Lappeenranta University of Technology
School of Engineering Science
Degree Program in Computer Science

Tatu Virta

RELATION OF LOW-CODE DEVELOPMENT TO STANDARD SOFTWARE DEVELOPMENT: CASE BIIT OY

Examiners: Professor Jari Porras
Ph.D. (Tech) Mikko Leskinen

Supervisors: Professor Jari Porras

ABSTRACT

Lappeenranta University of Technology
School of Engineering Science
Degree Program in Computer Science

Tatu Virta

Relation of low-code development to standard software development: Case Biit Oy

Master's Thesis

2018

63 pages, 8 figures, 4 tables, 2 appendices

Examiners: Professor Jari Porras
Ph.D. (Tech) Mikko Leskinen

Keywords: software development, software engineering, low-code development, Salesforce

A demand for software and skilled software developers have been increasing during the past few years. The low-code development platforms offer an alternative way to create software by using point and click tools instead of programming. The goal of this study is to find out what low-code development is, what it may be able to offer to the stakeholders and how it is different from standard software development. By interviewing employees of a Finnish Salesforce consulting company, called Biit Oy, it was confirmed that low-code has found its place as an option to develop software. When compared to standard software development, the main advantage of low-code is the speed of development, but the badly designed system may cause performance problems to the end user. Both customers and employees of Biit prefer low-code over the hand-coded solutions.

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
School of Engineering Science
Tietotekniikan koulutusohjelma

Tatu Virta

Low-code-sovelluskehityksen suhde tavalliseen sovelluskehitykseen: Case Biit Oy

Diplomityö

2018

63 sivua, 8 kuvaa, 4 taulukkoa, 2 liitettä

Työn tarkastajat: Professori Jari Porras
FT Mikko Leskinen

Hakusanat: sovelluskehitys, ohjelmistotuotanto, low-code-kehitys, Salesforce
Keywords: software development, software engineering, low-code development, Salesforce

Viime vuosina sovelluksien ja ammattitaitoisten sovelluskehittäjien kysyntä on kasvanut runsaasti. Low-code-sovelluskehitysalustat tarjoavat vaihtoehtoisen tavan kehittää sovelluksia käyttämällä ohjelmoinnin sijaan klikkailtavia työkaluja. Tutkimuksen tavoitteena on selvittää mitä low-code-kehitys on, mitä se voi tarjota eri sidosryhmille ja miten se eroaa tavallisesta sovelluskehityksestä. Haastattelemalla suomalaisen Salesforce-konsulttiyritys Biit Oy:n työntekijöitä saatiin vahvistus, että low-code on löytänyt paikkansa vaihtoehtona kehittää sovelluksia. Verrattuna tavalliseen sovelluskehitykseen sen pääasialliset vahvuudet ovat kehityksen nopeudessa, mutta toisaalta huonosti suunniteltu järjestelmä voi aiheuttaa suorituskykyongelmia loppukäyttäjälle. Sekä asiakkaat että Biitin työntekijät suosivat ensisijaisesti low-code-työkaluja koodin sijaan.

ACKNOWLEDGEMENTS

Jari Porras & LUT Tietotekniikka
Mikko Leskinen, Ikla Puustinen & Biit Oy

Ykkösketju, #paavo4ever & #ssn
Golden Boyz
Cluster ry & Cluster Alumni ry
Polytekninen Willimiesklubi PoWi ry & Giorgio Hässi
Lappeenrannan teknillisen yliopiston ylioppilaskunta LTKY
ATK-kerho Ruut & Lappeenranta Academic Gamers LAG
Tekniikan Akateemiset TEK
EI-13 & FV-13-14
STISS 2016 & SeoulTech '17

mom, dad & Siiri

and of course, you, my dear Miia.

Cheers!

Tatu Virta
Helsinki, 14.08.2018

Mmm, whatcha say~?

TABLE OF CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

4GL	A fourth-generation programming language
AD&D	Application development & delivery
AI	Artificial intelligence
API	Application programming interface
CMS	Content management system
CRM	Customer relationship management
DSL	Domain-specific language
Id	Identity
IEEE	Institute of Electrical and Electronics Engineers
IoT	Internet of Things
IT	Information technology
LCDPs	Low-code development platforms
NATO	North Atlantic Treaty Organization
PaaS	Platform as a Service
PoC	Proof of Concept
RAD	Rapid application development
RUP	Rational Unified Process
UML	Unified Modeling Language
UI	User interface
UX	User experience
WYSIWYG	What you see is what you get

1 INTRODUCTION

The software industry is huge. According to the Financial Times Global 500 rankings [1], in 2015 4 out of 10 of the world's largest companies by market value were at least partly software companies. According to the Forbes business magazine [2], in 2018 4 out of 10 of the world's wealthiest people were from the software industry. The software as we know it started as punched cards, which were loaded into huge computers, moved from minicomputers to microcomputers and survived through the dot-com bubble. Nowadays software is a part of our everyday lives, thanks to products such as iPhone and Android smartphones. Software development as a modern phenomenon was started by a few visionaries and it wasn't something that could be done by non-experts [3]. However, partly because of different mobile app builders as well as larger low-code development platforms, software development might have become easier than it has been ever before.

Software engineering is a process, which includes specifying, developing, validating and evolving the software [4]. Software development process' outcome is called source code. Of course, for some the software development could be just a hobby, but most often the source code is a product, which creates wealth for the developer and makes some task easier for the paying customer. There are many different processes, models, and methods on how to create software properly [4]. These process models, such as the waterfall model or more recent agile approaches, has always included a developer, who ultimately writes the code and therefore creates the product. The traditional software development process could be rather expensive and it doesn't get any cheaper if there are new laws affecting the developed software, or even if there are some changes in company's internal processes, which forces the customer to make changes to their software. These are some of the problems that the idea of low-code development tries to tackle [5].

Low-code development is a way to develop software. By creating low-code tools the software can be developed by using both declarative techniques and a lower amount of code than traditionally in the software development [6]. In theory, this kind of development could achieve many benefits, such as easier maintenance of the product [5]. However, there are some pitfalls, such as governor limits created by the PaaS (Platform as

a Service) provider, so just one application couldn't hog all the cloud service's resources [7]. The creation of complex applications using low-code tools could be even more time consuming than with code editor.

One example of such platforms is Salesforce. Essentially, Salesforce is a cloud-based CRM (customer relationship management) platform, but according to the company itself, "saying that Salesforce is "just a CRM" is like saying a house is just a kitchen [8]." Salesforce is an interesting product, but when thinking about the software development point of view, the most interesting parts of it are the low-code development tools which it offers to users for free of charge. With these tools, the developer, or whoever wants to do the development part, can create different applications, such as company's objective and feedback management application, or just smaller functionalities, such as data validation rules or some event-based triggers. Some smaller features can be developed without a single line of code at all but some, more complex applications, might need some previous programming experience and knowledge from the developer.

1.1 Goals and delimitations

The goal of this thesis is to find out what low-code development is, what it may be able to offer to the software developers as well as to the paying customers, what kind of platforms offer low-code development tools, and how low-code development is different from standard software development. The research will be done as a case study on a Finnish consulting company, called Biit Oy, and therefore the main platform on which this thesis concentrates is Salesforce. More about Salesforce and Biit Oy can be found from chapters 2 and 3, respectively.

This thesis attempts to answer the following research questions:

- RQ1: What are the current benefits and pitfalls of low-code development compared to hand-coded solutions?
- RQ2: What are the customers' expectations of low-code development solutions?
- RQ3: Which, in the case company's view, is the better software development method?

The research questions are answered by reviewing related research and by interviewing the case company's employees. More about the research methodology can be found from the chapter 3, where the used questionnaire and the backgrounds of the interviewees are presented.

1.2 Structure of the thesis

This thesis has been divided into two parts: theoretical and empirical. Chapter 2 gives the reader an overview of software development, software engineering, low-code development, and low-code development platforms, and takes a deeper look at one such platform's, called Salesforce, low-code development tools. Chapter 3 introduces the case company and research methodology used in the case study. Chapter 4 covers the results of the data collection and introduces the found themes of the held interviews. Finally, in chapter 5, the discussion and conclusions are presented.

2 SOFTWARE DEVELOPMENT

Almost everyone has written computer programs during their lives. The written program might be just a small script written as a hobby, some spreadsheets to simplify some complex calculation tasks or a complete space rocket software which controls the entire shuttle. When talking about software development, in most cases we mean the professional activity where software is developed for business purposes [4]. When the definition of this task is simplified, it could be said that the paying customer pays to the developer, or the developing team, which then develops a software product. This product includes the source code of the program as well as the documentation related to it.

Software products can be divided into two categories: generic products, which are stand-alone systems that the organization is able to sell on the open market and they also control the software specification, and customized products, which are developed for the specific customer, who, instead of the developing organization, controls the specification [4]. However, this categorization of software products is becoming blurred, because it is possible to customize generic products to fit an organization's needs.

2.1 Software engineering

Defined in the IEEE (Institute of Electrical and Electronics Engineers) Standard 610.12 [9], software engineering is "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software, and the study of such approaches." In other words, software engineering includes [3]:

1. Methodologies to meet customers' needs.
2. The philosophy of engineering.
3. Possible use of mathematics.
4. Project and quality management.

The birth of software engineering happened in the late 1960s when NATO (North Atlantic

Treaty Organization) Science Committee organized two conferences considering the topic. In these conferences, the term “software crisis” was used to refer to problems in software development [3]. These problems included challenges software development companies are facing even today: budget and schedule overruns of projects as well as quality and reliability issues of the delivered software. It was understood that programmers are like engineers when they first design and then build the products [3].

According to Sommerville [4], software engineering is “an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use.” Therefore, software engineers must take ethics into consideration as well. It goes without saying that the one should uphold normal standards of honesty and integrity, but also that the one shouldn’t use his or her computer skills and abilities to make harm to others. Basic principles, such as confidentiality, competence, and intellectual property rights might not be laws, but they are part of an engineer’s professional responsibility [4].

The key challenges in software engineering are the coping with increasing diversity and delivering high-quality and trustworthy software to the customers, while there are constant demands for reduced delivery times [3][4]. Organizations must estimate the time and effort it takes to develop the software. This estimation, of course, affects the price as well. Another challenge which software engineers are facing is related to the risk management. The developing organization must identify potential risks and actions to eliminate them, or at least reduce the probability of occurrence of these risks [3].

2.2 Software processes

A software process is an either structured or rather loose process, which consists of activities, which are related to each other. By following the process, the software developing team can build the software from scratch using standard programming languages. However, nowadays all the business applications don’t need to be developed by just coding, because many of them can be extended or modified by configuring or integrating them with other software [4]. Like mentioned, these four activities are, at least

in some form, part of every software process: specification, design and implementation, validation and evolution.

2.2.1 Software specification

Software specification, which is described in Fig. 1, and is also known as requirements engineering, is the first activity of the software process. The goal of this activity is to produce a requirements document, where all the requirements for the software are listed and explained. Requirements engineering starts with a feasibility study, where it is estimated if the user needs can be satisfied using current software and hardware technologies. It should also be considered that would the system be cost-effective enough. Right after the feasibility study requirements engineering continues with requirements elicitation and analysis. During this step, the system models are made by observing the existing systems or discussions with potential users and procurers. [4]

Third and fourth steps of requirements engineering are called requirements specification and requirements validation. During the third step, all the information gathered during the previous steps are put into a document that defines a set of requirements. There may be two types of requirements: abstract statements of the system requirements for the customer and more detailed description of the functionality for the developers. The last step of this activity is for validating the requirements for realism, consistency, and completeness. It is also worth noticing that requirements engineering is a particularly critical stage of the software process as errors at this stage inevitably lead to later problems in the system design and implementation. [4]

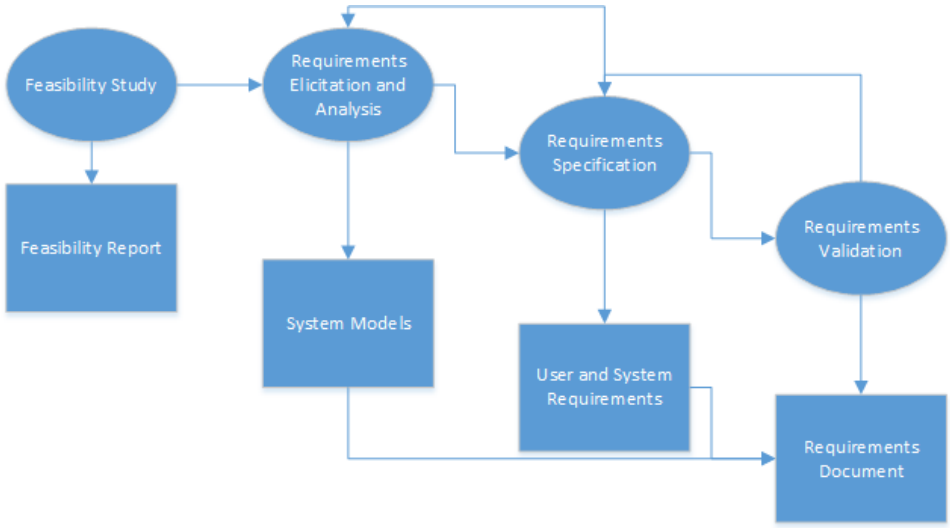

Fig. 1. The software specification process. [4]

2.2.2 Software design and implementation

Software design and implementation is a creative process where the program is actually first designed and then implemented according to the system specification. A software design phase, which is described in Fig. 2, can be divided into four steps: architectural design, where the overall structure of the system is identified; interface design, where the interfaces between system components are designed; component design, where each system component are designed; and database design, where the system data structures are designed. The detail and representation of the outputs of these steps may vary considerably depending on the criticality of the system as well as the used approach. Programming itself is more of a personal and creative activity and there are no guidelines on how to complete the task. After the programming programmers handle the unit testing and debugging of the components they have created. [4]

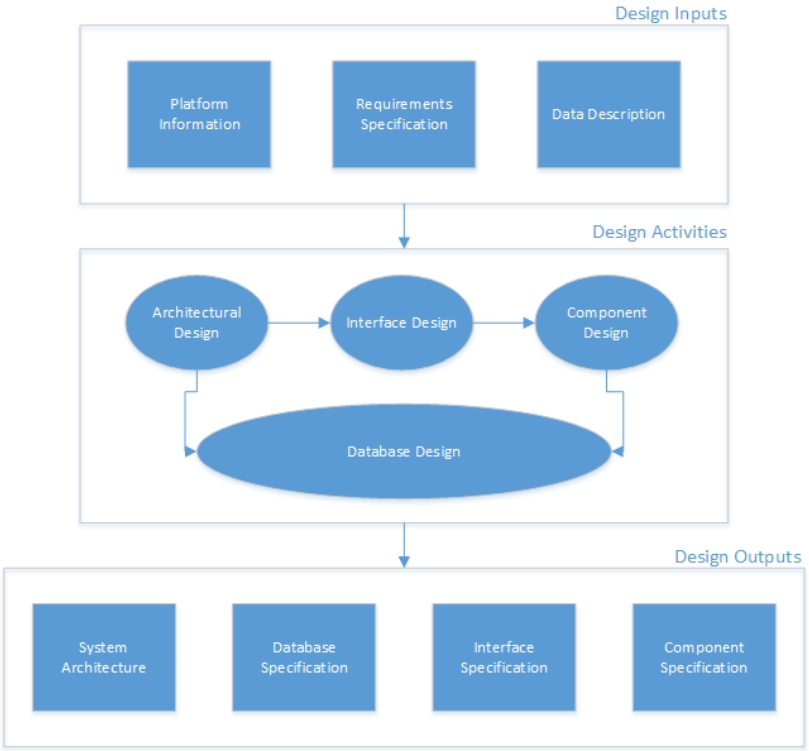

Fig. 2. The design process. [4]

2.2.3 Software validation

After the software has been implemented, it needs to be tested to show that a system both conforms to its specification and that it meets the expectations of the customer. The testing process is usually divided into three stages, as seen in Fig 3: component testing, system testing, and acceptance testing. In the first stage, which is also called a unit testing stage, the components of the system are tested by the developers themselves. Each component is tested independently, without other system components. This is the economically sensible approach, as the programmers know the components and they are able to generate proper test cases. In the second stage, the system components are integrated to create a complete system. The goal of this stage is to find errors that result from unanticipated interactions between components and component interface problems. This stage is usually completed by a separate testing team with different data than used in the previous stage. Finally, in the third stage, the software is tested either by the testing team or by the customer using the customer's own data, which shouldn't be simulated, like it might have been in the two previous steps. Acceptance testing may reveal errors in the system requirements definition

or even issues in the system's performance. If defects are found, the program must be debugged, which may require other stages of the testing process to be repeated. [4]

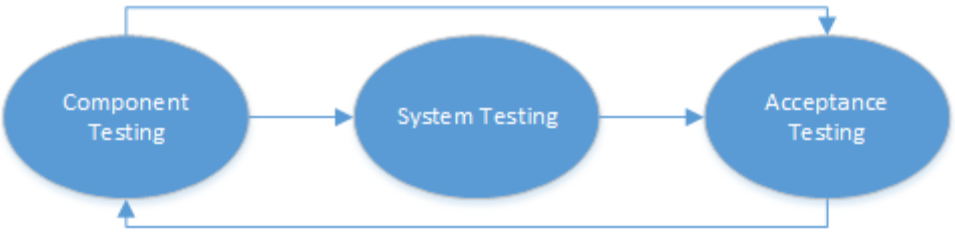

Fig. 3. The testing process. [4]

2.2.4 Software maintenance and evolution

It would be easy to think that software maintenance and software evolution are the same things. Where the term maintenance implicates on something that is done because of the software itself is deteriorating, evolution means that there are some changes that need to be done because of the requirements or the environment of the software has changed. When we are talking about the software maintenance, we mean some small bug fixes or patches, which help the system or software to keep running and providing benefits to the users. However, sometimes it is necessary to do some bigger changes, for example, if the user base or the purpose of the whole software has changed over time. [10]

Change is unavoidable. According to Robert L. Glass [11], the maintenance costs are averagely 60% of the software costs, and this enhancement is responsible for roughly 60% of the software maintenance costs, although it is possible that the maintenance process is less challenging than the original software development. Sometimes the maintenance is not enough and the new system is required. If the old system still provides something to the users, it can be left as a legacy system, but the migration to the new one is often encouraged and ultimately even forced. For the sake of the unavoidable change, during the implementation phase, it is important to make sure that the software is easy to maintain, extend and even evolve, so there would be no need to design and develop everything from the start. [4] [10]

2.3 Software process models

It could be said that a software process model is a simplified representation of the whole software process. They give guidelines and certain structure for the development team how to act on different stages of the software development. It is important to notice that these are just models and they are usually modified for the teams according to different factors, such as deadlines, budget, team size and the criticality of the software itself. There are as many software process models as there are software projects, but these three, the waterfall model, the RUP (The Rational Unified Process), and the agile software development, could be considered the main models or they are at least the most famous ones.

2.3.1 The waterfall model

The waterfall model was the first actual software development process model. It was introduced in 1970 and it has taken inspiration from other engineering processes. The model has 5 stages: requirements definition, system and software design, implementation and unit testing, integration and system testing, and operation and maintenance. This model is an example of a plan-driven process, where you must plan and schedule all the process activities before starting to work on them. Like the name suggests, there is a cascade from one phase to another, as seen in the Fig. 4. In theory, the following phase should not start until the previous one has finished, but in practice, these phases overlap and feed information from one to another. After each of the phases, some documentation is made. This documentation can be used in the next stage of the process. It is also possible to make some alternations to the documentation, for example, if some technical limitations or new use cases appear. Although the model is old, it is still widely used, mainly because it is easy to understand, easy to use, and it might feel familiar to the customers from other fields and projects. It is stated that the model works well if the requirements are well understood and unlikely to change during the system development. However, because the process is a bit stiff, iterations can be costly and involve significant rework, which could lead to problems in delivery speed and budgeting. [4]

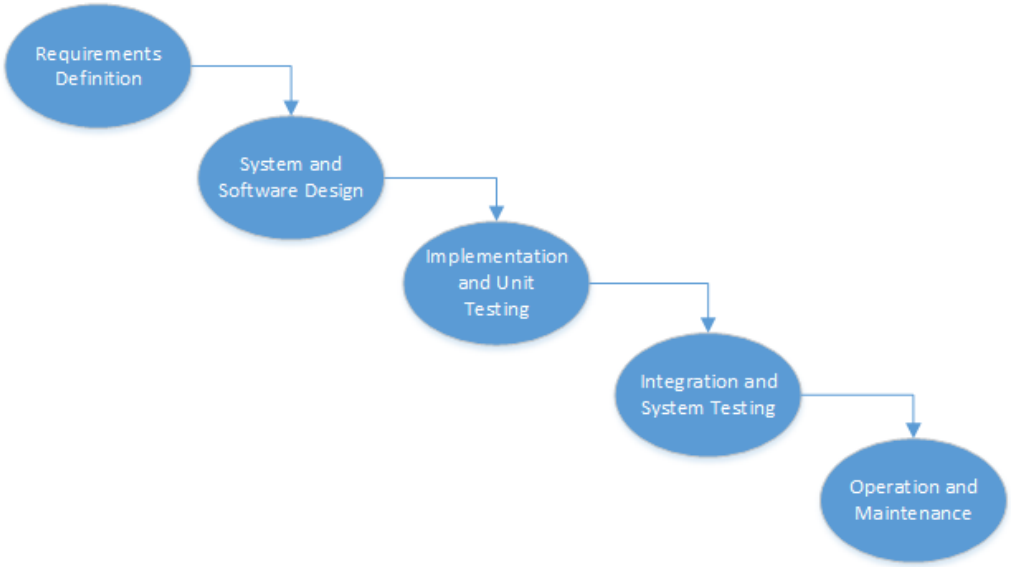

Fig. 4. The waterfall model. [4]

2.3.2 The Rational Unified Process

The RUP is a modern generic process model, which has been influenced by older process models, such as the previously mentioned waterfall model. It was created and published in the late 1990s by the Rational Software Corporation. The idea for RUP was to work well with UML (Unified Modeling Language). The RUP can be described from three perspectives: a dynamic perspective, which shows the phases of the model over time; a static perspective, which shows the process activities that are enacted; and a practice perspective, which suggests good practices to be used during the process. [4]

The RUP is divided into 4 phases, which are closely related to the business rather than the technical concerns. The first phase is the inception, where the goal is to establish a business case for the system. By identifying all the external entities that will interact with the system, the team is able to figure out the contribution that the system makes to the business. The second phase is an elaboration, where the goals are to develop an understanding of the problem domain, establish an architectural framework for the system, develop the project plan, and identify key project risks. The third phase, construction, is about system design, programming, and testing, where the goal is to have a working system and documentation ready for the delivery. The last phase is called transition, where the system is moved from the development team to the customer and making it work in the

real environment. In the RUP the whole set, or alternatively each phase separately, may be enacted in an iterative way, as shown in the Fig. 5. [4]

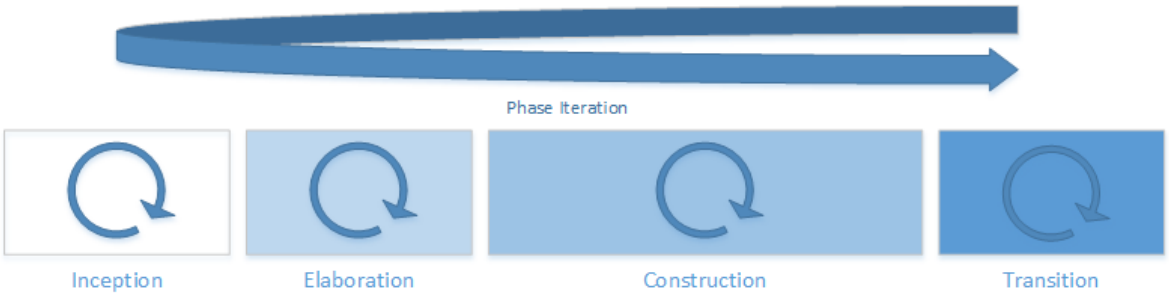

Fig. 5. Phases in the RUP. [4]

The activities in the RUP are called workflows. The most important innovations in the RUP are the separation of phases and workflows, and the recognition that deploying software in a user's environment is part of the process. Workflows include the same activities from every other software process models, such as requirements, analysis, and design, implementation, etc. All the workflows may be active at all stages of the process, but of course, some workflows are emphasized more in different parts of the development than others. [4]

2.3.3 Agile software development

In the late 1990s, many software developers were unhappy while working with heavyweight software development processes. All the time could be spent designing and documenting the software, while the quality of the program itself was poor, deadlines were missed and budgets went bigger than initially expected. To tackle this problem a so-called Agile Alliance was formed. This group of industry experts wanted to find values and principles that would allow software teams to work quickly and to respond to change more efficiently. The result of this group's work was the Manifesto for Agile Software Development [12], which consists of 4 main values: individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan. [13]

While different tools and processes are important to a certain degree, it is said that the most important ingredient of success in software development is the people, which in this context means the team working with the software. Project work is all about communication. When building a team, it is more important to have people who understand each other and can communicate with each other, than a group full of randomly selected programmers. For every person, there is a right team and for every team, there are the right tools. Like already mentioned, tools may help with the development, but they shouldn't be in a way of creativity. [13]

A software consists of two parts: the program itself and the documentation related to it. However, it doesn't really serve anyone if the development team writes huge piles of documentation just because it has been ordered to do so in the development process. The document is useless and purposeless if no one understands it or it doesn't provide anything new to the reader. It is more important to create a working program which provides something to the user. Of course, the program isn't software without the documentation, but the documentation should be both short and salient. The code itself, if it has been written well, could be a documentation on its own [14]. In many cases, both comments between the code and the main documentation don't get maintained as often and as quickly as the code itself, so they will only lie about the software. Besides the code itself, the most important document is the team, which transfers information to the new team members. It is always faster and more efficient to change knowledge from human-to-human, rather than by reading the documentation. [13]

A proper customer collaboration is often hard to achieve. The ordered, customized software isn't a piece of a product like a plate or a hammer from the store. It is pointless to give a development team a description, a fixed schedule, and a fixed price, as these attempts to develop a software usually fails. The software development requires continuous interaction between the development team and the customer. The development team creates a small feature for the software and the customer provides feedback by answering how the feature could be developed further. By doing this the software gets developed both iteratively and incrementally, piece by piece. This kind of development requires a certain input from the customer as well and it is important that they understand that software

development is different from the other engineering sciences. [13]

The final value of the manifesto is the ability to respond to change. It is impossible to plan the course of a software project very far into the future. We are living in a rapidly changing business environment, which causes the requirements to shift from the initial ideas. Also, when the system starts to get the shape, the customer usually understands new things about the possibilities what the software can provide. A good strategy is to create a plan for the next two weeks and then cope with the unavoidable change of requirements. Again, by building the software piece by piece, it will finally achieve its final purpose. The best of those are the ones that fulfill the customer's needs they didn't even know existed, not the ones they initially wanted to satisfy. [13]

The agile software development and the agile manifesto aren't methods themselves. They just summarize the values and principles behind each agile software development method. While every agile method has their own features, which make them special, the main idea is the same. When compared to more traditional methods, the software is made incrementally in short iterative periods, where the face-to-face communication inside the team and between the team and the customer are the key points of success. [13]

2.4 Low-code development

The low-code development doesn't have a general definition, but it isn't a completely new concept either; it's simply a practice that has never been defined before. According to Revell [15], the low-code development is "a way to design and develop software fast and with minimal hand-coding", or in the more abstract way: "a way for developers to get more done." The low-code development's roots can be found from 1982 when a concept of 4GL (a fourth-generation programming language) was introduced by James Martin [16]. 4GL includes database query languages, information retrieval systems, report generators, application generators, and very high-level programming languages. These languages could be used by the end users and they support application development without programming — hence the name of the book. After 4GL there has also been the RAD (rapid application development). Both of them are based on the same idea: model-driven

design and development, automatic code generation, and visual programming. However, the low-code development can be seen as different from these, because the targeted category of users is different [17].

The term “low-code development” was invented by an American research company called Forrester Research, Inc. According to Forrester’s analyst Clay Richardson, origins of the term trace back to a report published in 2011, called *The New Productivity Platforms: Your Solution To The AD&D Crunch*, but the first time it was used was in a report called *New Development Platforms Emerge For Customer-Facing Applications* in 2014. In their report [18] Richardson and Rymer argued that the so-called customer-facing applications are now the top technology priority in many enterprises. Developing applications for customers is different than the enterprise application development, as seen in the Fig. 6. The weight of the project is given to the product itself and to the collaboration between the development team and the customer rather than to the structured process. The delivery speed has perhaps always been the biggest challenge in the software development. Now, when the customers and markets define the speed of the business, the software development and delivery must keep up. This means that the project schedules have to be cut to weeks, while updates and fixes have to be ready just in days. Also, while customers demand faster delivery, they want the quality of the software to be very high. On top of that, it is not enough if the functionalities are great, the software has to also be intuitive, which means that there should be no training required in order to use the software and get some value out of it.

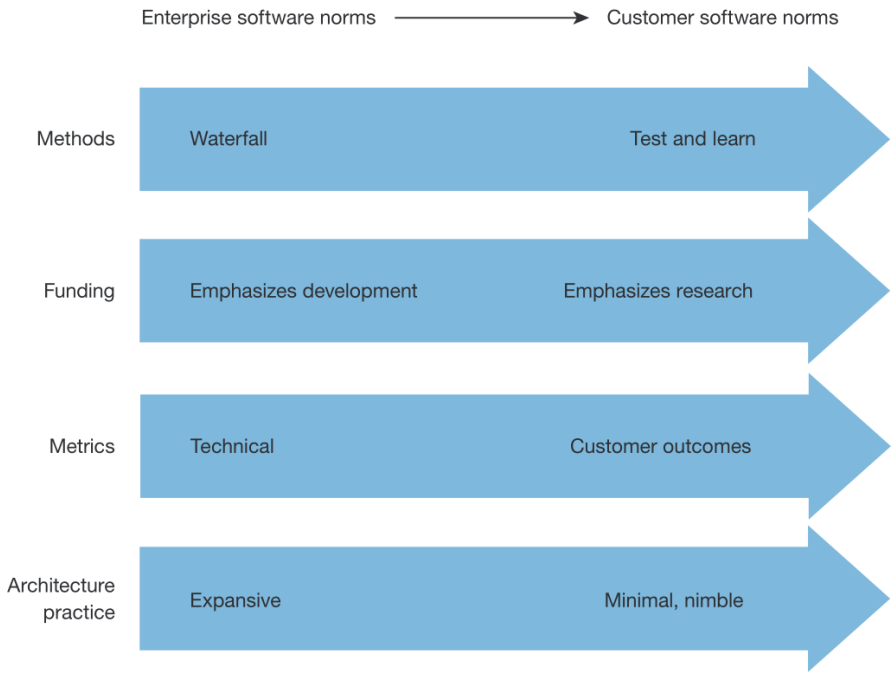

Fig. 6. The difference between enterprise software and customer software norms. [18]

These customer-facing applications tend to start from shaky ideas with uncertain requirements and critical features. If the company tries to hold onto the stiff and slow approaches in software development, the new promising ideas might get rejected, which could cause the company to fall behind from its competitors. There are some alternatives to the coding, like outsourcing some parts of the development process, using packaged applications, buying some specialized middleware, or even organizing hackathons, but none of these help with the actual problem: how can the application delivery teams speed up their useful output. These alternatives are either slow, inflexible, or the development team can't handle the customer experience. The development teams need faster development techniques, like visual development, automatic configuration and deployment, or user interface transcoding, which can speed up the delivery process compared to the hand-coding. With the tools like these, the project's value could be proven faster. [18]

In Forrester's second and third report in 2016 [19] and 2017 [20], respectively, it is mentioned that the low-code development has become an important way for the technology management and enterprise architecture to quickly deliver software to win, serve, and

retain customers and to keep that software evolving. A drought of development talent has often left the business' insatiable thirst for new web and mobile application unsatisfied. According to The Finnish Information Processing Association, TIVIA, which is an independent association of Finnish ICT professionals and companies, there is a shortage of IT professionals in Finnish job markets [21]. The same kind of trend can be seen in the other Nordic countries as well. [22]. According to the survey results from the US-based job search site, 86% of respondents said that it was "challenging to find and hire technical talent." 83% of respondents stated that their difficulty finding qualified candidates has harmed their organizations' revenues, product development, market expansion efforts, and the job satisfaction of the current employees [23]. While these kinds of problems can't be tackled by the low-code development only, it may increase the talent pool when searching the developer candidates, because the candidates don't need strong programming skills to build software, just general computer skills and domain expertise might be sufficient. Therefore, the low-code development is seen as a new, more attractive choice because it may help the business and tech management as well.

2.5 Low-code development platforms

The low-code development happens on the LCDPs (low-code development platforms). According to the original definition of LCDPs, given by Forrester in 2014 [18], these platforms "enable rapid application delivery with a minimum hand-coding, a quick setup and deployment, for systems of engagement." In 2017 [20] Forrester updated the definition as follows: "low-code development platforms are products and/or cloud services for application development that employ visual, declarative techniques instead of programming and are available to customers at low or no cost in money and training time to begin, with costs rising in proportion of the business value of the platforms." Nepal [24] even argues that LCDPs are like the next step from CMSs (content management systems), like WordPress, which are made for web development, except that with LCDPs, you are making full applications, not just web pages. They initially sprang up as just auto code generation tools but has evolved since then into the enterprise-grade app development platforms covering the entire app delivery.

LCDPs rely on declarative development tools instead of programming languages. Visual and declarative development tools have been available in various forms for decades, and the vendors have applied those techniques to both narrow and broad application development. These tools could be DSLs (domain-specific languages), WYSIWYG (what you see is what you get) UX (user experience) definition, flow diagramming, or visual data modeling. Even with the agile development approaches, creating new applications by hand-coding continues to be both time-consuming and labor-intensive process. With the listed tools, the development team will be able to develop custom applications much faster than with programming languages. They also open a possibility to quickly test a business idea in a working app. Companies have reported that visually configuring new applications using these platforms has helped to speed up their software development up to 5 to 10 times compared to standard software development, as seen in a Fig. 7, and allowed the business to provide real-time feedback on the resulting applications' functionality [19] [20]. In today's markets, according to Forrester, "quickly building and launching new digital products and services that deliver convenient, simple and engaging customer experiences is a key lever."

Enterprise	Result	Code	Low-code
US government (Affordable Care Act)	Document compliance module	100 person-months	5 person-months
British insurance provider	Agent portal	Unknown*	10 days to minimum viable product (MVP)
Call center operator	Customer-specific app	4 months	3 weeks
Spanish insurance provider	Web channel and administration system	2.7 years (estimated)	13 weeks

*The project was on the technology management backlog list for years with little hope of ever starting

Fig. 7. Comparison of different software projects made by hand-coding and using low-code development platforms. [19]

In 1998 [25], Apple's late CEO Steve Jobs said that "people don't know what they want until you show it to them." Today, when there's an endless stream of product choices in the software market, building a new custom application for a customer has become rather annoying. Like already understood in the agile software development [14], people often change their minds after seeing the product and its features in action. They could

understand that their visions might not work in the way they thought they would, either because of the limitations in the technology or in the habits of the common workflow. Therefore, it is often better to quickly build minimum viable products to validate ideas and customer requirements before wasting resources in features and functionalities that customers may not even value. LCDPs allow companies to quickly translate ideas into the low-cost working prototypes that they can deploy and test in the market. Of course, the prototypes could be built by just hand-coding them, but these prototypes may need to be rebuilt as the volume and diversity of users increase. With low-code, the platform itself handles the quick deployment, scaling, and performance-tuning of applications.

2.5.1 Features and benefits

Although declarative software development tools have been around almost since the time of 4GL languages, LCDPs are different. They are more open and more easily integrated with other systems. Before the current generation, these tools and platforms used to be completely proprietary and thus tended to be isolated and difficult to connect into broader application portfolios. LCDPs, too, employ many of these elements, but they also provide features, like platform APIs (application programming interface), and most of them are compatible with open source development frameworks, most notably AngularJS and React. It could be said that the current LCDPs are more complete as platforms because they are not just tools, they are complete platforms. Features like application deployment and application life-cycle management set the current products apart from those that preceded them. [19]

The application development with LCDPs is fast. They allow the team to succeed with less-skilled developers by minimizing the hand-coding and therefore speeding up the delivery. They may also eliminate barriers to customer participation in projects. It is easy to gain quick feedback from the customer, but also from other employees and partners as well, thanks to the so-called test-and-learn approach, where the application can be shown to the customer on the early stage of development. This removes the guessing about the requirements because the requirements could be tested in action on a very early stage of the development. Sustainable digital transformation requires a build-once, adaptively re-

use-many-times approach to the application development that dramatically simplifies the software development and maintenance. It makes it easy to scale across the enterprise and adapt to the business changes. The most if not all platforms provide responsive design, mobile-ready functionality and offline support by default. Everything is in the unified and centralized environment, from user authentication and UI (user interface) to repository control and role-based access data. [18]. Nepal from KiSSFLOW identifies [24] five the most important features of a good LCDP as follows: easy visual configuration, ready integration options, mobile compatibility, scalability and full life-cycle support, which are all, according to Forrester's reports, parts of the most successful LCDPs [6].

Van Schetsen from SD Times [26] argues that development with LCDPs doesn't follow the traditional "meeting-programming black box-repeat" flow, because they make it possible to have a near-real-time collaboration and exchange of ideas. Applications are more than just the code, they're driven by the logic itself, as the business rules are "baked in" to the application at a core level. Also, developers' productivity may increase as they can spend less time writing code and more time creating objects representing the functionality of many lines on the code. With simplified and centralized systems like LCDPs, modifications and other maintenance tasks can be done in one part of the system only and they propagate throughout the who system. Part of the innovation of these low-code platforms has been to relax these constraints so that schema changes can be made without the user having to worry about impacting performance. De and Jones see [27] that one of the challenges, but also the benefits, of the LCDPs, is the security of the applications. Fortunately, also the user permissions can be set in place by an administrator without any requisite knowledge of the programming language. The streamlined and code-free approach of this functionality makes it easy to prevent users from seeing data they are not supposed to see.

There have been a few myths about the LCDPs and the low-code development in general. One of these has been that the LCDPs are only for so-called citizen developers, not pro developers. According to Forrester's research [19], developers use LCDPs to create tools for citizen developers, who are then able to deliver apps. Another myth is that LCDPs would eliminate the need for any programming. While it is possible to write some, but

often relatively simple applications without having to write any code, still anything more complex must be done by hand-coding. These include features such as integrations and custom algorithms. Some sources [28] have even been talking about a phenomenon called no-code platforms, but they are seen just as an evolution of LCDPs, or more like a special event inside LCDPs, when already made and customized components can be used with, for example, drag-n-drop tools, to create new applications. Overall LCDPs should be seen as new possibilities, not something that would jeopardize the work of the professional developers. In a survey about the RAD (rapid application development) [29], 74% of IT professionals saw the no- or low-code as a key when evaluating the RAD platforms. In the same survey, 56% expected that the amount of the no- and low-code applications will increase. However, as stated in the SD Times' column, it is worth noticing that in the end LCDPs are not the answer to every programming challenge and there will always be a place for more traditional frameworks. The next new big innovations awaited to be integrated into LCDPs are the sensors and actuators of IoT (Internet of Things) as well as the AI (artificial intelligence) [19] [20].

2.5.2 Possible pitfalls

LCDPs are not perfect and while they have received praise from multiple directions, some are against them. Reselman explains [30] why the idea of thinking that LCDPs make everyone a professional programmer is a dangerous thought and why the current generation of LCDPs is just another fad like 4GL and Visual Basic programming was before it. He argues that companies don't accept that making software is and will always expensive, hard and that anyone just can't write code. It is always cheaper to do the software well instead of fixing it later. It doesn't really matter which tools the developer is using; the software development project will fail if the developers are not high skilled enough. Behind every successful software development project, there must be someone who understands how things are working beyond the LCDPs. There must be someone who understands the basics of software engineering.

One of the biggest issues of many LCDPs is that they don't actually eliminate code; they just hide it from the end user [23]. Creations of visual tools have to be converted,

compiled, and executed just like normal, hand-coded software. Idesis sees [31] that, unlike according to Forrester's reports, LCDPs are just black box tools, which may be able to solve some simple business cases but doesn't really remove the need of hand-coding at all because of all the limitations. No one should run any critical services on LCDPs, because they wouldn't really have a total control over how they work. LCDPs always offer a possibility to fall back to hand-coding when some more complex logic has to be done. This is very unsatisfactory because companies can't still replace the real software developers who understand the fundamentals of software development [32]. Although LCDPs may not be the answer to eradicate the need for skilled developers, it still makes the code a bottleneck because of all the speed of development it is able to offer.

2.5.3 Low-code development platform markets

LCDPs were initially [19] divided into 5 segments: general-purpose platforms, process app platforms, database app platforms, request-handling platforms, and mobile-first app platforms. A year later in 2017, Forrester found out [20] that these segments have blurred as the most if not all LCDPs have become general-purpose platforms. Two new segments were recognized, defined by a target buyer as well as the vendor's ambition to influence who delivers applications and how they deliver them: platforms for AD&D (application development & delivery) pros and platforms for business developers. The former provides a more productive alternative to programming, or in other words a strategic alternative to Java and .NET. The latter is for the nontraditional developers, who can create applications with simple and direct tools. More accurate comparison of these two segments can be seen in Table 1.

Table 1. Comparison of LCDP segments. [20]

	Low-code for AD&D pros	**Low-code for business developers**
Goal	Speed the software delivery by making pro developers more productive.	Speed the software delivery by letting business experts drive it.
Target buyer	AD&D leaders.	Business leaders.
Designed for	Developer control and flexibility.	Simplicity to empower non-traditional developers.
Amount of use cases	Many.	Narrow range.
Targeted app size	Medium-sized to large.	Medium-sized
Provides tools for	Business experts.	AD&D and IT (information technology) pro support.
Preferred message	Low-code.	No-code.
Vendors	Appian, Mendix, OutSystems, and Salesforce.	Caspio, Kintone, MatsSoft, and Nintex.

According to Forrester's estimations, the growth of LCDP markets is accelerating as large enterprise companies have started to appreciate LCDPs as tools for more complex problems. The revenue of the tracked 67 vendors reached $2.6 billion in 2016 and was estimated to reach $3.8 billion in 2017. The acceleration is estimated to be growing until the year 2020 when the rate would be 55%, as can be seen in Fig. 8. It is worth noticing that market is still fragmented, as the most low-code vendors have annual revenues less than $100 million, Salesforce being the only vendor with $1 billion-plus revenue. The trend seems to be that the new platforms target narrow use cases to establish themselves. There have been predictions that the traditional IT enterprises, such as Dell, Microsoft, Oracle, IBM, and SAP, will join the market as well, most probably by buying smaller vendors instead of creating new platforms themselves, because building an LCDP from scratch could be beyond the capabilities of even the largest vendors. [20]

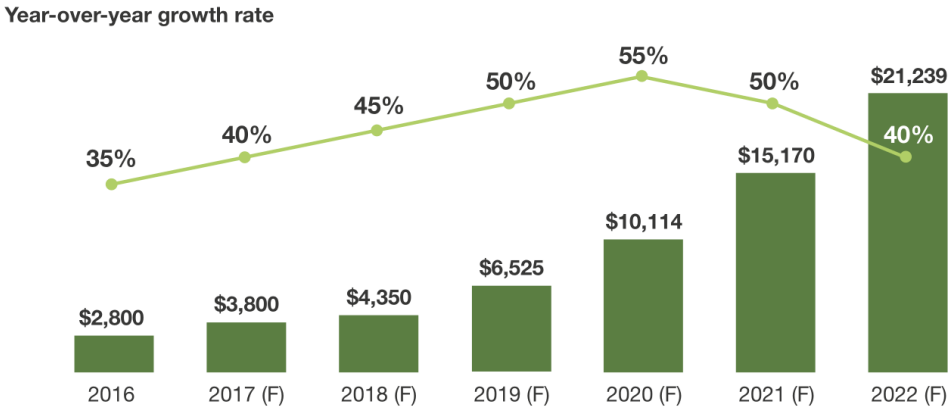

Fig. 8. Predicted LCDP market growth. [20]

2.6 Salesforce

Salesforce is a cloud-based CRM platform made by an American company called Salesforce.com, Inc, which was founded in 1999. With a revenue of over $8 billion in 2017, it is by far the biggest LCDP of all [6]. In 2017 and 2018 Salesforce was named "leader of low-code application development platforms" and "leader for enterprise high-productivity application platform as a service" by Forrester [6] and Gartner [33], respectively.

In Salesforce, the data is modeled using objects, fields, and records [34]. Salesforce comes with multiple different standard objects, such as Account, Contact, Lead, and Opportunity, but administrators can create custom objects and fields as well to fill up their companies' business needs. Every object, standard or custom, has a few mandatory fields, such as Id (identity) which is created by the platform automatically, and Name, which is required to be chosen by the one who creates the record. Administrators may choose from a selection of custom field types, such as checkbox, date, currency, and formula. Administrators can also create custom relationships between objects, either master-detail relationships, which mean that the detail-object is dependent on the master object, or lookup relationships, which just mean a rather loose relationship between two objects [35]. Data access on Salesforce is handled by multiple flexible layers [36]. With these layers, the administrator can open access based on users' profiles from the login to objects and fields. On top of that, record-level access can be controlled either with organization-wide defaults, role

hierarchies, sharing rules, or even with manual sharing.

In Salesforce there are 3 kinds of low-code development features: field tools, automation tools, and builders. Field tools include formula fields [37], roll-up summary fields [38] and validation rules [39]. Formula fields share the same features as Microsoft Excel's formulas. With them, developers can express rather simple calculations with different conditions to create a chosen output, such as text, number or currency. One use case for formulas is to get some data from related records using the created custom relationships between objects. While formula fields calculate values using fields within a single record, roll-up summary fields can calculate values from a set of related records. The developer may choose 4 different types of summaries: the number of related records, the sum of some field, the lowest value of some field or the highest value of some field. Roll-up summary fields work only on the master record of the master-detail relationship. The final tool of this category is the validation rules, which doesn't work like the other two. A validation rule contains a formula or expression that evaluates the data in one or more fields. By using validation rules the developer can easily control the quality of the data, for example by forcing the length or form of some input string.

Perhaps the most complex low-code tools in Salesforce are the process automation tools, which are named Process Builder [40], Cloud Flow Designer [41], and Approvals [42]. For every automation tool, there are a few sample scenarios listen in Table 2. The first, Process Builder, is typically used when the developer wants to automate something, which happens behind-the-scenes from the user and where the user input isn't needed. The process created with Process Builder can be started when a record is created, removed, or a platform event occurs. One process is always related to only one object, but it may include multiple criteria and either immediate or scheduled actions. While Process Builder is for seamless automation, Cloud Flow Designer is made to create guided visual experiences between the system and the user. With Cloud Flow Designer it is possible to create wizards, which collect information according to the user inputs. It is also possible to first create a more complex functionality with Cloud Flow Designer and then call the process created with Process Builder to handle the rest [43]. The last automation tool is called Approvals, which are for creating an approval process for some record. One use case for Approvals would

be, for example, a holiday request, which is first filled up by the employee and then approved or rejected by the supervisor. There could be more steps than just one, in case there is a need for multiple approvers.

Table 2. Example scenarios for each automation tool. [44]

Scenario	**Tool**
Guide a community member through requesting a new credit card with a step-by-step wizard.	Cloud Flow Designer
A sales rep clicks a button on an opportunity, which launches a discount calculator.	Cloud Flow Designer
When an account is updated, update all the contacts related to that account.	Process Builder
When an opportunity stage is updated, update a custom checkbox field.	Process Builder
Create a task when a platform event occurs.	Process Builder
Update a lead record in Salesforce after a certain amount of time passes, or when a specified time is reached.	Process Builder
When an opportunity closes, automatically create a renewal opportunity.	Process Builder and Cloud Flow Designer
Route an employee's time-off request to a manager for approval.	Approvals

Automation tools have certain limits, which are made to ensure that all the resources of the multi-tenant environment are not hogged by one complex automation process. For example, one organization may have 500 flows and processes activated at the same time. These limits can be bypassed to a certain level by using traditional programming languages. Therefore, automation tools are closely related to Apex, which is the Java-like programming language used by the Salesforce platform. For example, a process created with Process Builder can be replaced with Apex trigger [45]. However, there are Apex Governors and Limits as well, such as the number of allowed database queries per-transaction is 100 in total [46].

Third tool category of Salesforce are builders, which includes Schema Builder [47] and Lightning App Builder [48]. Schema Builder is a drag-n-drop tool, which can be used by the developer to modify the data model of the organization, either by creating new objects

or new fields to the schema. Lightning App Builder is also an easy-to-use drag-n-drop tool, but with a bit more complex use case. With Lightning App Builder, the developer can create applications from either standard, custom, or third-party components, which are usually made by hand-coding.

The latest innovation of Salesforce has been AI for CRM, called Salesforce Einstein [49]. Salesforce Einstein can provide typical AI benefits, such as personalized recommendations, intelligent search results, and automated assistants. While it might be hard to justify AI to be a low-code tool, it is more of a demonstration what are the LCDPs' next potential steps to keep evolving even more and to be more than just LCDPs.

3 RESEARCH METHODOLOGY

Biit Oy is a privately-owned Salesforce-consulting company from Espoo, Finland. Biit was founded in 2007 to help other organizations and companies to become smarter and more customer-oriented by offering services, solutions, and consultation related to the Salesforce platform. The most often these solutions are custom implementations using either low-code tools or more traditional programming techniques. Biit is also the oldest Finnish Salesforce-partner with 40 employees. For this study, Biit's employees were divided into 3 role-based groups: consultants, developers, and architects. Consultants are usually responsible to create low-code solutions using Salesforce's point and click tools, while developers create the hand-coded solutions using proper programming languages, such as Apex or JavaScript. Architects take more responsibility on the designing of the whole system, but they are usually able to create either just low-code solutions or sometimes both.

The qualitative data was collected via interviews, which were held at Biit's office in Espoo, Finland, in June 2018. The goal of the interviews was to get Biit's employees' opinions on the low-code in general; get the confirmation for low-code's benefits and pitfalls, which were proposed by the literature; understand how they think that their customers feel about the low-code; and which one do they prefer, low-code or hand-coded solutions. This data gathered to make conclusions and to have answers to the research questions. To achieve this, an a-three-part questionnaire was built based on the 3 research questions of this thesis. The questionnaire included questions about 1) employees experiences with the low-code, benefits and pitfalls of it and their opinion about the future of low-code, 2) their customers' attitude and expectations on low-code solutions, possible price difference and the maintenance differences between low-code and hand-coded solutions, and 3) which is their usual recommendation and if there is a difference in it based on the employee's role.

The interviews were constructed as semi-structured interviews with a list of questions, and the whole sessions were tape-recorded for qualitative analysis. 11 interview sessions were held in total. 10 of these were face- to-face with one employee from Biit and one

researcher. The remaining one was held via Google Hangouts video call. 9 of the interviews were held in Finnish and then translated into English, and the remaining two were held in English. To understand if there was a difference in the views of the low-code between Biit's employee roles, 4 consultants, 4 developers, and 3 architects were chosen to be interviewed. The decision of who was chosen to be interviewed was made randomly by the researcher, depending on who was available at the office during the time when interviews were held. The participants were from junior to senior level with the different amount of experience in Salesforce and software development in general, ranging from a minimum of 1 year to over 15 years of experience. One interview lasted from around 10 to 20 minutes. A total of 138 minutes of interview data were collected. During the interviews a certain flow was tried to achieve, with varying success, to keep the interviews semi-structured. The achieved flow was only interrupted to keep the discussion on the topic. No personal data or any unrelated background info were collected, other than the employee's role in the company. To identify the repeated themes, the collected interview data was classified following the principles of the open coding method from Grounded Theory [50].

4 RESULTS

The results of this study are qualitative data, which were collected by interviewing 11 Biit's employees with different background and experience in low-code and software development in general. These employees were divided into 3 groups: consultants, developers, and architects. The goals were to identify use cases where low-code development or hand-coding is better than the other; to identify the differences in customers' expectations between low-code and hand-coded solutions; and to identify if there is a difference in the preference of software development methods depending on the role of the employee. From the interviews, a total of four conjectures were found through the qualitative theme identification described in Chapter 3.

4.1 Low-code's main advantage is the simple process automation

The most of Biit's employees don't have a software engineering background or even any programming experience. The need of IT experts seems to be continuous and like the theory suggests, with the low-code development platform, such as Salesforce, a skillful and motivated person may work as a developer without any programming skills. However, it certainly helps if you know how to either code or even develop software.

I don't have a strong programming background. It has been a good step for me because I've been able to produce quick results with low-code without familiarizing myself with the structure of the application. – Architect 1.

If we're thinking about my background, I had neither configured nor programmed before I came to Biit. After a one-and-a-half-week course, I have been able to do productive work for the customer. – Consultant 3.

It makes the low-code easier if you know something about programming. You're still able to try and create things with just a small amount of knowledge. It's all visual, which makes it easier. – Architect 3.

The single most mentioned advantage of Salesforce's low-code tools was the simple process automation. The developer can automate some small business processes quickly using just point and click tools which Salesforce offers. One example, which concerns automated working time records using just Cloud Flow Designer, is given by Architect 3.

A while ago I made a solution for automated working time records using Flow. Every time a user made a working time record, it copied the information to the report objective. If there were no existing record for that date, a new record was made. If there was such a record, it was updated instead of creating a new one. – Architect 3.

The second most mentioned advantage was the speed of development when using low-code tools. It is possible to test some ideas quickly, then validate them with the customer and gain feedback to iteratively and incrementally develop the applications further. A few even stated that the low-code might be even faster than hand-coding.

You can create something visible quickly and then you may validate it with the customer. – Architect 1.

If there are no special requirements, the system can be wired up quickly. – Consultant 2.

You're able to create the same things with the traditional software development techniques, but it may be possible to do the same things faster with the low-code. – Architect 2.

One unexpected advantage of the low-code development was that there is no need for unit tests. In Salesforce, every hand-coded Apex class and trigger needs a unit test, which takes its own time during the development. Low-code solutions can be just deployed to the production straight after the development and testing. It is even possible to do the development in the production environment, which is, of course, not recommended.

It certainly makes life easier. We don't need to do unit tests from them. – Developer 1.

It is faster to do low-code when we're creating some simple automation because there's no need for unit tests. Also, the code needs to be developed in the development sandbox, low-code can be developed straight in the production org. – Architect 2.

Many thanks were given to the logic of low-code in general. Visual tools give the user a better understanding of what is happening when the application executes. This helps the non-technical people, who could be either the people from the business or even the customer.

It is possible to write things on the board in the way you think the process would work and then just click the same kind of icons inside the tool. Very easy-to-understand logic. – Consultant 3.

Low-code is easier to translate to either Finnish or English, and therefore it is easier to understand. – Developer 1.

One more advantage worth mentioning is the maintenance aspect of the development. The maintenance of the application is seen easier than before because it could be handled by the non-technical people as well.

Lots of maintenance responsibilities have moved from the IT to business people because it is just so easy that there is no need for the developer to be available. – Consultant 1.

If the customer wants to hire experts after the delivery, it is much easier to find someone who is able to maintain low-code applications than hand-coded ones. – Architect 1.

It also makes the maintenance easier, because the customer may do changes to the configuration without any programming skills. – Architect 2.

When talking about maintenance, few interviewees stated that Salesforce promises that low-code solutions will work regardless of the automatic updates of the whole system. If

there's a lot of custom code used in the environment, it could break down more easily after the system updates. Small fixes are easier to do through the Salesforce's Setup menu instead of the code.

I have more confident feeling during the scheduled updates when the system has just low-code solutions. It is also easier to just check from the low-code if there's something new or something which requires any attention, instead of reading through the custom code. – Consultant 1.

If you have a product catalog and you need to update the prices, there's no need to open the code when you can just use the UI and click the correct prices there. When we teach the customer how to use and manage the system, it will be cheaper for them, because they don't need our expertise to handle such simple tasks. – Consultant 4.

Table 3. A summary of the results on the first part of the RQ1.

Backgrounds of the interviewees.	• Most architects and consultants don't have programming skills. • Low-code is easier if you have some previous experience. • Basics of low-code development are easy to learn.
The most mentioned advantages of the low-code development.	• Simple process automation, such as automated working time records. • The speed of development when compared to the standard software development.
More minor benefits of the low-code development.	• No need for unit tests. • Development can be done in the production org. • Easy-to-understand logic of visual tools. • The easier maintenance of simpler solutions. • The promised compatibility with the future Salesforce releases.

4.2 Low-code's main disadvantage is the performance

It seems that although the low-code tools are highly liked among Biit's employees, it is easier to find something that is wrong rather than how something helps with the job. According to the interviewees, easily the biggest issue of the low-code is the performance in general. Only the Workflows, which is a tool Salesforce is trying to deprecate, is seen as a bit more efficient than the rest. Unfortunately, both Flow and Process Builder can't handle larger data volumes.

If you want to go with the low-code, the efficient and complex solutions are missing from there. Workflows might be efficient, but both Flows and Process Builders can't be bulkified and they're just inefficient and slow. – Developer 2.

Workflows work when you have a lot of data, but either Flows or Process Builders doesn't. They're not able to scale them well and their performance isn't just good enough. There are no common structures, such as maps, included. There are only lists which have to be looped through, which is not efficient. If you want to create more complex solutions, for example with multiple classes or inheritance, it is not possible with the low-code. – Architect 2.

When the developer creates something too massive for the low-code tools to handle, he or she might counter unfortunate performance issues, which usually comes hand in hand with maintenance issues. This could be related to the previously mentioned matter that the user may be able to work as a developer using low-code tools without the general understanding of the software engineering.

Low-code is so easy. You just put something here and there and suddenly you have one big mess in your hands, which is hard to control and even harder maintain. – Architect 1.

When developers are writing the code, they have to think how their code works with the system's existing code. With low-code, this doesn't seem to happen at all, or it might be just impossible to take everything into account. "Someone" just does "something" which

"just works", but it could break something else in the system. It should be possible to design the system in a way there wouldn't be any conflicts. – Architect 2.

The risk is that it is too easy to do things. When multiple consultants are creating "something", the system becomes too complex to handle. – Consultant 2.

The second most mentioned issue was the obscurity of the offered tools. It isn't always clear which tool should be used, as the tools have some overlapping with each other. Salesforce has some documentation [44] and examples about the issue, but it doesn't seem to be enough.

There's some overlap. In general, I think that the tools aren't mature enough yet. – Developer 1.

It feels dumb when Workflows, Flows, and Process Builders have some overlapping issues. It is not clear when to use what. The portfolio of tools should be more distinct. – Architect 3.

Other but less mentioned shortages were the facts that the version control is basically nonexistent and integrations with other systems are hard or even impossible to create with just low-code tools. Also, putting comments inside the low-code developed applications is impossible in Salesforce, which might make the maintenance even harder.

The version control is poor. You have to replace the old version with the new one, and that's it. Also, integrations are hard to implement using just low-code tools. Actually, the maintenance could be harder, because version controlling and commenting is harder. Low-code applications need a very good documentation. – Architect 1.

Integrations are hard or impossible to do, there's almost always some API calls, which have to be handled with code. – Architect 3.

Especially consultants accepted and even liked the fact that it is not possible to do

everything with low-code, but you can be sure that the supported methods really work.

At the same time, you don't know what is happening under the hood, but you can be sure that is executed every time the same way. It is both a good and a bad thing. – Consultant 3.

With low-code, you're able to do just the things which are supported by the tools. With hand-coding, you're free to do almost anything. – Consultant 4.

When low-code is not enough, there's always the possibility to use code. Using both is possible, too, but it could make things more complex. Usually, if low-code is not enough, it is just better to move everything to code.

It the solution is too complex, it is better to re-create it by hand-coding. It is possible to do working solutions using both low-code and hand-coding, but it is more likely to be ugly, unusable, and hard to maintain. – Consultant 4.

Things you're able to do with low-code work pretty well, but after a while, you're against the wall and you have to start coding. – Consultant 1.

When you mix process automation to many different layers, in a way that there are low-code and hand-coded solutions messing up together, it is hard or impossible to debug and find the defects in the system. – Developer 2.

Unfortunately, it seems to be the general procedure is to start with the low-code, build up the system little by little, and then realize that either the developer or the system can't handle the complexity of the system anymore. It is rather common that the low-code solutions are ditched and converted to code, mainly because of the performance issues. There are some rare situations where code have been converted to low-code as well, mainly because there have been some requirements from the customer, such as the need for the simpler system. Of course, if the code works and the conversion doesn't give the customer anything new, it is rare that the customer wants to pay for such conversion, although it could make things easier in the long run.

One customer had a lot of workflows, process builders as well as code. There was a lot of traffic in one transaction, so we hit Salesforce's governor limits. All the low-code were converted to code in order to make the system efficient enough. – Developer 2.

It's always the performance. Firstly, we start with something small, which works just like it should. Then the second, then the third. After a while, it doesn't work anymore and we need to convert something or everything to code because of the performance. It doesn't really happen the other way around. Sometimes we are asked to make the system simpler, but that often means that we have to get rid of some automation. It might make the maintenance easier, but it costs money when such conversion is made, so I don't know which will be cheaper in the long run. Usually, customers don't want to pay for such a thing. – Architect 2.

In my opinion, it is unacceptable that low-code solutions aren't as efficient as hand-coded solutions. It should be as good or even better. When thinking about the architecture in general, the best practice is to have all the automation of one process in the same place. I think that if there's anything that requires a code, then everything should be there. There's no idea to mix things up just for the sake of the possibility. – Consultant 3.

Many agreed that in the right hands the hand-coded solution is both more efficient and faster to create. The biggest problem might be that there are so few skilled developers available. Although low-code isn't the backup option for coding, things might be different if there were more developers available and looking for a job.

It is possible that some professional developer is able to create code faster than with low-code, and the code is also more efficient. – Architect 3.

Low-code tools are never as efficient as the code of your own. Of course, there are bad developers out there, too. – Developer 4.

Table 4. A summary of the results on the second part of the RQ1.

The most mentioned disadvantages of the low-code development.	• The inefficiency of the solutions. • Major performance issues with larger data volumes. ○ The lack of ability to bulkify database calls. • Maintenance issues with more complex solutions. • The obscurity of the offered tools.
More minor pitfalls of the low-code development.	• The lack of common structures, such as maps. • The lack of ability to create more complex solutions, such as using multiple classes or inheritance. • The low-code developers may not take other developers into consideration, because they are not familiar with traditional software engineering methods. ○ Not enough skilled developers available. • The lack of version control. • No possibility to create comments inside the solution.
Possible workarounds.	• Solutions can and sometimes must be converted to code.

4.3 If the solution works, the customer doesn't really care about the used methods

Consultants solve problems which the customer can't or doesn't want to solve. During the interviews, many of the Biit's employees emphasized the fact that customers trust that the consultants are experts in their field and can do the right choices to create the right solutions for their needs.

Basically, everyone's waiting and assuming that the consultant has a vision. What is the right solution and why? – Architect 2.

Our customers trust us and they have given us the responsibility to come up with the right solution. We do what is best for them, and if it is possible to do in low-code, we do it with it. – Consultant 4.

Especially the new customers are not aware of the possibilities Salesforce's low-code tools may be able to offer them. When the time goes by, project after project, they become familiar with the system, start to understand more and become rather curious what could be done or achieved with the same license price.

New customers are always surprised how quickly we're able to do changes without code. They don't know much, but the surprise is always huge. – Architect 2.

New customers don't know about the possibilities. Old ones do know that we can build automation, but I doubt that they're interested in the used methods. They understand the possibilities, but they don't know about the different tools. – Architect 3.

There might be someone who is more technically oriented, but usually, they don't know. They just think that it is configuring the system, what it basically is. They're always surprised how fast and easy it is. They're certainly expecting code. – Consultant 4.

It was said that many customers are stuck in the past. They don't understand the terms and they always assume that all the new methods are always cheaper, faster and an easier way to complete the project. Low-code is yet another IT term, just like agile development have been for a while.

They mix things up a lot. We're like "hey, we have these low-code tools which are fast to use and give you instantly something visual. Here's a new field, now we changed its place, now there's many of them, now they're sorted in a different way." They think that this means the projects will be a lot shorter. In the reality, the development won't even take the largest part of the time. They don't really know about cloud-based platforms, either. Like, you're able to do something more with it than just CRM. They still think that they would need to have one system for one purpose, one provider for one system, and they would need one expert with at least 10 years of experience in order to accomplish anything. – Consultant 3.

Sometimes customers want things to be done in a certain way. Usually, this is because of

either the easier maintenance possibilities or previous performance issues with the system. No one has ever demanded to not to use low-code, though.

During my last project, there was a requirement that everything must be done with low-code. There are few customers who understand the ease of maintenance of low-code solutions. – Architect 1.

I have never encountered anyone who doesn't want low-code solutions. Usually, solutions using standard features are very popular and wanted. – Consultant 1.

Of course, there is always the time when the customer gets interested in the used methods: when something doesn't work.

It is pretty rare that we get any specific requirements from our customers on how we should accomplish things. Maybe from the older partners, who know how they've done things previously. I doubt that they're interested at all. Of course, if something doesn't work, for example, if there are some major performance issues which can be seen by the end-user, they suddenly become interested and hand out demands. – Architect 3.

It seems that there is no difference between different sized companies. Everyone prefers the low-code, although the reason could be different. With smaller companies, it is usually important to keep the expenses small, while big ones prefer the low-code mainly because of the easier maintainability.

When we're thinking about bigger and smaller companies, smaller ones want low-code solutions because they want to minimize their expenses. They think that they get enough flexibility with the small budget when they stick close to the out of the box features. Bigger ones might already have more experience on software projects and they might have realized that their system is too large and complex for anyone, even for the supplier, to handle. It is normal to hear these words during the first meeting: "there will be no code, right?" The code has almost become a bad thing they just want to avoid. – Consultant 3.

The price of the project or solution depends on the time used during it, which could mean that the low-code could be a bit cheaper, depending on the task. If the data volumes are small and there is not much automation required, the project will be rather inexpensive from the delivery to maintenance. Things will become more expensive if there are constant additions to the system and the data volume increases when it might have been better to choose the hand-coded solution in the first place. Of course, then the maintenance could be harder.

Usually, when we're making an offer for the customer, we don't define how the solution will be done. We will check the requirements and then see what is the best choice. If the problem is complex, something that requires coding, it naturally increases the price. We don't really use low-code as a weapon in the price war. – Architect 1.

There is a price difference right in the starting price. Some small thing could be cheap to develop with just low-code tools. When the complexity grows, it won't be so clear, which one is the cheaper option. The price difference comes from the time and maintainability. It is also easier and perhaps cheaper to find a good administrator rather than a good developer. – Developer 2.

Small changes are fast to make with the low-code tools. Some simple automation might be the same priced but much more robust when done with the traditional code. Customers have to think about the life-cycle cost. It might be better to use the code instead of the low-code if the low-code doesn't fit right into the use case. Life-cycle cost is the most important aspect. Unfortunately, it is always easier to find someone who is able to make these low-code solutions. You need a different mindset to be a developer. – Consultant 3.

4.4 Low-code is always the primary option for everyone

The interviewees hadn't encountered much of the Salesforces advertisements, but they had a hunch that Salesforce promotes the use of their low-code tools. The most common point in the advertisements is the speed of development, which is promised to be from 50% to 70% faster [51] [52] than with the traditional coding. The most common place where Biit's

employees have encountered such advertisements have been in Salesforce seminars around the world.

Yes, they do mention low-code in their advertisement materials. They say that it is 70% faster or something to develop applications with low-code than with hand-coding. I've not seen that much of it, mainly in Dreamforce, which is this huge event held by Salesforce. – Developer 1.

I think that they urge and encourage us to use low-code. In Trailheads, there are a lot of modules teaching us about low-code. They always say that we should at least try to use low-code first. It's the same thing in Salesforce's seminars, where they have this term "Awesome Admin", which means an administrator, who is able to do things by configuring and without code. Lots of success stories, etc. – Architect 2.

Biit's employees see low-code tools as a selling point. It works as a selling point for both Salesforce and Biit. The main reasons are the speed of development, easier maintenance, customizations, and lower life-cycle cost.

I see the low-code as a selling point. And we hope that the customer sees it as the same way. It is faster for them, they're able to maintain it by themselves without any outsourcing. – Architect 2.

Absolutely. It has surprised me that many larger companies have chosen to outsource everything. They feel that they don't want to keep such know-how inside their company. With Salesforce, they can just throw one guy from their IT to the one-week course and after that, they have a sufficient resource to make changes to the system. You just have to think about the life-cycle cost. – Consultant 3.

Everyone, which includes consultants, developers, architects, and customers, prefers low-code over hand-coding. Developers, of course, are willing to do some coding, which is their job, but if the solution doesn't require it, they don't try to push it forcibly.

I think that the recommendation is to use low-code before anything. There are arguments for and against the speed of development, but at least it is easier to read. – Developer 1.

Low-code all the way. If it solves the customer's problem, it will be cheaper, faster and iterations will be easier. And, of course, it is always possible to switch from low-code to hand-coded solution, if low-code is not enough. – Architect 1.

The listed reasons were the same as before: the speed of development, affordability, maintainability, support by Salesforce and easier to find resources for maintenance. One even wanted to point out that customers are pleased when they can utilize the whole system and therefore they get more with the same license price.

Primarily I prefer low-code. I trust that the system is more reliable and will always work after the system updates. – Consultant 1.

Salesforce is constantly developing their tools and by using them customers get the most out of the license prices they pay. Less is always better when we are talking about customers' maintainable systems. – Consultant 3.

All the interviewees were sure that the low-code is here to stay. It has found its place as a part of software development and it will ease the recruiting pressure on companies, as skillful developers seem to be rare.

Lots of config responsibilities have move from developers to business persons, which is the reason that the value of low-code will grow even more. – Consultant 1.

Companies need more experts all the time. I don't know how to code, I wouldn't have got this far without low-code. – Consultant 3.

It is certainly not a fad. It is the future. – Consultant 2.

On the other hand, they also believe that low-code won't take anything off from the

traditional software development. Hand-coding will always be part of the software development, even in Salesforce. It will always be the backbone behind the limitations of the low-code tools.

Both coding and low-code continue to grow. You need more code to create more complex things, but when the low-code tools develop, you're able to do more with them, too. – Consultant 4.

At least Salesforce won't go towards one direction. They have provided new techniques to both, code and low-code. – Developer 4.

5 DISCUSSION AND CONCLUSIONS

In this study, three separate research questions were defined. These were *What are the current benefits and pitfalls of low-code development compared to hand-coded solutions?*, *What are the customers' expectations of low-code development solutions?*, and *Which, in the case company's view, is the better software development method?*. By answering these questions, the aim was to get more in-depth data on how low-code works together with standard software development, but also to understand why such a method has developed, is there a need for it and do customers see it as a possibility to achieve something more.

RQ1, pros of the low-code development: the interviews confirmed all the benefits of the LCDPs listed in the literature and in previous reports. One of the main benefits is the speed of development. By using the point and click tools provided by LCDPs, the developer can create something working and visual much faster than with traditional programming. Related research [51] [52] as well as one of the Biit's employees estimated that the application development life-cycle may be from 50% to 70% faster than with standard software development. It also helps non-technical people to participate in the project at a much earlier stage and therefore they can provide feedback back to the developer. Although the term "citizen developer" is criticized by some, alternative ways of developing applications, such as using LCDPs, may provide help to the unfortunate situation with the lack of skilled developers. Then the more skilled developers may focus on more critical areas of software development, while low-code developers are creating applications and perhaps bringing new perspectives to the field.

Although the maintenance was mentioned mainly as a benefit by both the literature and the interviewees, there were also some arguments against it. If the system's customizations are well designed and structured, the maintenance of low-code solutions is certainly easier, as it could be done by almost anyone, even by the customer. However, there is a risk that the system will grow to be too big to handle and maintain. The same risk exists when creating software with standard code, but usually, the one who is responsible for the maintenance knows the system much better than just someone from the customer.

The interviewees pointed out a couple of benefits which weren't listed in the literature, perhaps mainly because they're more of Salesforce-specific features. The first one was the lack of unit tests, which are required to be done for every Apex class and trigger in Salesforce. It makes it possible to develop some smaller applications straight to the production because there is no need to follow the systems own testing requirements. Although this is not the preferred way to do things, it works well when there are some obvious and small bugs in the applications, such as spelling mistakes. These bugs could be fixed without setting up sandboxes and playing with Salesforce's change sets. The second, and the most mentioned advantage and use case of the low-code, was the process automation. With low-code, the developers can create small automation features to remove some of the steps and therefore provide ease to the workflow. These process automation features could be an automatically created record, an updated field or even a confirmation email, which is sent to the customer after the project's stage has been set to be accomplished.

RQ1, cons of the low-code development: the literature didn't list so many disadvantages as advantages and they came mainly from smaller individuals. There were doubts that low-code is just another fad like 4GL languages and Visual Basic programming was before it. Each of the interviewed employees shot that argument down stating that low-code is certainly something, which has found its place as a part of the portfolio of software development methods and will develop even more in the following years. They admitted and understood that one of the weaknesses of low-code is the fact that the most often the developers creating low-code applications are not familiar with the principles and techniques of software engineering, which may lead to badly designed and structured systems, which are hard or even impossible to handle, control and maintain. This is the result when multiple low-code developers are doing their own thing on the same environment without considering the whole system.

The low-code tools have quite a few limitations, which force the developer to open the code editor and start programming. According to the interviewees, these limitations are unfortunate, but they are well known and they can live with them. They understand that right now low-code tools might be better when creating smaller automation or PoCs (proof

of concepts), but everything more complex should be left to the actual software engineers. Biit's employees could list multiple customer cases where the low-code wasn't enough and everything must have been hand-coded from the scratch.

According to Biit's employees, the worst disadvantage is the performance issues. The low-code applications' performance is nowhere near as good and efficient as hand-coded applications', which was described to be "unacceptable" by one of the consultants. Because of the possible performance issues with larger data volumes, developers have to consider carefully is the low-code even a suitable option for the customer's problem. In Salesforce, there are also some overlapping between the tools, so it is not always clear which tool should be used in some certain situation. There are high hopes that Salesforce cleans the selection of the tools, but it might take a while. The clearest missing features are the native version control of the system as well as the commenting of the low-code solutions inside the point and click tools. Integrations were also mentioned. While they are certainly hard to and sometimes impossible to create, according to one of the consultants, there are some possibilities to develop those only with the low-code tools.

RQ2, customers' expectations: consultant companies, like any other companies, are highly dependent on their customer satisfaction. According to the interviews, Biit has gained their customers' trust by working hard and creating suitable solutions. Customers trust that consultants can offer, develop and maintain the best possible solutions to their problems. It is rather surprising how little customers seem to know about the possibilities Salesforce can offer. Some of the interviewees were sure that especially the new customers don't know about the opportunities. Perhaps they just think that Salesforce is just a CRM and nothing else. One consultant pointed out that it is common for the customers to think that they need one system for one purpose and one provider for one system. The whole concept of the PaaS seems to be unknown. Interviewees were guessing that perhaps the low-code sounds like yet another fad, just like the agile development. They think that everything is cheaper, faster and easier, which isn't, of course, always the case. Customers don't really care how their solutions are made if they work and there are no hiccups visible to the end user. Of course, if something doesn't work as swiftly as it should, they suddenly become interested and hand out demands how things should be done.

If the preference is asked after explaining the terms, low-code is always the preferred option. Some customers might even demand to use only low-code solutions, usually because of the easier maintenance possibilities. In the best-case scenario, the customer can maintain the system by itself, which gives them savings in the long run. It was also said that standard features are popular because they feel like a safer option; the solutions are promised to be supported by Salesforce after the system updates as well. For smaller companies, which could have tighter budgets, standard and low-code features are often cheaper than highly customized solutions made by programming. The price at Biit depends on the used time, so depending on the complexity and handed demands, the price could go either way. The most important factor for the customer to consider is the life-cycle cost of the system.

RQ3, the preferred method: Salesforce promotes low-code and its benefits quite a lot in their marketing materials, conferences, and training courses. They often mention a concept of "Awesome Admin", which means an administrator who can maintain the Salesforce system without programming skills. The low-code is seen as a selling point by both Salesforce's and Biit's point of view. Salesforce can promote their platform and engage companies in their ecosystem, while Biit is able to offer something unique and new in the world of software development and IT consulting. Although low-code solutions have been criticized by experienced developers and influential persons of software engineering, it was surprising that every single interviewee convinced that low-code is always the primary method to develop applications for customers. Even the developers, who are mainly programming, said that low-code is the main way of doing things in Salesforce and programming is the backup option. Reasons are the same advantages which are listed many times before: possibility to have a cheaper and faster solution with easier maintainability. Although everyone was sure that the low-code is here to stay, they didn't see it as a replacement for standard software development. It is a new opportunity to create software and they have different focuses with hand-coded solutions; the first is made for smaller and lighter solutions, while the latter doesn't really have limits at all.

Overall, the results of the interviews confirmed the theory from the previous reports on this

subject. The main advantages of low-code are the speed of development and easier maintainability when the designing is done right. The main use case for the low-code in Salesforce platform is the simple process automation. The main disadvantage of low-code is the performance. The team may be able to avoid performance issues by designing the system well from the start, which requires that the team is familiar with the best practices of software engineering. People without this knowledge can use the low-code tools and provide great results with it, but in the more complex systems, the performance might become an issue. Customers assume that the low-code is cheaper, faster and easier for everyone, but usually, they don't really care which method is used if everything works correctly. The main reasons to use low-code from customers' point of view are the affordability in a long run and the maintainability. The preferred method among the Biit's employees is also the low-code. The main principle is that the low-code tools are the first option, but if there are some limitations which makes them too stiff to use, everything may be converted to traditional code.

For the future work, it would be important to include more companies which use different LCDPs. Some of the benefits and pitfalls introduced in this study may be Salesforce related and the results could differ when comparing other platforms. It would also be great to have some data how quickly professionals are able to develop applications with both methods and then compare the speed of development, the maintainability as well as the general feelings of the said developers during and after the process. Another important step would be to include some customers in the study as well. All the results about the customers are according to Biit's employees' experiences only. A more specific data would be gained if the customers were also interviewed.

6 REFERENCES

1. Financial Times Global 500 2015. *Financial Times*, [online] Available at: https://www.ft.com/ft500 [Accessed 5 May 2018].
2. The World's Billionaires. *Forbes*, [online] Available at: https://www.forbes.com/billionaires/list/ [Accessed 5 May 2018].
3. O'Regan, G. (2012). *A Brief History of Computing*. 2nd ed. London: Springer London Ltd, pp.60-61, pp.145-170.
4. Sommerville, I. (2011). *Software engineering*. 9th ed. Boston: Pearson, pp.5-82.
5. Weinmeister, P. (2015). *Practical Salesforce.com Development Without Code: Customizing Salesforce on the Force.com Platform*. Berkeley: Apress, pp.xxiii.
6. Rymer, J. (2017) *The Forrester Wave™: Low-Code Development Platforms For AD&D Pros, Q4 2017*. [online] Cambridge: Forrester. Available at: https://hosteddocs.emediausa.com/new-forrester-lowcode-development-wave-report-2017.pdf [Accessed 5 May 2018].
7. Salesforce.com. *Apex Developer Guide – Execution Governors and Limits. Salesforce*. [online] Available at: https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_gov_limits.htm [Accessed 5 May 2018].
8. Salesforce.com. Trailhead – *Get Started with the Salesforce Platform*. [online] Available at: https://trailhead.salesforce.com/en/modules/starting_force_com/units/starting_intro [Accessed 5 May 2018].
9. *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Standard 610.12, 1990.
10. Mens, T., Demeyer, S. (2008). *Software Evolution*. Berlin: Springer-Verlag, pp.1-11.
11. Glass, R. (2001). Frequently forgotten fundamental facts about software engineering. *IEEE Software*, (18), pp.110-112.
12. Manifesto for Agile Software Development, (2001). [online] Available at: http://agilemanifesto.org/ [Accessed 01 Jun 2018].
13. Martin, R. (2003). *Agile Software Development*. Upper Saddle River, N.J.: Pearson Education, Inc., pp.3-9.

14. Martin, R. (2009). *Clean Code*. Upper Saddle River, N.J.: Prentice Hall, pp.1-16.
15. Revell, M. (2017). *What is Low-Code?* [Blog] Outsystems Blog. Available at: https://www.outsystems.com/blog/what-is-low-code.html [Accessed 15 May 2018].
16. Martin, J. (1982). *Application Development Without Programmers.* New Jersey: Prentice-Hall, Inc., pp.14-28.
17. Rossi, B. (2015). *On the down low: Why CIOs should care about Low Code*. [online] Information Age. Available at: http://www.information-age.com/down-low-why-cios-should-care-about-low-code-123459895/ [Accessed 09 Jun 2018].
18. Richardson, C., Rymer, J. (2014). *New Development Platforms Emerge For Customer-Facing Applications*. [online] Cambridge: Forrester. Available at: http://content.k2.com/Global/FileLib/Analyst_Reports/Forrester_New_Development_Platforms.pdf [Accessed 16 May 2018].
19. Richardson, C., Rymer, J. (2016). *Vendor Landscape: The Fractured, Fertile Terrain Of Low-Code Application Platforms.* [online] Cambridge: Forrester. Available at: https://informationsecurity.report/Resources/Whitepapers/0eb07c59-b01c-4399-9022-dfc297487060_Forrester%20Vendor%20Landscape%20The%20Fractured,%20Fertile%20Terrain.pdf [Accessed 22 May 2018].
20. Rymer, J. (2017). *Vendor Landscape: A Fork In The Road For Low-Code Development Platforms*. [online] Cambridge: Forrester. Available at: http://www.flowforma.com/forrester-vendor-landscape-report-august-2017 [Accessed 22 May 2018].
21. Muukkonen, H. (2017). *Koodaripula iski Suomeen – palkat jopa 15 000 euroa kuussa*. [online] Talouselämä. Available at: https://www.talouselama.fi/uutiset/koodaripula-iski-suomeen-palkat-jopa-15-000-euroa-kuussa/013efa5f-f25b-37c8-bed8-04054362a5f6 [Accessed 09 Jun 2018].
22. Daxx.com, (2017). *Talent Shortage in the Nordics: Finland, Sweden, and Denmark Are Coming Short of Software Developers*. [online] Available at: https://www.daxx.com/article/nordics-tech-talent-shortage [Accessed 09 Jun 2018].
23. Shiah, J. (2018). *Why "low-code development" isn't always truly "low-code"*. [online] ITProPortal. Available at: https://www.itproportal.com/features/why-low-code-development-isnt-always-truly-low-code/ [Accessed 09 Jun 2018].

24. Nepal, M. (2018). *Launchpads to Build Powerful Apps Easily*. [online] Kissflow.com. Available at: https://kissflow.com/low-code/low-code-development-platform-launchpad-build-apps-easily/ [Accessed 09 Jun 2018].
25. Bloomberg.com, (1998). *Steve Jobs: 'There's Sanity Returning'*. [online] Available at: https://www.bloomberg.com/news/articles/1998-05-25/steve-jobs-theres-sanity-returning [Accessed 02 Jun 2018]
26. Van Schetsen, A. (2016). *Guest View: Five reasons low-code development is no longer optional*. [online] SD Times. Available at: https://sdtimes.com/agile/guest-view-five-reasons-low-code-development-no-longer-optional/ [Accessed 09 Jun 2018].
27. De, S., Sam, J. (2015). *Low-code development meets fine-grained security*. [online] InfoWorld. Available at: https://www.infoworld.com/article/3000572/application-development/low-code-development-meets-fine-grained-security.html [Accessed 09 Jun 2018].
28. Viswanathan, K. (2017). *No-code vs. Low-code – Is there a difference?* [Blog] WaveMaker.com. Available at: https://www.wavemaker.com/no-code-vs-low-code-different/ [Accessed 09 Jun 2018].
29. PMG, (2017). *IT and the Rise of the RAD Enterprise*. [online] Available at: http://www2.pmg.net/RADsurvey [Accessed 09 Jun 2018].
30. Reselman, B. (2018). *Why the promise of low-code software platforms is deceiving*. [online] DevOpsAgenda.com. Available at: https://devopsagenda.techtarget.com/opinion/Why-the-promise-of-low-code-software-platforms-is-deceiving [Accessed 09 Jun 2018].
31. Idesis, S. (2017) *Why Developers Fear Low-Code*. [online] DZone.com. Available at: https://dzone.com/articles/why-developers-fear-low-code [Accessed 09 Jun 2018].
32. Nyssen, A. (2017). *Why low code can only be the beginning*. [Blog] itemis.com. Available at: https://blogs.itemis.com/en/why-low-code-can-only-be-the-beginning [Accessed 09 Jun 2018].
33. Natis, Y. et al., (2015). *Magic Quadrant for Enterprise Application Platform as a Service, Worldwide.* [online] Stamford: Gartner, Inc. Available at: https://www.ibm.com/cloud-

computing/files/GARTNER_COMP_magic_quadrant_for_enterpris_271188_Bluemix.pdf [Accessed 09 Jun 2018].

34. Salesforce.com. Trailhead – *Understand Custom & Standard Objects*. [online] Available at: https://trailhead.salesforce.com/en/modules/starting_force_com/units/starting_intro [Accessed 11 Jun 2018].
35. Salesforce.com. Trailhead – *Create Object Relationships*. [online] Available at: https://trailhead.salesforce.com/trails/force_com_dev_beginner/modules/data_modeling/units/object_relationships [Accessed 11 Jun 2018].
36. Salesforce.com. Trailhead – *Overview of Data Security*. [online] Available at: https://trailhead.salesforce.com/trails/force_com_dev_beginner/modules/data_security/units/data_security_overview [Accessed 11 Jun 2018].
37. Salesforce.com. Trailhead – *Use Formula Fields*. [online] Available at: https://trailhead.salesforce.com/trails/force_com_dev_beginner/modules/point_click_business_logic/units/formula_fields [Accessed 11 Jun 2018].
38. Salesforce.com. Trailhead – *Implement Roll-Up Summary Fields*. [online] Available at: https://trailhead.salesforce.com/trails/force_com_dev_beginner/modules/point_click_business_logic/units/roll_up_summary_fields [Accessed 11 Jun 2018].
39. Salesforce.com. Trailhead – *Create Validation Rules*. [online] Available at: https://trailhead.salesforce.com/trails/force_com_dev_beginner/modules/point_click_business_logic/units/validation_rules [Accessed 11 Jun 2018].
40. Salesforce.com. Trailhead – *Automate Simple Business Processes with Process Builder*. [online] Available at: https://trailhead.salesforce.com/trails/force_com_dev_beginner/modules/business_process_automation/units/process_builder [Accessed 11 Jun 2018].
41. Salesforce.com. Trailhead – *Guide Users Through Your Business Processes with Cloud Flow Designer*. [online] Available at: https://trailhead.salesforce.com/trails/force_com_dev_beginner/modules/business_process_automation/units/flow [Accessed 11 Jun 2018].
42. Salesforce.com. Trailhead – *Customize How Records Get Approved with Approvals*. [online] Available at:

https://trailhead.salesforce.com/trails/force_com_dev_beginner/modules/business_process_automation/units/approvals [Accessed 11 Jun 2018].

43. Salesforce.com. Trailhead – *Combine the Power of Process Builder and Cloud Flow Designer*. [online] Available at: https://trailhead.salesforce.com/trails/force_com_dev_beginner/modules/business_process_automation/units/business_process_automation_combined [Accessed 11 Jun 2018].
44. Salesforce.com. Trailhead – *Choose the Right Automation Tool*. [online] Available at: https://trailhead.salesforce.com/trails/force_com_dev_beginner/modules/business_process_automation/units/process_whichtool [Accessed 11 Jun 2018].
45. Salesforce.com. Trailhead – *Get Started with Apex Triggers*. [online] Available at: https://trailhead.salesforce.com/trails/force_com_dev_beginner/modules/apex_triggers/units/apex_triggers_intro [Accessed 11 Jun 2018].
46. Salesforce.com. Developer Documentation – *Apex Governor Limits*. [online] Available at: https://developer.salesforce.com/docs/atlas.en-us.salesforce_app_limits_cheatsheet.meta/salesforce_app_limits_cheatsheet/salesforce_app_limits_platform_apexgov.htm [Accessed 11 Jun 2018].
47. Salesforce.com. Trailhead – *Work with Schema Builder*. [online] Available at: https://trailhead.salesforce.com/trails/force_com_dev_beginner/modules/data_modeling/units/schema_builder [Accessed 11 Jun 2018].
48. Salesforce.com. Trailhead – *Get Started with the Lightning App Builder*. [online] Available at: https://trailhead.salesforce.com/modules/lightning_app_builder/units/lightning_app_builder_intro [Accessed 11 Jun 2018].
49. Salesforce.com. Trailhead – *Lean Salesforce Einstein Basics*. [online] Available at: https://trailhead.salesforce.com/en/modules/get_smart_einstein_feat/units/get_smart_einstein_feat_basics [Accessed 11 Jun 2018].
50. Borgatti, S. *Introduction to Grounded Theory*. [online] Available at: http://www.analytictech.com/mb870/introtogt.htm [Accessed 14 Jun 2018].
51. Salesforce Research. (2017). *Top Trends In Low-Code Development*. [online] Salesforce Research. Available at: https://www.salesforce.com/form/platform/state-

of-low-code-report.jsp [Accessed 14 Jul 2018].

52. Carvalho, L., Marden, M., Arora, U. (2016). *The ROI of Building Apps on Salesforce*. [online] IDC. Available at: https://platform-roi-calculator.salesforce.com/IDC-ROI-of-Building-Apps-on-Salesforce.pdf [Accessed 14 Aug 2018].

APPENDIX 1. Interview questions in English

Research question		**Goal**
RQ1: What are the current benefits and pitfalls of low-code development compared to hand-coded solutions?	What kind of low-code solutions have you done?	Identify the use cases where low-code development or hand-coding is better than the other.
	Have low-code made your job easier? If so, how?	
	What is the greatest benefit low-code has offered to you?	
	Are there some features missing from Salesforce's low-code portfolio? If so, what kind of features?	
	What solutions cannot be done with low-code tools?	
	What is the biggest pitfall bothering low-code?	
	What do you think that will happen to low-code in the future?	
RQ2: What are the customers' expectations of low-code development solutions?	Do customers know about Salesforce's low-code tools?	Identify the differences in customers' expectations between low-code and hand-coded solutions.
	What kinds of customers want low-code solutions?	
	Is there a price difference between low-code and hand-coded solutions?	
	Do you see low-code tools as a selling point?	
	Have any customer specifically demanded for low-code, or against it?	
	Is the low-code solutions' maintenance after deployment different than hand-coded solutions'?	
RQ3: Which, in the case company's view, is the better software development method?	Which type of solutions do you recommend and offer first to the customer? Why?	Identify if there is a difference in preference of software development methods depending on the role of the employee (consultant/developer/architect).
	Have you ever been in a situation where low-code were converted to hand-coded solution? If so, why? How about vice versa?	
	Is Salesforce promoting the usage of low-code tools? If so, how?	
	Have you used any other low-code development platform?	

APPENDIX 2. Interview questions in Finnish

Research question		**Goal**
RQ1: Mitkä ovat low-code -ohjelmistokehityksen vahvuudet ja heikkoudet verrattuna perinteiseen ohjelmointiin?	Millaisia low-code -ratkaisuja olet tehnyt?	Tunnistetaan käyttökohteet, milloin low-code -ohjelmistikehitys tai perinteinen ohjelmointi on parempi vaihtoehto kuin toinen.
	Onko low-code helpottanut työtäsi? Jos on, niin miten?	
	Mikä on suurin hyöty, jonka low-code on sinulle tarjonnut?	
	Puuttuuko Salesforcen low-code -työkalujen valikoimasta jotain ominaisuuksia? Jos puuttuu, niin mitä?	
	Millaisia ratkaisuja ei voi toteuttaa low-code -työkaluilla?	
	Mikä on low-coden suurin heikkous?	
	Mitä luulet low-codelle tapahtuvan tulevaisuudessa?	
RQ2: Mitkä ovat asiakkaan odotukset low-code -ohjelmistokehityksen ratkaisuista?	Tietävätkö asiakkaat Salesforcen tarjoamista low-code -työkaluista?	Tunnista asiakkaan odotusten eroavaisuudet low-code- ja perinteisen ohjelmoinnin ratkaisujen välillä.
	Millaiset asiakkaat haluavat low-code ratkaisuja?	
	Onko low-code -ratkaisujen ja perinteisen ohjelmoinnin välillä hintaeroa?	
	Näetkö low-code -työkalut myyntivalttina?	
	Onko kukaan asiakas vaatinut käyttämään tai olla käyttämättä low-code -työkaluja?	
	Eroaako low-code -ratkaisujen ylläpito perinteistein ohjelmointiratkaisujen ylläpidosta?	
RQ3: Kumpi ohjelmistokehitysmenetelmä on case-yrityksen näkemyksen mukaan parempi?	Kumman tyyppisiä ratkaisuja suosittelet ja tarjoat ensin asiakkaalle? Miksi?	Tunnista mahdolliset erot eri roolien (konsultti/kehittäjä/arkkitehti) suosimissa ohjelmistokehitysmenetelmissä.
	Oletko koskaan ollut tilanteessa, jossa low-code -ratkaisu on ohjelmoitu uudelleen käsin? Jos on, niin miksi? Entä toisinpäin?	
	Mainostaako Salesforce low-code -työkalujen käyttöä? Jos mainostaa, niin miten?	
	Oletko käyttänyt mitään muuta low-code -ohjelmistokehitysalustaa?	