project using agile development makes a difference. A more experienced team is more likely to make good strategic architectural decisions that are in line with the agile method and avoid needless documentation.

Waterman, Noble and Allan (2015) once again pursued their previous work to find how much up-front architecture is needed while exploiting an agile method of development. As follow up to their work in [1], they did another set of interviews with agile practitioners but this time, they interviewed 44 people in 37 semi-guided interviews. In this paper, the authors laid out 6 dimensions that were a threat to any software development project and they enumerated 5 strategies used by agile development to respond to those threat. The conclusion the team came up with in this paper is still in line with the first work they did [1], but this time focusing more on agile practitioners address problem that come with up-front architecture and agile development.

The work done by Highsmith and Cockburn (2001) in [3] depicts reasons why previous ways of working in the waterfall era don't work as they were intended in the real world. [3] describes how the agile development methods should prepare themselves for a project. [3] doesn't talk about how the architecture of a project suffers or benefits from agile methods though. It focuses more on explaining what exactly agile methods aim to do, how they do it and why they do it.

In this short yet interesting paper [4], Kruchten (2010) tries to demonstrates that there is actually very little conflict between using agile development methods like XP or SCRUM and still focus on early on an architectural design. The analysis of Kruchten in [4] is done by stating seven different dimensions that help identify why agile and up-front architectural planning seem in conflict and later answer them in a tutorial aiming at helping people find a balance between planning and doing work.

In [5], Nord and Tomayko (2006) begin by describing the view of agile practitioners stating that the majority of them see architecture planning as a very work intensive process that does not add any value. The authors also states that the architecture of any software projects reflects on the quality of that said projects and that is isn't exclusive to any development method. With an extensive analysis of the agile methods all the while identifying flaws, the authors of [5] show how agile development is not fitted to respond to all problems and how the lack of preparation can cause disastrous results sometimes. Nord and Tomayko (2006) conclude their work by showing point by point how architecture driven methods can not only help agile practitioners with their work but that it complements very well XP agile development method.

In their paper [6], Schramm and Daneva (2016) explore the implementation or SOA and agile software development. The subject the two authors set for in their research was to understand how SOA and agile development work together with three questions. Those three question help their methodology and helps guiding the work they did. Schraam and Daneva (2016) approached their subject with qualitative data in a well-known professional blogging platform publicly available. The idea behind choosing to study blogs is that by focusing on blogs written and were people who comment are practitioners they'd have a better understanding of real practices in the industry. After gathering the data, the two authors analyzed everything using the open coding and other technique. The results presented in this study are quite extensive. The authors reported three main categories that practitioners talk about a lot and then elaborated more on each one of them. In the end, it shows that SOA and agile development can work together and that utilizing certain concepts like collaboration

can increase productivity and response to change.

Chen and Babar (2014) did an empirical study of the emergent architecture in a software while using agile methodology. The goal of this study was ”to explore the perceptions and experiences of practitioners for identifying and building a taxonomy of the factors that can influence the emergence of a satisfactory architecture through continuous refactoring”[7]. The authors of this paper also used an interview and surveying method in order to gather information about their subject and all together, they interviewed 102 experienced software professionals from 13 countries and 6 continents. This study found 20 factors that contribute to the emergence of an architecture in the development of a software and they categorized these 20 factors into 4 category that they each explored more in detail. The results were that a satisfactory architecture can emerge from constant refactoring alone but that contextual factors play a huge role in these. Interestingly enough, they also found out that around 9% of that were interviewed have never encountered an emerging architecture that was not satisfactory.

Santos (2016) for his part focused exclusively on agile methods and what drives agile methods - why is it so popular. Santos states that simplicity is the what makes agile so great and what makes more software project successful. His problem with simplicity and agile methods though is that agile methods do not provide a good framework for applying simplicity. To explore his subject, the author divided his research in two part, beginning with a theoretical method followed by a second phase of industrial case study. The findings of this particular research is that agile methodology only encourage and emphasis simplicity but that still some concepts behind agility and simplicity are not enough understood by practitioners and researchers. A better understanding of these concept would be helpful to improve agile software development.

In paper [9], I. Hadar, Sherman, E. Hadar and Harrison (2013) focus on the challenges associated with agile development methods and architecture documentation. The authors lead a research inside a large internationally distributed firm that utilized both agile and none-agile methods. Their objective was to create an architecture specification document format that can coincide with agile way of things. The finding of this team were made possible by a number of interview of architects and by analyzing artifact produced by different project inside the firm. The conclusion of the team was that the best results are achieved by making ”an abstract architecture specification document using a supporting specification tool for creating a short and focused architecture document” [9]. This makes the architecture easy to understand and easy to modify without taking too much time.

IV. METHODOLOGY

The methodology we used decays into several distinct stages. First, we found the purpose of the literature review. Here, this paper aims at helping people to make decisions regarding how much up-front architecture (if any) is needed to mitigate the risks failure associated with the lack of planning philosophy of agile development. This first step allowed us to clearly identify the purpose of this paper so that the reader will know explicitly what is the subject. The second step is crucial since we are a team of two to write this SLR. It was essential that we were fully clear on the detailed procedure to be followed. This requires both a detailed written protocol document and an agreement on how we evaluated each literature. The procedure consisted of analyzing each of the sources according to 5 criteria on which we give them a score out of 10. These 5 criteria were:

- Reliability of the source
- Reputation of the author
- Objectivity of the information
- Accuracy of information
- Actuality of the information

The first step was to do literature research. We queried two databases: *ACM Digital Library* and *IEEE Xplore* using therms like ”agile development” and ”software architecture”. Another method to find more source for our study was to do an activity that is called snowballing. We actually did both kind of snowballing (forward and backward). The main purpose of this step was to expand our sample of potential studies. The second step was to evaluate the papers found in the search. As a result of the evaluation each participant retains only papers that scored above 35 out of 50 and added them to a potential nomination. Then the other partner must evaluate the papers retained and keep only those who have a grade greater than 35 upon a second examination. The last step was to synthesize the studies. The primary goal was to combine the data extracted from studies using qualitative and quantitative techniques.

V. RESULTS

Several results emerged from the analysis of the related works. These results are divided into two main sections: agile architecture forces and agile architecture strategies. M. Waterman, J. Noble, and G. Allan [2] defined an agile architecture as an architecture who meets the definition of agility by being incrementally and iteratively designed. In addition the agile architecture must be able to easily be modified in response to changing requirements, so it must tolerate to changes.

A. Agile Architecture Forces

The majority the research in our related works agree on the six forces identified by the study of Mr. Waterman, J. Noble, and G. Allan in 2015 [2]. [2] found six forces that

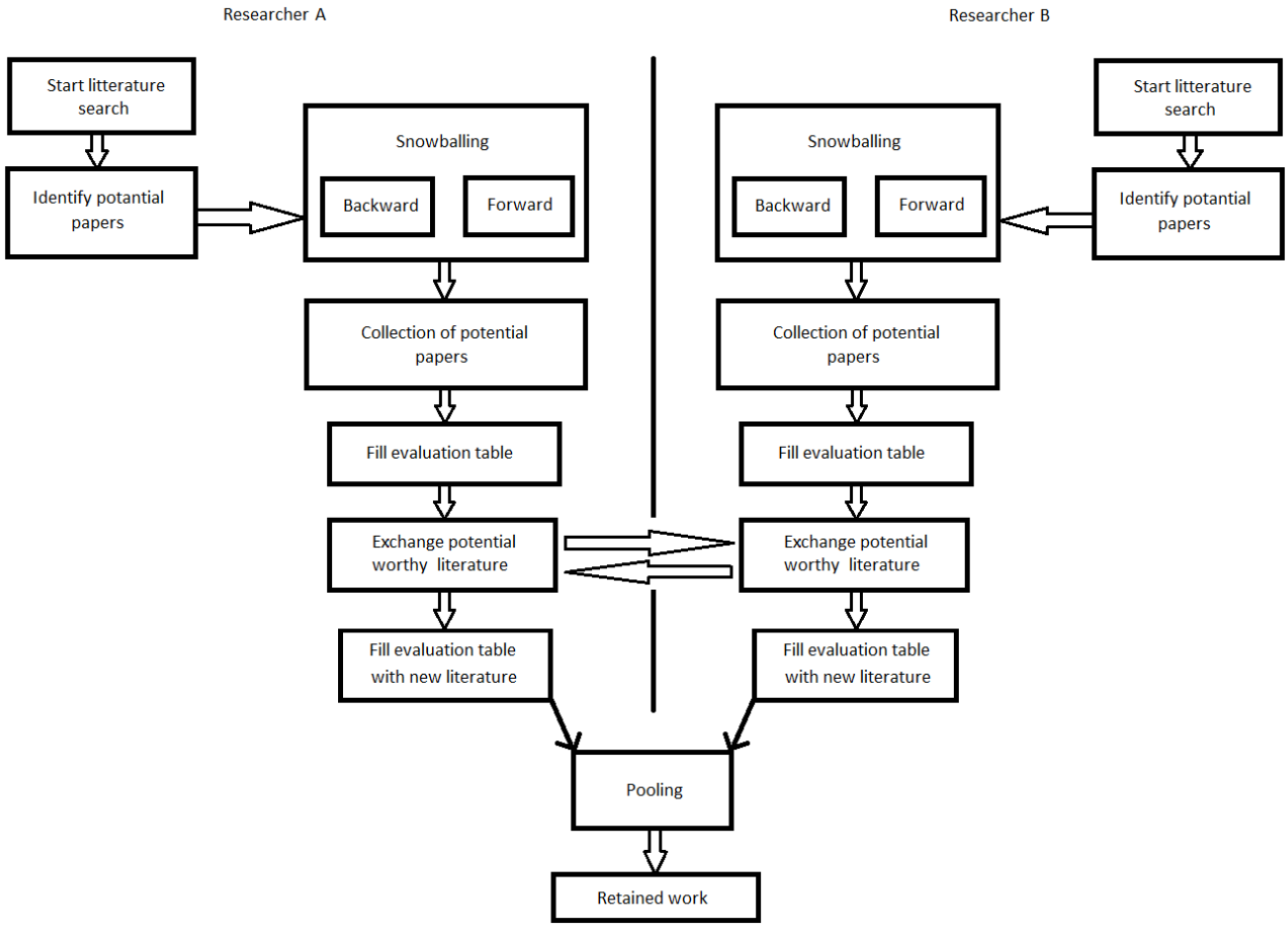

Fig. 1. Methodology graph.

		Criteria						
		Reliability	Reputation	Objectivity	Accuracy	Actuality	Total	Decision
Possible source	Source 1	7	5	9	6	4	31	NOT SELECTED
	Source 2	8	7	7	8	10	40	SELECTED
	Source 3	6	6	3	5	10	30	NOT SELECTED
	Source 4	9	8	8	10	10	45	SELECTED
	Source 5	7	5	8	9	6	35	SELECTED
	Source 6	5	5	8	7	9	34	NOT SELECTED
	Source 7	9	8	9	7	10	43	SELECTED
	Source 8	3	3	4	6	7	23	NOT SELECTED
	Source 9	7	7	7	7	7	35	SELECTED
	Source 10	8	7	9	6	9	39	SELECTED
	Source 11	8	10	9	7	10	44	SELECTED
	Source 12	6	5	7	5	9	32	NOT SELECTED

Fig. 2. Example of evaluation table of potential sources.

make up the agile development team in the development of architecture. The six forces are:

1) Requirement Instability
2) Technical Risk
3) Early Value
4) Team Culture
5) Customer Agility
6) Experience

Let's elaborate on each of these six characteristics forces that are specific to the agile methods and that reduce up-front planning.

1) Requirement instability: The Manifesto for Agile software development encourage responding to change over following a plan. Therefore the requirement instability need to be taken care of. Unstable requirements are caused by incomplete requirements and by changing requirements. Incomplete requirements are caused by the customer not initially knowing what they want as features in the system. Plus, they can come up with new ideas about what they want during development. In some cases, the requirements are more stable, but it is not possible for the team to develop a complete understanding of the system and these requirements from the start. Complete understanding comes later when the program is already under development.

Requirements may also change frequently following the first delivery due to customers changing their minds as they have already changed their usage patterns. It is often impossible for a team or client to know how their system will be used once it is operational. Therefore, development teams only need to define high-level requirements when initiating a project since the more specific requirements tend to be unstable. By avoiding the collection of detailed requirements, architecture analysis and design, software engineers can begin development and demonstrate the product to the customer early in the development, allowing them to gain feedback from the customer.

2) Technical Risk: The technical risk describes the effect that exposure to potentially negative outcome has on the initial effort of a team. The risk may be caused by three factors: having challenging or demanding architecturally significant requirements, by connecting many integration points with other systems and by involving legacy systems.

Architecturally significant requirements are the requirements that dictate and constrain the architecture of a system, and consist of mainly qualities, or non-functional requirements, such as performance, security and reliability. Also, architecturally significant requirements are difficult to design and can lead to many compromises.

Legacy systems are those that are no longer designed but are simply modified to meet changing needs. Good architecture and software quality practices aim for simplicity, modularity and strong cohesion are at risk because of the maintenance of legacy systems.

The complexity requiring the integration of such a system requires more initial exploration.Teams must reduce risk to an appropriate level. The amount risk reduction, and therefore the architectural effort, depends on the tolerance for the risk of the team and the client. A team mitigates risk by using a risk-based strategy as described in section *B. Agile Architecture Strategy.*

3) Early Value: The early value refers to the need for a customer to increase the value of a system or product under construction before all functionality has been implemented. The initial value is often required by companies in a dynamic business environment that cannot wait for the complete production of the software. Teams that offer early value must reduce the time of the first release by spending less time on the initial architecture design. This can be achieved by reducing the planning horizon. The planning horizon is how far ahead the team considers high level requirements for the purpose of architecture planning. In extreme cases teams do not plan anything in advance. No initial design increases the overall effort because the architecture must evolve with each iteration and so the cost of development. At each iteration, the team should consider whether the existing architecture is suitable for the current set of requirements implemented. If it is not the case, then the architecture needs to be redesigned.

Overall, it is likely that more time is spent on architectural development if there is little initial architecture. For example, as the product grows and the user base grows, the architecture that was designed day one may no longer be suitable and may need to be redesigned. A refactoring that may not have occurred with a larger planning horizon. This is the cost that you must be willing to pay to have the opportunity to obtain early value.

4) Team Culture: Team culture is an important factor in reducing the initial planning of a project. A team culture that encourages collaboration is very important for communication within the team. In addition, trust between members increases cohesion and reduces the up-front planning. However, it is also a feature that can increase necessary effort. A team where trust and communication are not present needs to rely on documentation and formal plans for communication. So, it requires more up-front planning to guide development. As the Agile Manifesto states, we must put forward individuals and interactions over processes and tools.

Another characteristic of the agile is the small size of the teams. This small size makes it possible to increase collaboration, unlike larger teams who need more structure and more planning. Finally, the last characteristic of team culture that affects initial planning is the experience. Indeed, a collaborative and confident team towards its members does not form instantly. Team cohesion comes with the experience and practice that each member works together. A new team for agility is likely to experience difficulties in achieving a successful project without a predefined full architecture. However, as the team gains experience developing together, it will become easier to work by reducing up-front planning.

5) Customer Agility: Customer agility is part of the culture of the customer's organization. This culture can have a great impact on the initial effort required on the design of the team's architecture. The customer must have an agile vision that is similar to that of the development team, otherwise the client and developers will work harder to achieve a result. Like the trust of team culture, trust between the customer and the development team is important to integrate the customer into the development team and thus eliminate the formal processes that require a lot of time. As the Manifesto for Agile software development states, we must prioritize the customer collaboration over contract negotiation. So, highly agile customers do not need development teams to produce exuberant documentation when there is mutual trust between the parties. In contrast, it is difficult for an agile development team to work on projects that are cluttered with processes that are only useful for the client. This translates into unnecessary production of documentation for the development team and a waste of time.

Another situation where a customer who does not have the agile mentality can negatively affect the development of a team is when the client must approve each of the architectural decisions at the expense of development. A customer may want to make sure that the decisions are compatible with the politics of the company. In addition, many non-agile customers prefer a fixed price contract with a fixed range and fixed delivery dates. For a team, these contracts involve investing a lot of time in advance to determine in detail the work required to meet the client's needs list.

6) Experience: The experiment describes the impact that the tacit knowledge and implicit decision-making ability of an experienced architect has at the time that an agile team devotes to initial design. Experienced architects have extensive knowledge, they are more likely to be aware of the appropriate options to implement a solution and better understand what will work and what will not work. Experience architects are generally important for all software development methods, but experienced architects are more important in agile development because tacit knowledge and implicit decision making leads to reduced process and documentation and reduces initial efforts. Inexperienced architects rely more on explicit decisions that are written. The later require more effort and time to make informed decisions.

It is also important that developers know the technology that will be used during the project. The knowledge or experience of the technology helps the team to speed up the design. If a team does not have the required experience, they may need to acquire this knowledge through research, or they may have someone with the appropriate experience to join the team. In both cases it will increase the time and effort to do the initial planning to learn the technology and / or to integrate the resource into the team.

These six forces of agile software development help reduce up-front planning, accelerating development and being a more agile process.

B. Agile Architecture Strategy

Following the analysis of strengths within our own team, strategies can be used to improve desired behaviors or to mitigate the effect of negative impacts on the initial planning for a project. Adopting these strategies will help development teams determine how much effort to put on up-front design. Theses five strategies are:

1) Respond to change
2) Address risk
3) Emergent architecture
4) Big design up-front
5) Use frameworks and template architectures

Let's develop each of these five strategies to improve the fluidity of software development through an agile process.

1) Respond to change: The ability of a team to respond to change is directly related to their agility. The first strategy increases the agility of the architecture by increasing its changeability and tolerance for change, and allows the team to ensure that the architecture continually represents the best solution to the problem as it evolves. So the development team is continually reviewing the solution to ensure that it is optimal for each iteration.
There are five maneuvers that development teams can use to implement the first strategy and design an agile architecture:

- Keeping designs simple
- Proving the architecture with code iteratively
- Using good design practices
- Delaying decision-making
- Planning for options

First of all keeping the designs simple means only designing for what is immediately required in the iteration and not

worrying about what might be required in the eventual future.

Secondly, proving the architecture with code iteratively means testing the design by actually developing it rather than building it theoretically. This maneuvers must be used once development has started.

Thirdly, the use of good design practices is important in any development, agile or otherwise, but it is particularly important in agile development, as it facilitates the modification of the architecture as requirements change.

Fourthly, delaying decision making means not making architectural decisions too early. Indeed, it is better to wait until we have enough information on the required. This is when the requirements will be less likely to change.

Fifthly, planning options means building in general and avoiding making unnecessarily compelling decisions that could block potential requirements without significant refactoring.

Three of these maneuvers: *keeping the design simple*, *proving the architecture with code iteratively* and *following good design practices*, increase the modifiability of the architecture so that when requirements change or become known the architecture can be easily updated. The other two maneuvers, *delaying decisions* and *planning for options* increase the architecture's tolerance of uncertainty, so that any changes to the requirements have less impact on the architecture. Finally, *Keeping the design simple, proving the architecture with code* and *delaying decisions* reduce the up-front effort while the other two may slightly increase the initial architectural effort, but will likely reduce the overall effort.

2) Address risk: The address risk strategy reduces or eliminates the impact of risk before it causes problems. This strategy should preferably be put in place before the risk occurs. This is why it is important to list the potential risks from the beginning of the project. Using this strategy, a team of architectural designs can develop an architecture that meets significant architectural requirements with an acceptably low risk. More technical risk means that more up-front architectural design is required. As a result, the team is less able to use the first strategy that is responding to change. Thus, to reduce technical risks, a team must sacrifice some of the team's ability to respond to change. The team must find a compromise between the first two strategies by making an initial design sufficient to reduce the risk to a satisfactory level using a technique that allows to delay decisions when the impact of the risk is low. The higher the risk impact, the more important it is to mitigate this risk early to reduce the cost of time and money. Development teams can reduce risk by using alternative search, modeling, and analysis, experimenting, or building a prototype of the system.

3) Emergent architecture: Emergent architecture means producing an architecture in which the team only makes minimal architectural decisions, such as the choice of technology and higher-level architectural styles and models. When using the emergent architecture strategy, the team only considers the needs that are immediately needed for its design, ignoring low-level and even high-level requirements that need to be implemented in the longer term. This strategy ensures that the design is as simple as possible and that the manufactured product can be launched on the market as quickly as possible, thus meeting the strength of agile development that is the early value. The emergent architecture is likely to be used when developing a minimum viable product. If the system has significant architectural requirements or unique requirements, it may require a more complex solution that requires custom components or multiple frameworks and a more advanced design for risk management. Therefore, the technical risk excludes an emergent architecture.

4) Big design up-front: The initial grand design strategy requires the team to acquire a comprehensive set of requirements and complete a complete architectural design prior to the start of development. There are no emerging design decisions, however architecture can evolve during development. This strategy is undesirable in agile development as it reduces the ability of the architecture to use the strategy to respond to change by increasing the feedback of information by designing the architecture at the beginning of the process, increasing the chances that decisions will have to be changed later, and increasing the risk of over-working.

While this strategy can be considered a cure to reduce the risk like the second strategy, address risk, in reality the use of this strategy is mainly motivated by a lack of agility of the client. The up-front design is sufficient to satisfy the non-agile customer. Indeed, it is enough to prove that the team knows how to solve the problem before starting. In addition, this strategy allows the team to estimate the cost of the system for the given requirements and to be able to complete the design without continued interaction with their client. While the team using big design up-front can not fully implement the respond to change strategy they may be able to use some maneuvers such as: *using good design practices* and *planning for options*, so as to they can always evolve their architecture as requirements change. Not being able to use the maneuver: *delaying decision-making*, however will compromise their ability to be agile. Large software vendors often use this strategy because their clients are more likely to be larger process-oriented

organizations that require greater financial accountability. These companies often have a hard time becoming as agile as smaller organizations. The frameworks also greatly reduce the complexity of the architecture because many architectural decisions are integrated in the framework, and therefore the architectural changes can be made with much less effort. In addition, frameworks and templates are extremely beneficial to most of development methods, and the ability to change architectural decisions more easily is very useful for agile methods.

5) Use frameworks and template architectures: This last strategy is the fact of using of frameworks and architectures templates to emerge an architecture. Frameworks such as .NET, Hibernate, and Ruby on Rails include default architectural patterns that constrain systems to these models, giving the opportunity to reduce the up-front design. Despite their importance, development teams need to be aware that frameworks can not always provide a complete solution. If the problem is not standard and the required architecture is complex or unique, there may be no existing frameworks or templates to address all or part of the problem. In these situations, a team needs to design and build custom components, so companies do not benefit from the advantage of this strategy. In fact, having to design custom components increases the complexity of the architectural decisions to be made and also increases the technical risk.

VI. DISCUSSION

This section will mostly discussed the implications and effect that this paper will or can have over the scientific community, agile practitioners and non-agile practitioners.

This paper can help researcher and the scientific community by orienting future research. Although both [1] and [2] are research that took real agile practitioners with knowledge and experience and interviewed them, [1] has a really small sample. Moreover, [1] and [2] are written by the same authors and [2] is just the second part of the research that the authors began in [1]. Not only the scientific community, but every person that develops software on a bigger than small-ish scale would benefit from the experience of other. Everything that has to do with planning software project is strongly dependent on context and every project has a different context. By having data on lots and lots of projects it becomes easier to spot similarities with ones the will be achieved in the future. The insight given by previous projects is very valuable and thus next research should aim to gather more of this type of data.

As said earlier, agile practitioners can also benefit from this paper. Practitioners can use this paper as a reference to better understand that the duality between agile and planning is just an illusion. It can help practitioners to understand that ad hoc architecture can sometimes be acceptable but that in most case an emergent architecture is just bad practice. It can also clarify what agility is and what are the primary goals when utilizing this method. It can also help more advanced practitioners by giving more details and insight on how to adjust their practice depending on the circumstances of the project. This research also shows that their is no one true ”one size fits all” way to go about planning architecture during a project and that is the most important lesson practitioners should remember of this paper.

The last group that can benefit from reading this paper would be everyone that does not uses agile methods. It can help clarify what agility really means, show the benefits that agility can offer, but also the challenges associated with it and that it is none trivial. Hopefully this paper also shows that correctly applied, agile methods can be very valuable to both clients and developers. This paper can also serve as a guide for more beginner agile practitioners for traps to dodge and pitfalls of agile methods.

VII. THREATS TO THE VALIDITY

This paper contains a lot of threats to it’s validity. In this section, we will discuss more in detail every threats in detail and explain why we think some of them are worst than others. Here is a list of the different threats we will be discussing.

- The experience of the authors
- The time constraint of this paper
- The number of sources used in this research
- The accuracy of the sources

A. The experience of the authors

The better part of this study was made by both student E. Asselin and V. Rodier. This is the first time either one of them has made an SLR paper thus their lack of experience may have lead to some result or statement not supported by any source. In other words, some of the statement made in this study may not be scientific, but only opinions disguised as facts. Although everything that was written in this study was read by both student and that particular attention has been made to reduce the amount of unsupported statements it is still possible that some may still be present and that the reader show be aware that not everything presented here is a supported statement.

B. The time constraint of this paper

This paper was part of an assignment given at the Polytechnique of Montreal. It means that this paper had to be made in under two months, part time, while having other course. It also means that the amount of time spent doing research and reading sources is considerably smaller than what you

TABLE I

INTERACTIONS BETWEEN FORCES AND STRATEGIES

	Requirement Instability	Technical Risk	Early Value	Team Culture	Customer Agility	Experience	Respond to change	Address risk	Emergent architecture	Big design up-front	Use frameworks and template architectures
Requirement Instability	-						triggers				
Technical Risk		-						requires use of			is reduce by
Early Value			-						triggers		
Team Culture				-			increases ability to			absence leads to	
Customer Agility					-		increases ability to			absence leads to	
Experience						-	increases ability to			absence leads to	
Respond to change	is trigger by			is increase by	is increase by	is increase by	-	is in tension with	increases ability to	reduces ability to	is increase by
Address risk		profit					is in tension with	-	reduces ability to		
Emergent architecture			is trigger by				is increase by		-	can not be use with	
Big design up-front				is useful in the absence of	is useful in the absence of	is useful in the absence of			can not be use with	-	
Use frameworks and template architectures		reduces					increases ability to				-

would expect in a normal SLR paper. There is potentially information skipped dues to lack of time or even solutions that are not present in this research because their was not enough time to cover it all.

C. The number of sources used in this research

The number of sources studied while making this research is really small. To be able to make better recommendation and have a better understanding of the problem and all the possible solutions a broader scope of previous research would be better. The time constraint and the experience of the authors explained earlier explains the reason why this paper doesn't contain more sources, but it doesn't mean that the everything is bad. The approach of this research was to focus more on a thorough study of every sources instead of having a lot of study. Unfortunately, this also mean that the generalization made in this paper is somewhat limited.

D. The accuracy of the sources

As described in the methodology section, every research used to make this paper was read and inspected by both member of this team. While inspecting the papers for the first time, one of the criteria used to select the research was the trustworthiness of the said research. Due to the inexperience of both authors it is possible that some of the paper used are less trustworthy. Also, the flaws in each previous research used to make this one are cumulated. Meaning that if one research has a flaw and we took it and used it's result to make our own, our research also has this same flaw.

But steps were taken in the hope to reducing the likeliness of those flaws. All the sources were found in reliable data base and were peer reviewed by many people additionally to making our own reliability rating.

VIII. CONCLUSIONS

This paper allowed us to answer the three questions listed in the introduction. These questions tried to help people make a decision about how much up-front architecture (if any) is needed to mitigate the risks of failure associated with the lack of planning philosophy of agile development. Here are the answers:

A. Is a project without or very little plan of architecture a project set-up for failure?

Here, the answer depends on the strengths and strategies used by the development team. As previously developed, a team with Team Culture and Experience as a strengths for a project with low technical risk can use the Respond to change and Emergent architecture strategies to reduce the up-front planning. This is only true if the customer is at an agile vision for his project.

B. Is it possible to apply the Agile Manifesto philosophy all the way and not plan anything up-front?

It will be very difficult to see impossible to do no up-front planning. Even if you have the best Team Culture and Experience and an agile customer you still need to plan. Whether in terms of the technology used, the type of architecture or the development of prototypes development teams require minimal planning to coordinate the team towards an effective solution. Moreover, agile does not mean without process and coordination. Agile means flexibility and increased communication between the development team and the customer for a product that is constantly evolving.

C. How to know exactly how much up-front architecture is needed if any?

Once again, their is not one good answer to this question. Everything depends on the specific of the project ahead. We saw that team culture, the level of agility of the client, the time before releasing value to the costumer and the scope of the project all have an influence on the amount of up-front architecture work is needed. Team culture and the experience of the team has an influence because if the team is not used to the agile method or it is too inexperienced, the amount of work up-front has to be higher to avoid making early mistakes that can cost a lot in refactoring down the line. The level of agility of the client also has an impact on the amount of work to put in at the beginning of a project because a less agile client means more planning right away and that also translate into more up-front architecture. The time constraint is easier to understand, the less time you have before you have to deliver value to the customer, the less planning you can do. And last but no least, the scope of the project. As a general rule, the bigger the scope, the more up-front work their is to do because their is a higher chance of failure with a big project with a large scope and little planning. Of course all of these are just guidelines and it does not mean that if a team does not follow these recommendation that it is doomed, all these rules do is giving your team a better chance at success!

REFERENCES

[1] M. Waterman, J. Noble, and G. Allan, "How much architecture? Reducing the up-front effort," Proc. - Agil. India 2012, Agil. 2012, no. 2007, pp. 56–59, 2012.

[2] M. Waterman, J. Noble, and G. Allan, "How much up-front? A grounded theory of agile architecture," Proc. - Int. Conf. Softw. Eng., vol. 1, pp. 347–357, 2015.

[3] J. Highsmith and A. Cockburn, "Agile software development: The business of innovation," Computer (Long. Beach. Calif)., vol. 34, no. 9, pp. 120–122, 2001.

[4] P. Kruchten, "Software architecture and agile software development: a clash of two cultures?," 2010 ACM/IEEE 32nd Int. Conf. Softw. Eng., vol. 2, pp. 497–498, 2010.

[5] R. L. Nord and J. E. Tomayko, "Software architecture-centric methods and agile development," IEEE Softw., vol. 23, no. 2, pp. 47–53, 2006.

[6] M. Schramm and M. Daneva, "Implementations of service oriented architecture and agile software development: What works and what are the challenges?," Proc. - Int. Conf. Res. Challenges Inf. Sci., vol. 2016–August, 2016.

[7] L. Chen and M. A. Babar, "Towards an evidence-based understanding of emergence of architecture through continuous refactoring in agile software development," Proc. - Work. IEEE/IFIP Conf. Softw. Archit. 2014, WICSA 2014, pp. 195–204, 2014.

[8] W. Santos, "Towards a better understanding of simplicity in Agile software development projects," Proc. 20th Int. Conf. Eval. Assess. Softw. Eng. - EASE '16, pp. 1–4, 2016.

[9] I. Hadar, S. Sherman, E. Hadar, and J. J. Harrison, "Less is more: Architecture documentation for agile development," 2013 6th Int. Work. Coop. Hum. Asp. Softw. Eng. CHASE 2013 - Proc., pp. 121–124, 2013.