Developer Experience in Low-Code Versus Traditional Development Platforms - A Comparative Experiment

Arian Kermanchi

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 21.11.2022

Supervisor

Prof. Fabian Fagerholm

Advisor

MSc Aki Teronen

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Arian Kermanchi

Title Developer Experience in Low-Code Versus Traditional Development Platforms - A Comparative Experiment

Degree programme Computer, Communication, and Information Sciences

Major Software and Service Engineering **Code of major** SCI3043

Supervisor Prof. Fabian Fagerholm

Advisor MSc Aki Teronen

Date 21.11.2022 **Number of pages** 54+22 **Language** English

Abstract

Developer Experience stems from the concept of User Experience but focuses on software developers and their experience when using development platforms and tools to produce software. In recent years, as industries need more software developers with higher productivity to produce increasingly domain-specific and complex digital solutions, the concept of low-code software development has gained popularity. Low-Code Development Platforms (LCDPs) enable users with or without prior programming experience to create applications using model-driven or graphical user interfaces instead of writing textual code. Currently, there is limited research on the differences in developer experience between traditional and low-code development tools. This thesis explores the differences between developers' episodic experience when using traditional development platforms versus low-code development platforms by conducting a comparative experiment using Salesforce Flow Builder (a LCDP) and APEX programming language. Participants with varying levels of programming experience but no Salesforce experience were taught the basics of Salesforce Flow Builder and then tasked to accomplish a number of tasks using the tool. Then they were taught the basics of APEX language and tasked to accomplish the same set of tasks using the language. Their experiences and thoughts were captured using questionnaires and interviews and their performance was graded by running test cases on their delivered solutions. The results indicate that past programming experience appears to strongly influence the developers' performance, experiences, and tool preferences, while most developers remain skeptical of the scalability and maintainability of applications developed using LCDPs. There are also polarized opinions on the efficiency of the tools. In addition to discussing the experiment results and findings, this thesis also provides insights for future research in this area.

Keywords developer experience , low-code development platform , software development , enterprise applications , Salesforce , declarative development , programmatic development

Contents

Abbreviations and Acronyms

CRM	Customer/Constituent Relationship Management
DX	Developer Experience
EUD	End-User Development
GUI	Graphical User Interface
HCD	Human-Centered Design
HCI	Human-Computer Interaction
ISO	International Organization for Standardization
IT	Information technology
LCDP	Low-Code Development Platform
PaaS	Platform as a Service
SD	Service Design
SFDC	Acronym for “SalesForce Dot Com”
SQL	Structured Query Language
UCD	User-Centered Design
UEQ	User Experience Questionnaire
UI	User Interface
UX	User Experience

Acknowledgements

I would like to thank everyone who has shown interest in this study. I am thankful to my supervisor, Fabian Fagerholm, for his continuous support and invaluable feedback throughout the process. I wish to also thank my thesis advisor, Aki Teronen, for providing insights and support in the early stages of the study. Lastly, I am grateful to my family, friends, and especially my better half, for their encouragement and moral support, and for putting up with my occasional unavailability and coffee-induced wakefulness during the writing of the thesis.

Paris, 21.11.2022

Arian Kermanchi

1 Introduction

Information Technology (IT) has been rapidly evolving and businesses are pivoting accordingly to keep up with the pace of technological innovations. As the world is going towards digital transformation, customers' needs are changing and businesses will need to change their processes and offerings accordingly to maintain their market shares and expansion plans (Wilburn & Wilburn 2018). Software development is at the heart of this transformation and therefore software developers play a significant role in the process. Due to the ever-growing demand for the rapid development and deployment of enterprise applications with complex logic, Low-Code Development Platforms (LCDP) have become even more prevalent in recent years (Gomes & Brito 2022). Terms such as "low-code" and "no-code" refer to platforms and tools that allow end users without a programming background, also known as citizen developers, to participate as producers in software development processes (Sahay et al. 2020). By providing a high level of programming abstraction, they aim to reduce the learning threshold for implementing new logic and reduce the need for software development professionals. At the same time, as the complexity of software requirements increases, the disparity between the domain-specific requirements and the developers' understanding of domain- and industry-specific concepts grows larger. LCDPs aim to address this issue as well by making software development more accessible to end users with limited programming skills and high domain and industry knowledge (Khorram et al. 2020). They do so by leveraging cloud infrastructures, automatic code generation, as well as declarative and graphical abstractions (Wong et al. 2021).

As software developers are not only users of software development products, systems, and services, but also producers, they interact with the platforms in more complex manners than end-users. This has given rise to a new concept called Developer Experience (DX), which is defined by Fagerholm and Münch (2012) as "a means for capturing how developers think and feel about their activities within their working environments". The study of DX is important, as poor DX can lead to developer unhappiness and reduce motivation, which in turn affects their performance negatively. This thesis aims to compare the DX among software developers in low-code development environments with traditional development environments.

1.1 Motivation

As of the writing of this thesis, there has been little research on DX compared to user experience (UX) and even less on the differences in DX between traditional programming tools and low-code tools. The lack of comparative DX research between the two programming tool types has been acknowledged by Kuusinen (2015), Nylund (2020), and Dahlberg (2020) as well. Due to the differences in how software developers use tools compared to end users and their dual role as users and producers, not all UX research can be directly applied to DX. When developing using traditional programming languages, software developers sometimes have the option to choose their development tools of choice in terms of Integrated Development Environment (IDE) and code review and version control tools. However, as most LCDPs exist on the cloud via a Platform-as-a-Service (PaaS) model (Sahay et al. 2020), software developers are more limited in customizing their development workflows and tools when using LCDPs. This makes DX considerations even more important in the design and implementation of LCDPs.

Even though LCDPs primarily target citizen developers, professional software developers will also need to learn to use LCDPs as they get adopted by organizations. According to Accenture research (Accenture 2019), 43% of IT executives report that more than 60% of their workforce will move into new roles within three years requiring reskilling due to the impact of new technologies. Successful adoption of LCDPs requires planning, implementation, and maintenance of the tool within the organization, which not only includes technical development but also effective change management to train and convince end users and software developers to utilize the new tools. Positive developer experience increases the developers' motivation, which, according to Kinnula (1999), "influences the likelihood and the extent to which a person will actually use the time allocated to carry out the tasks and may affect the quality of the results as well". When considering the introduction of new LCDPs or switching from traditional programming tools to LCDPs, the developer experience should be considered as well, as it may directly impact the utilization and output of the developers. Therefore, this thesis aims to explore the differences between the developer experience of software developers when using traditional development platforms versus LCDPs.

1.2 Research Problem and Questions

The main goal of this thesis is to **explore the differences between developers' episodic experience when using traditional development platforms versus low-code development platforms**. In order to further elaborate on this problem, the following research questions are posed:

RQ1: How does the developer experience differ in traditional versus low-code development?
RQ2: How do developers view the development infrastructure in traditional versus low-code platforms?
RQ3: How do developers feel about their work in traditional versus low-code platforms?
RQ4: How do developers see their contribution's worth in traditional versus low-code platforms?

The first research question is a direct comparison of a traditional programming language and an LCDP. The remaining questions are based on the three categories presented in the Developer Experience Framework (Fagerholm & Münch 2012). The second question covers the **cognition** category, exploring the developers' thoughts on the infrastructure and build of each environment, the platform itself, and the learning process. The third question covers the **affect** category, focusing on how developers feel about their work in each environment and their sense of belonging and respect in the social environment. The fourth and final question covers the **conation** category, focusing on how developers experience their contribution's worth in each environment by exploring motivation, commitment, alignment, and plan.

1.3 Scope and Focus

This thesis focuses on the developer experience of software developers with varying levels of past programming experience when presented with an unfamiliar programming language versus an unfamiliar LCDP. Software developers in this study refer to traditional developers who have either studied IT or computer science or have learned and used at least one programming language in the past. This study involves an experiment using one traditional programming language and one LCDP and explores the DX when the participants use each of the tools to implement a set of requirements. This study also explores the impact of past programming experience and prior knowledge of different tools and technologies on the DX across the two tools.

1.4 Structure of the Thesis

The structure of this thesis is as follows: Chapter 1 presents the problem of the thesis and motivates the subject. Chapter 2 explores the background of the subject via a literature review and presents the programmatic and declarative tools chosen for the experiment. Chapter 3 discusses the process of designing and conducting the experiment and the choice of tools for gathering quantitative and qualitative data from the participants. The experiment results are presented and analyzed in Chapter 4. Chapter 5 revisits the research questions, answers them based on the analysis done in the previous chapter, discusses the limitations of the study, and explores a few relevant topics not directly covered in the study. Chapter 6 presents a summary and conclusion of the results of the thesis.

2 Background

This section is a literature review on the current state of the information technology sector, elaborating on the possible reasons behind the rise of low-code development platforms. Relevant concepts such as human-centered design, user-centered design, user experience, and developer experience are defined and discussed in relation to one another, the developer experience conceptual framework is introduced, and the importance of developer experience is highlighted.

2.1 Rise of Low-Code Development Platforms

IT acts as a tool that supports and enhances human workers to focus on complex thought processes such as innovation and creativity by automating more mundane tasks. Nowadays IT connects businesses, people, and governments by helping users cover distances with speed, communicate effectively, control and scale more easily, manage and distribute knowledge, and research and collaborate seamlessly. In response to the digitalization trend, businesses are looking into harnessing the power of innovative technologies such as social media, data analytics, cloud computing, mobile computing, the internet of things, blockchains, artificial intelligence, and virtual reality to influence and enhance their work processes, products, services, and business models (Urbach et al. 2018). The digitalization of products and services is a global megatrend that is rapidly changing value networks across all industries (Collin et al. 2015), where businesses adopt digital technologies to improve their business models, processes, products, and services (Gartner 2016). As businesses move towards digitalization, there is an increasing need for software to be designed and developed rapidly and with high quality and security. This has in turn increased the demand for IT professionals specializing in various technologies, paradigms, and programming languages. However, this increased demand has created a deficiency for IT professionals, particularly software engineers, in the job market (SHRM 2019, CompTIA 2019).

Most innovative digital technologies are not groundbreaking on their own, but rather become powerful tools when integrated into existing processes to increase efficiency, network possibilities, and communication speed, thereby increasing their business impact even further (Urbach & Ahlemann 2018). Companies are leaning increasingly toward IT solutions to gain and maintain their competitive advantage in the market. As a result, the demand placed on the IT departments of organizations is increasing (Mohapatra & Singh 2012). Unlike earlier times when the IT department mainly focused on ensuring regular IT operations, nowadays they are expected to also proactively identify technological innovations, build marketable solutions based on them, and thereby contribute to their organization's key value propositions (Urbach et al. 2017). However, in many organizations, IT departments are still seen chiefly as service providers without innovation skills. Urbach (2018) believes this is due to many IT departments still being predominantly technology-oriented with little business know-how. Moreover, past cost optimizations have often reduced the capacity of IT departments to a point where they struggle to test and implement new ideas beyond daily business (Urbach & Ahlemann 2018). As the strategic importance of IT is increasing, IT departments' responsibilities, importance, and role are rapidly shifting. The digitalization of various business units in organizations will usher in a rapid increase in IT expertise across all departments, thereby relieving IT departments from the pressure of delivering innovation by spreading the responsibility to all units of the organization (Urbach 2018).

The deficiency of IT talent in the job market coupled with the increasing need for departments other than IT to go through reskilling and become well-versed in using new digital solutions has given popularity to the idea of low-code development. Richardson and Rymer (2014) define a low-code development platform (LCDP) as a platform where applications are created via means such as a model-driven or graphical user interface instead of programming and explain that firms prefer to choose low-code alternatives for fast, continuous, and trial-and-error delivery by being able to create quick yet functional prototypes.

LCDPs are built upon the principles of model-driven visual programming. Burnett (1995) defines visual programming as a type of programming where additional dimensions such as multi-dimensional objects, spatial relationships, and the use of time dimension are used to convey semantics. Multidimensionality is the main difference compared to purely textual programming languages, as textual languages are at most two-dimensional: one dimension conveying the lines of code, and another dimension for optional spacing, formatting, and in-code documentation. Two common strategies used in visual programming languages are concreteness, which is the ability to express aspects of the program using certain actions (Burnett 1995), and directness, which from a cognitive perspective is achieved by reducing the actions needed by the user to achieve a certain goal (Green 1996). Both strategies can be achieved if the tool allows the user to manipulate an object or value directly rather than describing the semantics textually using code. Another key aspect of visual programming is immediate visual feedback, which is measured by Tanimoto (1990) as the speed at which semantic feedback is automatically provided by the programming tool during the editing of the program. Visual Programming and LCDPs are both considered forms of End-User Development (EUD), which is defined as any set of methods, techniques, and tools that allow end users without prior programming experience to engage in software development processes and create, modify, or extend software artifacts (Lieberman et al. 2006, Sahay et al. 2020).

A major benefit of the low-code approach is that a wider range of people can contribute to the application's development, not only those with coding skills. Nearly all the companies interviewed by Richardson and Rymer (2014) confirmed that LCDPs allow successful software projects with less-skilled developers. Other benefits of LCDPs include providing a prototyping ground for testing and experimenting with new ideas, allowing the delivery of cross-platform solutions addressing all customer channels, including mobile, and providing a single control point for application configuration, delivery, and maintenance (Richardson and Rymer 2014). In addition to many startups, tech giants such as Salesforce, Oracle, Microsoft, and Alibaba have developed their own no- and low-code development platforms, which they have been using internally and providing to their customers to extend their existing solutions. According to Gartner (2021), 70% of new enterprise applications will be developed using low-code or no-code technologies by 2025, up from less than 25% in 2020. LCDP adoption has been growing even further since the outbreak of COVID-19 due to increased concerns about time to value and IT expenditure (Greig 2020). LDCPs have already been used in practical scenarios where time to market is of the essence, such as applications built rapidly in early 2020 to manage the response to the COVID-19 pandemic (Woo 2020).

Compared to traditional software development methods that involve manual coding, LCDPs provide improvements in automating and speeding application delivery and help with high vendor growth rates. However, Lethbridge (2021) argues that, in practice, it is common for enterprises to write far more code and far more complex code than the LCDP designers originally expected. Therefore, the method of building powerful applications with little code can result in increasing amounts of technical debt. Lethbridge (2021) gives Microsoft Excel as an example, explaining that a spreadsheet may be packed with complex formulas and macros, yet none of the logic can be easily browsed, tested, documented, or reused. From the enterprise's viewpoint, Richardson and Rymer (2014) highlight the risks of companies utilizing too many low-code solutions provided by multiple small vendors and having little consensus on how LCDPs fit into their larger enterprise software landscape. Tisi et al. (2019) identify scalability as an issue, explaining that LCDPs are currently mainly utilized for the development of small applications and their use in mission-critical enterprise applications and large-scale projects is not yet covered. They also highlight fragmentation as a potential risk, as depending on the vendor and the particular programming model they use, different low-code development paradigms can be created, eventually forcing users to learn multiple tools that are potentially incompatible with one another.

2.2 Human-Centered Design and Developer Experience

The International Organization for Standardization (ISO 2019) defines human-centered design (HCD) as a design process focused on usability and ergonomics, putting human perception, interaction, and psychology above other considerations when designing services. On the other hand, user-centered design (UCD) is an iterative design process where each phase of the design process is focused on the end users and their needs. According to ISO (2019), these terms are often used synonymously. Roto et al. (2021) have discovered that UCD also overlaps with Service Design (SD). However, after interviewing academics and industrial practitioners in both UCD and SD fields, they have concluded that UCD is considered to have a deeper focus than SD, which is considered to be abstract and broad in nature (Roto et al 2021). Furthermore, while UCD is often seen as a subset of SD, it contains certain user testing and research methods that are not found in SD. While UCD refers to the process or strategy applied to engineer experiences, user experience (UX) deals with the actual experiences users have with certain products. The ISO (2019) defines UX as a "person's perceptions and responses resulting from the use and/or anticipated use of a product, system, or service." Similarly, Law et al. (2009) define UX as something individual (instead of social) that comes from users interacting with a product, system, service, or object via a user interface.

UX has emerged from the traditional field of human-computer interaction (HCI), a field of research that explores the interface and interaction between humans and computers. It drills down on the HCI scope and specifically focuses on user interactions with products, systems, and services. Software developers are not only users of software development products, systems, and services, but also producers of them. Therefore, software developers interact with such artifacts in complex manners, as they iteratively use and produce such tools. Kuusinen et al. (2016) explain that while UX considers the context of the use of a system, developer experience (DX) considers "the context of software development, including aspects beyond software tools, such as development processes, modeling methods, and other means of structuring software engineering tasks." Due to the user/producer roles of software developers, they also occasionally alter their own experiences by modifying their development environments by changing the configurations and using different development tools (Kuusinen et al. 2016). This process of users modifying software based on their use cases and needs is known as software appropriation (Dourish 2003). Tchounikine (2017) explains that, in some cases, the software may also be used in ways that it was not originally designed for, and this is particularly likely among power users such as developers. Fagerholm and Münch (2012) explain that UX involves the experiences of using a product (user-centric) while DX focuses on the development processes of creating products (process or product-centric), each bringing its own priorities and goals.

While there is little consensus currently on the definition and measurement of DX, one comprehensive definition is brought up in the shape of a framework by Fagerholm and Münch (2012), where the Developer Experience concept is categorized into three areas: cognitive (how developers perceive the development infrastructure), affective (how developers feel about their work), and conative (how developers see the value of their

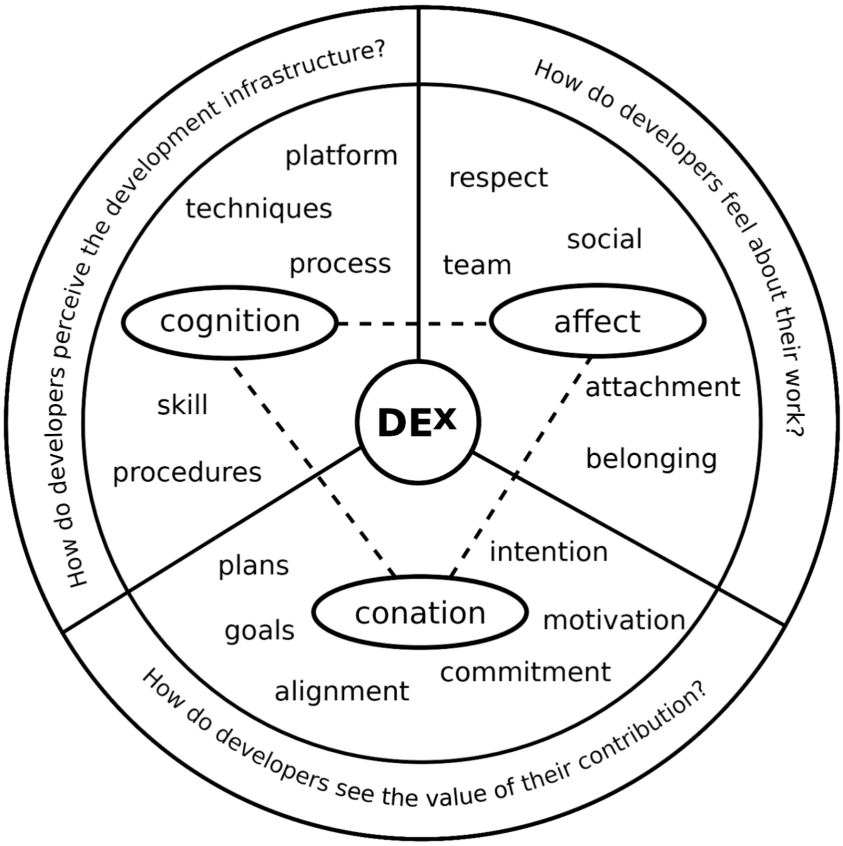

Figure 1: Developer Experience: Conceptual framework. (Adapted from Fagerholm Münch 2012.)

contribution). As shown in Figure 1, each of the three areas contains several subsections that explore different aspects of how the developer interacts with the infrastructure, their own work, and their contribution. For example, a developer's perception of infrastructure elements such as platforms, programming languages, frameworks, and methods may change. They may also feel respect for and attachment to their work, as well as social factors that contribute to a sense of belonging. Finally, they might see the value of their contribution and feel fulfilled by carrying out their goals and aligning them with others in the environment.

Fagerholm (2015) further puts DX into the social and technical environments and builds on Fagerholm's and Münch's (2012) psychological division of DX. The technical environment includes programming languages, infrastructure, processes, techniques, plans, diagrams, et cetera., and every time a developer interacts with one of these artifacts, they generate an experience (Nylund 2020). Depending on the time period

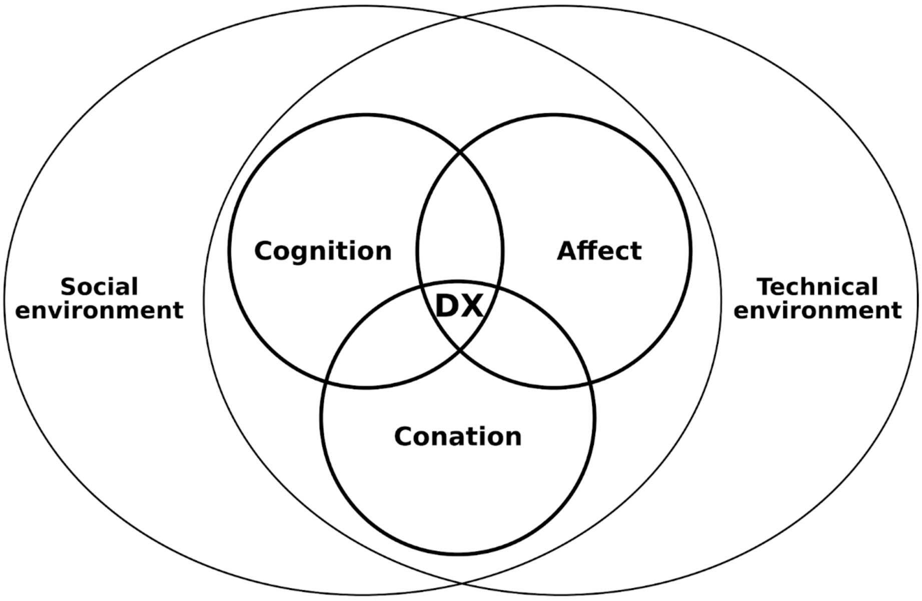

Figure 2: The Developer Experience Concept. (Adapted from Fagerholm 2015.)

considered, these experiences can be classified as immediate, such as keystrokes and moment-to-moment changes, episodic, such as user sessions of certain tools or interfaces, or cumulative, which focus on a set of experiences stacking up over a certain time period (Fagerholm 2015). The conative and affective sub-categories of DX consider the social aspects of being a software developer, focusing on what the developer feels when trying to achieve their goals. In addition to the technical environment, social factors such as the social impact, variety, creativity, and difficulty of the work as well as coworker engagement and feedback culture within the team can also affect developer happiness and motivation (Oliveira and França 2019). Technical debt can also impact developer experience. Past research has shown that technical debt negatively affects the developer's emotions (Yli-Huumo et al. 2014, Codabux and Williams 2013) and morale (Spínola et al. 2013, Peters 2014, Tom et al. 2013), as it reduces work value, hinders progress, and decreases confidence (Besker et al. 2020).

The social aspect of DX also includes the study of the motivation of developers (Morales et al., 2019). While the quality of technical tools and interfaces that developers use in their work can positively or negatively affect their motivation (Kuusinen et al., 2016), Fagerholm (2015) explains that motivation and desire often come as a response to the social environment around the individual. The developer's experiences with the social environment can directly affect the outcome of software development projects. Graziotin et al. (2017a, 2017b) explain that developer unhappiness can impact productivity and performance and may lead them to take shortcuts, leading to lower code quality and, in extreme cases, causing developers to resign. Unhappiness can ultimately lead to low motivation among developers (França et al. 2014, Graziotin, 2017a). In a study done by Baltes and Diehl (2018), when asked about the possible causes of poor work performance of colleagues, the most common answer given by developers was demotivation. On the other hand, motivating factors for developers include solving problems, seeing the positive impact of their work, creating new solutions, and helping peers (Baltes and Diehl 2018). Graziotin et al. (2015) suggest that intervention on the affect experience of developers may "have relatively low costs but astonishing benefits". According to Capretz (2014), compared to the technical aspects, the social and human aspects of software engineering get less attention than they deserve, even though they are also important and social and human shortcomings can also contribute to software project failures alongside technical shortcomings.

3 Research Methods and Procedure

In order to explore the research questions described in Section 1.2, an experiment was conducted on the Salesforce Platform, which provides declarative and programmatic tools for customizing their applications. This section presents the platform where the experiment was conducted as well as the methods used for data gathering, participation selection, experiment design, and execution.

3.1 Salesforce Platform

Salesforce is a cloud-based enterprise information solution provider whose core solution is a customer relationship management (CRM) platform. Enterprise information systems are integrated information systems that support core business processes and functions (Turban & Volonino 2011). From the business's point of view, enterprise information systems are systems that enhance the performance of companies allowing seamless data sharing across departments and with external business partners. With the use of technology, such systems allow companies to analyze information and transaction processes in real-time across the entire organization (Turban & Volonino 2011). Enterprise systems are more desirable than legacy department-specific systems because they not only optimize business processes further but also allow companies to have unprecedentedly high-level visibility into end-to-end processes (Cleo et al. 2019). CRM systems are enterprise information systems that help segment and manage customers to optimize their long-term value throughout their life cycles through targeted sales processes. (Turban & Volonino 2011)

As shown in Figure 3, Salesforce is a leader in Gartner's Magic Quadrant for Enterprise Low-Code Application Platforms (Wong et al. 2021) and, according to Forrester (2021), it remains "a solid choice for current customers", being noted as a "strong performer". Its LCDP offering is the Salesforce Platform (also known as SFDC), where many actions can be performed using Graphical User Interface (GUI) configuration and coding is only necessary when the GUI tools are insufficient to meet the requirements. One of the main LCDP solutions available in Salesforce is the Salesforce Flow, which is

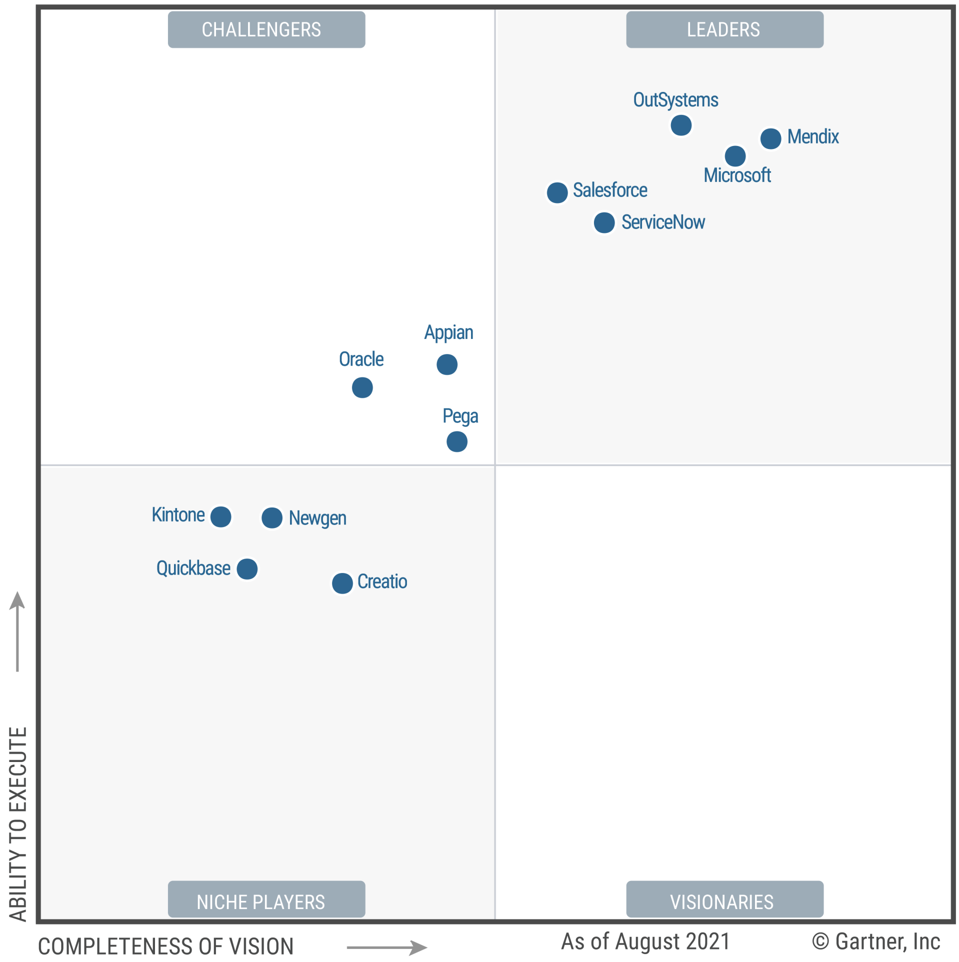

Figure 3: Gartner's Magic Quadrant for Enterprise Low-Code Application Platforms, August 2021. (Adapted from Wong et al. 2021.)

the product that encompasses building, managing, and running Flows, which are multi-step logical processes that allow the automation of business processes. Flow Builder is a point-and-click tool for building Flows, allowing users to create complex automation logic without the need to write code (Salesforce Trailhead 2022). Salesforce Flows are made up of three building blocks: elements, which are bits of logic, connectors, which define the path that the Flow takes at runtime, and resources, that act as constant or variable data containers that can be accessed at any time by elements.

The Salesforce Flow illustrated in Figure 4 performs a simple conditional logic. Every time an account record is modified in the system, if the account's description field is empty, then it is populated with the account's name. Otherwise, the SOW checkbox is checked on the account record. In addition to the LCDP offerings, Salesforce allows

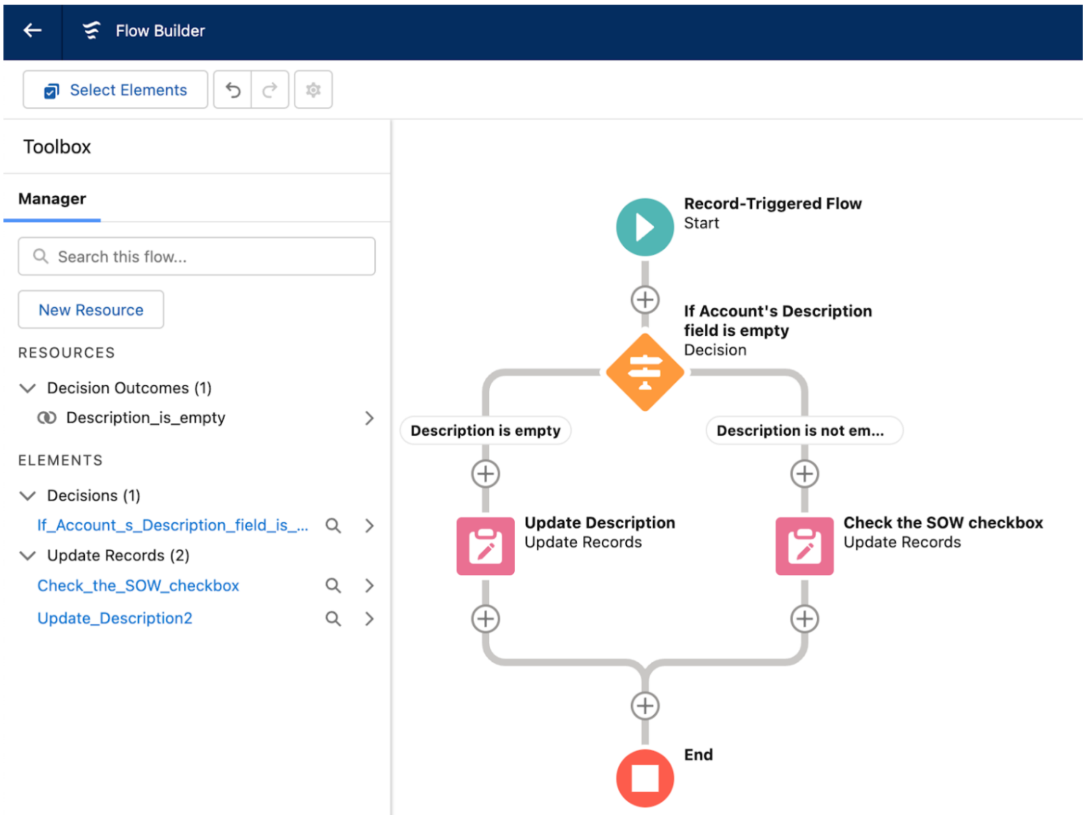

Figure 4: Example of a Salesforce Flow.

customization using the APEX programming language. Official Salesforce Documentation (Salesforce Developers 2010) states that APEX is ”a strongly typed, object-oriented programming language that allows developers to execute transaction control statements on Salesforce servers along with calls to the API”. In this way, Salesforce allows its users to choose the amount of coding they want to use in their solutions based on the complexity of the solutions and the availability of developer resources. The logic in Figure 4 can also be written as an APEX class, as demonstrated in Figure 5. In this study, Salesforce Flow Builder and the APEX programming language are selected as the tools of choice for the experiment.

3.2 Data Collection Methods

Both questionnaires and interviews were used to gather both quantitative and qualitative data in the experiment. Qualitative research interviews are a way of capturing qualitative

```
trigger accountTrigger on Account (before update) {
    for(Account acc :Trigger.New) {
        if(acc.Description == null) {
            acc.Description = acc.Name;
        } else {
            acc.SOW_Checked__c = true;
        }
    }
}
```

Figure 5: Example of an APEX class.

data about personal experiences and thoughts (DiCicco-Bloom & Crabtree 2006). The interviews in this experiment were conducted using semi-structured questions after both development sessions were done. Semi-structured interviews allowed the researcher to spontaneously create follow-up questions and get even more detailed answers regarding each interviewee's personal experience and thoughts, allowing more flexibility and depth in the interviews compared to structured and closed-question interviews (Beck & Manuel 2008). Beck and Manuel (2008) also explain that interviews provide insights into human behavior at a deeper level than quantitative data points and are useful for research questions that focus more on experiences rather than numerical values.

Questionnaires also help capture qualitative and quantitative data from the target audience. They provide objective means of gathering information about people's beliefs, attitudes, knowledge, and behavior (Oppenheim 1992, Boynton & Greenhalgh 2004). The power of questionnaires lies in their customizability. for instance, closed-ended designs such as checkboxes would quickly produce quantitative data at the cost of a loss of richness, while open-ended designs containing free-text inputs can add qualitative angles to the otherwise quantitative data (Boynton & Greenhalgh 2004). Despite the abundance of specialist literature on questionnaire design, poor question phrasing, inappropriate instruments, and lack of rigor can lead to questionnaires that fail to produce high-quality generalizable data (Gillham 2000, Boynton & Greenhalgh 2004).

3.3 Experiment Design

The experiment aims to compare the participants' performance, thoughts, and feelings when using Salesforce Flow Builder and the APEX programming language in an unbiased manner and as objectively as possible. Therefore, an attempt was made to keep most variables constant: the participants received the same amount of training in Flow Builder and APEX right before using each tool and they performed the exact same task using each tool. Furthermore, in order to reduce the impact of different development environments on the experiment's results, all the participants were asked to use the same integrated development environment (IDE), namely the web-based Salesforce Developer Console.

The task itself was deliberately designed to be easy to complete using either tool, but difficult to fully complete within the time limit of 60 minutes. It was further divided into three discrete sub-tasks in ascending order of difficulty in order to allow the participants to complete part of the task at least and not lose motivation in case they run out of time. The task description is available in Appendix A.

Two questionnaires were developed for this experiment: the Pre-Experiment Questionnaire and the Post-Challenge Questionnaire. The Pre-Experiment Questionnaire (see Appendix B) consisted of questions about the participant's academic and professional background and their prior familiarity with Salesforce as well as some popular programming languages and LCDPs. The purpose of this questionnaire was to be able to understand the effect of the participant's background on their developer experience when using Flow Builder or APEX. The Post-Challenge Questionnaire (see Appendix C) was designed based on a combination of parts one and three of the Survey for Measuring DX of LCDPs (Gao 2022) as well as the User Experience Questionnaire (UEQ), a well-known UX questionnaire aiming to cover a comprehensive impression of the end users' experience (Laugwitz 2008). The purpose of the post-challenge questionnaire is to capture the participant's episodic experience of each hands-on challenge as it occurs under specific circumstances. The questionnaire's questions are answered on a 7-point Likert scale. In order to prevent participants from giving the same answer to multiple questions in a row, the order of the positive and negative terms in each pair is randomized throughout the questionnaire (i.e. some items start with the positive term and others start with the negative term). All questionnaires were shared with the participants online using Google Forms and the responses were received online as well.

3.4 Experiment Execution

The experiment was conducted with participants of various technical backgrounds who had little to no prior experience in Salesforce development. The experiment included 18 participants in total. They were all asked to sign the consent form to agree to voluntarily participate in the research study and were given a privacy notice and an information sheet prior to consenting. The procedure overall complies with the guidelines of the Aalto University Research Ethics Committee.

The experiment consisted of three questionnaires, two training sessions, two hands-on experiment sessions, and one interview per participant. At first, a two-hour teaching session was conducted, where the basics of Salesforce Flow Builder were taught to the participants. Afterward, each participant took part in a hands-on development session (the declarative task), where they were tasked to use Salesforce Flow Builder to implement a set of requirements explained in Section 3.2. The participants had exactly one hour to finish the task. Immediately after the hands-on session, each participant was asked to fill out the post-development questionnaire, allowing the researcher to capture each developer's immediate thoughts and feelings after having completed the task and gather data on their episodic experience based on their recollection. A week later, a similar two-hour teaching session was conducted to teach the participants the basics of the APEX programming language, then the participants were tasked to implement the exact same requirements, but this time using the APEX language (the programmatic task). After the session, the same post-development questionnaire was given to each participant to capture their immediate thoughts and feelings about the APEX programming task.

After completing both hands-on sessions and returning the questionnaires, each participant took part in a 20-minute semi-structured interview, where they were asked a set of predefined questions related to each of the research questions, but were allowed to freely lead the discussion away from the questions as well. At the end of each interview, the participant was asked if they had any final comments to share. All interviews were conducted online and in English. Furthermore, each participant's solutions were graded against a set of test cases and were given a grade from 1 to 10, depending on the degree of success the participants had in implementing the requirements. For each declarative and programmatic solution, the three test cases described in Appendix D were executed by the researcher, and points were given for each expected result observed.

4 Results and Analysis

This section first presents the data on the background information of the participants and the post-challenge survey responses for both the declarative and the programmatic tasks and categorizes the participants and the data. Afterward, the Paired Samples T-test was conducted on the newly grouped data and the bivariate correlations using the Pearson correlation coefficient were used to analyze potential correlations and formulate findings based on the quantitative data along with the qualitative inputs collected during the interviews.

4.1 Results presentation

The Pre-Experiment Questionnaire provided information on the backgrounds of the participants. The descriptive statistics of this questionnaire's responses are displayed in Tables 1 and 2 along with the average grades that the participants received in each task. As seen in Table 1, on average the participants performed better in the declarative task compared to the programmatic task and the spread of their grades in the programmatic task is more than twice that in the declarative task (standard deviation of 2.372 as opposed to 1.026). Furthermore, the standard deviation of their past programming experience is greater than that of the familiarity questions. This may be partially due to the Dunning-Kruger effect (Kruger and Dunning, 1999), which states that due to the connection between competence and metacognitive skills, the less experienced participants overestimate their competence, while more experienced participants underestimate their competence as much as 20% below their actual abilities.

As exactly half of the participants had reported having two or more years of programming experience, they were divided into groups of experienced juniors (those with two or more years of programming experience) and inexperienced juniors (those with under two years of programming experience). Furthermore, the programming languages and technologies were grouped into *multi-paradigm* (Python, JavaScript, C++, and PHP), *object-oriented* (Java and APEX), and *LCDP* (Scratch, Flow, and Process Builder) and the means of the participants' responses to each question were calculated

Table 1: Means and standard deviations of the responses to the Pre-Experiment Questionnaire. Range of familiarity values from 1 (no familiarity) to 7 (very familiar). Range of years of programming experience from 0 to 6. Range of task grades from 0 to 10 (grading criteria explained in Appendix D). N = 18.

Question	Mean	Standard Deviation
How familiar are you with Python?	2.72	1.602
How familiar are you with JavaScript?	4.06	1.697
How familiar are you with Scratch?	1.94	1.11
How familiar are you with Java?	4.28	1.742
How familiar are you with C++?	2.61	1.42
How familiar are you with PHP?	2.61	1.577
How familiar are you with Salesforce Flow?	3.28	1.179
How familiar are you with Salesforce Process Builder?	2.78	0.808
How familiar are you with APEX?	2.5	0.786
How familiar are you with low-code programming?	3.61	1.145
How familiar are you with programming?	5.11	1.605
Programming experience in years	2.06	1.83
Declarative Task: Grade	9.1	1.026
Programmatic Task: Grade	7.28	2.372

based on the new groupings. None of the participants reported having used Salesforce products significantly before the experiment. The results divided by participant group are illustrated in Table 2, which further supports the possible impact of the Dunning-Kruger effect, as inexperienced juniors reported themselves to be as competent or more competent than experienced juniors in all three categories of *multi-paradigm experience*, *object-oriented experience*, and *LCDP experience*. This is in contrast with the groups' actual performance in the tasks, where experienced juniors performed significantly better in the programming task compared to inexperienced juniors (with a score of 8.67 versus 5.39). Both groups, however, performed similarly at the declarative task, with inexperienced juniors even performing slightly better than experienced juniors (with a score of 9.11

versus 9.08).

Table 2: Means and standard deviations of the responses to the Pre-Study Questionnaire, grouped by technology types and calculated per participant group. Range of familiarity values from 1 (no familiarity) to 7 (very familiar). Range of years of programming experience from 0 to 6. Range of task grades from 0 to 10 (grading criteria explained in Appendix D). N = 18.

	Experienced juniors		Inexperienced juniors	
Question	Mean	Standard Deviation	Mean	Standard Deviation
Multi-paradigm experience	2.72	1.087	3.28	0.843
Object-oriented experience	3.39	1.054	3.39	0.993
LCDP experience	2.22	0.601	3.11	0.527
How familiar are you with low-code programming?	3.11	1.269	4.11	0.782
How familiar are you with programming?	6.56	0.726	3.67	0.500
Programming experience in years	3.56	1.333	0.56	0.527
Declarative Task: Grade	9.08	1.173	9.11	0.928
Programmatic Task: Grade	8.67	1.118	5.89	2.522

Table 3 shows the participants' responses to the easiness and understandability of the tasks. On average, experienced juniors found both tasks to be easier to understand and complete than inexperienced ones. Surprisingly, inexperienced juniors found the programmatic task easier to complete than the declarative task, even though they performed worse in the programmatic task. Inexperienced juniors found the task description more difficult to understand than experienced ones, which could be partially explained by their lack of programming experience. In general, participants found the task to be moderately easy to complete (mean score of 5.28 out of 7), which is in line with the direction of the task design.

Table 3: Means and standard deviations of the easiness and understandability of the task description scored from 1 (completely disagree) to 7 (completely agree), calculated per participant group. N = 18.

	Experienced juniors		Inexperienced juniors		All participants	
Question	Mean	Standard Deviation	Mean	Standard Deviation	Mean	Standard Deviation
Declarative task was easy to understand	6.11	1.269	4.11	1.616	5.11	1.745
Declarative task was easy to complete	5.44	1.236	5.11	1.167	5.28	1.179
Programmatic task was easy to understand	6.11	1.269	3.89	1.764	5.00	1.879
Programmatic task was easy to complete	5.33	1.225	5.22	1.202	5.28	1.179

The participants also answered the Post-Challenge Questionnaire after each of the two hands-on challenges. The UEQ groups its questions into six attributes: attractiveness, perspicuity, efficiency, dependability, stimulation, and novelty (Laugwitz 2008). Laugwitz (2008) describes the categories as:

- **Attractiveness**: Overall impression of the product. Do users like or dislike the product?
- **Perspicuity**: Is it easy to get familiar with the product? Is it easy to learn how to use the product?
- **Efficiency**: Can users solve their tasks without unnecessary effort?
- **Dependability**: Does the user feel in control of the interaction?
- **Stimulation**: Is it exciting and motivating to use the product?
- **Novelty**: Is the product innovative and creative? Does the product catch the interest of users?

The additional five descriptive questions taken from the Survey for Measuring DX of LCDPs (Gao 2022) are also added to the attributes provided by UEQ (Laugwitz 2008) in order to simplify the data presentation and analysis, as illustrated in Table 4. The final list of descriptive questions and attributes is available in Appendix E. The grouped responses for each participant group as well as for the whole group are illustrated in Figure 6, which highlights the difference in responses given by inexperienced and experienced juniors. The two variables *Task Was Easy to Understand* and *Task Was Easy to Complete* are not added to any attribute category and are treated as stand-alone variables, as they describe the task itself (as described in Appendix A), while the descriptive questions describe the tools.

Table 4: Descriptive questions adapted from the Survey for Measuring DX of LCDPs (Gao 2022) and their attribute categories.

Attribute category	Negative term	Positive term
Stimulation	Exhausting	Smooth
Efficiency	Distracted	Concentrated
Dependability	Lack of control	In control
Dependability	Limited	Free to explore
Efficiency	Unproductive	Productive

4.2 Data analysis

This section builds on the descriptive statistics and data groupings presented in Section 4.1 to analyze correlations between various variables using paired sample t-tests, bivariate Pearson correlation analysis, and a multivariate general linear model. The statistical analysis and observations are summarized into concrete findings that are then used to address the study's research questions.

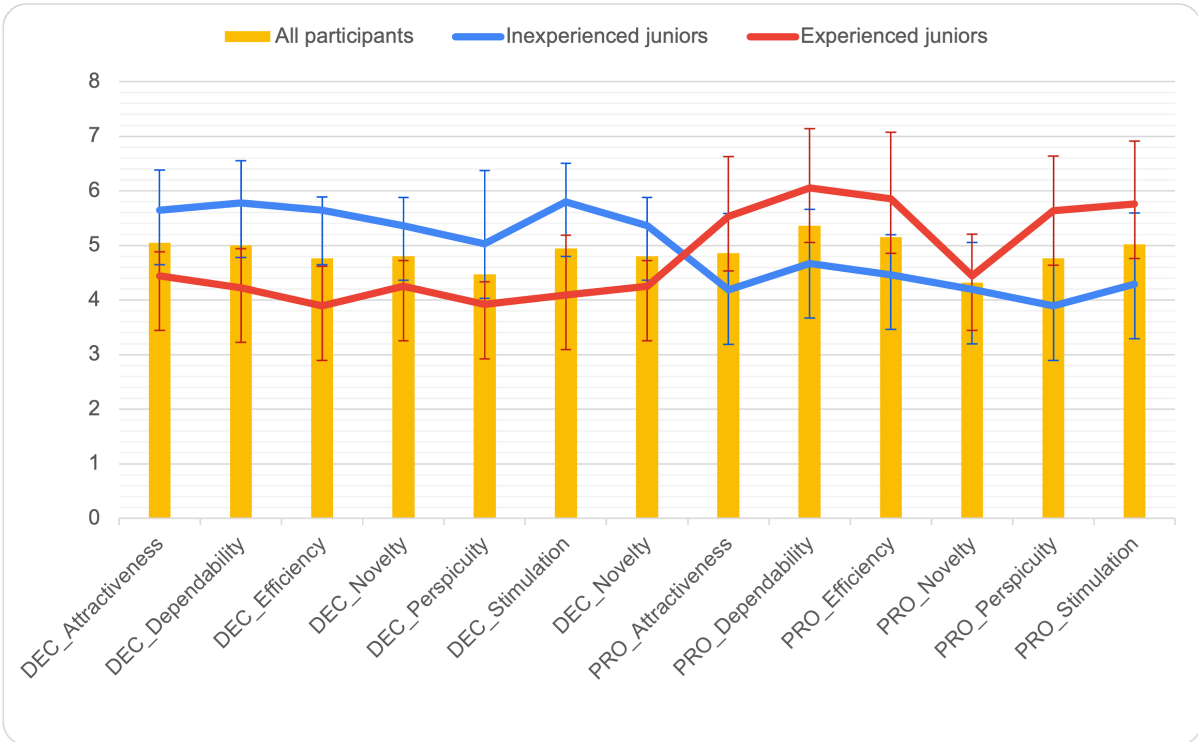

Figure 6: Mean responses along with standard deviation to each attribute category per participant group and in total. N = 18.

4.2.1 Paired Sample T-Test

Looking at Figure 6, experienced juniors have, on average, given higher ratings in all categories to the programmatic tool. On the other hand, inexperienced juniors have strongly favored the declarative tool. One exception is the *PRO_Novelty* attribute, where inexperienced and experienced juniors have given similar results (the inexperienced gave on average 4.19 and the experienced gave 4.44). This implies that neither participant group found the programmatic solution significantly more creative, inventive, leading edge, or innovative than the other. It is worth noting that experienced juniors found the programmatic tool still more novel than the declarative one (4.44 versus 4.25). In order to further analyze the experiences of each participant group using each tool, a paired sample two-sided T-test is performed. The test results in Table 5 further support the notion that experienced juniors favor the programmatic tool and inexperienced ones favor the declarative tool with a two-sided significance (p) at the 0.05 level.

The only exception to the tool preferences of the participant groups shown in

Table 5: Paired sample t-test of grouped attributes, separated by participant group. Only t-values with two-sided p-value under 0.05 are shown. N = 18.

Variable pair	Experienced juniors t-value	Inexperienced juniors t-value
DEC_Attractiveness - PRO_Attractiveness	-2.512	4.916
DEC_Dependability - PRO_Dependability	-4.093	4.558
DEC_Efficiency - PRO_Efficiency	-3.109	4.945
DEC_Novelty - PRO_Novelty	n/a	3.395
DEC_Perspicuity - PRO_Perspicuity	-4.064	2.444
DEC_Stimulation - PRO_Stimulation	-2.519	3.915

Table 5 is the *PRO_Novelty* attribute, where the experienced juniors did not favor either tool over the other. Drilling further into the *Novelty* attribute category, experienced developers consider the programmatic tool more creative ($|t| = 1.633$), yet slightly less inventive ($|t| = 0.664$) and innovative ($|t| = 0.800$) than the declarative tool. The relative lack of interest in either tool in terms of novelty among experienced juniors was also echoed by some of the participants in the interviews. One participant with 3 years of development experience and prior experience with the Java language said that the programmatic tool "felt like your average Java", commenting on how the APEX language is strongly typed and object-oriented like Java and has a Java-like syntax. Another participant with 6 years of experience said that "one of the nicest things about APEX is how you can combine SQL and Java in creative ways". On the other hand, inexperienced juniors consistently found the declarative tool more creative and leading-edge than the programmatic tool (t-value of over 2.5 with a two-sided significance level of 0.05). One developer with zero years of programming experience said that the declarative tool is "futuristic" and another said that "this could be the future of programming". These observations lead to Finding 1:

Finding 1: Developers with more programming experience found neither tool more novel than the other, while developers with less programming experience found the declarative tool more creative, leading-edge, innovative, and inventive.

4.2.2 Bivariate Pearson Correlation Analysis

Going back to the relationship between the years of programming experience and the attitude towards each tool, Table 6 shows all bivariate correlations significant at the 0.001 level (2-tailed), and the absolute Pearson correlation coefficient of greater than 0.7 for the variable "past programming experience in years" against each descriptive variable pair.

Table 6: Bivariate correlation of the participant's reported years of programming experience against different variables, sorted by the Pearson correlation coefficient, where |p|>0.7. All correlations are significant at the 0.001 level (2-tailed). N = 18.

Correlated Variable	Attribute Category	Pearson Correlation
Declarative Tool: Dull or Creative	Novelty	-0.904
Declarative Tool: Lack of Control or In Control	Dependability	-0.81
Declarative Tool: Inferior or Valuable	Stimulation	-0.8
Declarative Tool: Annoying or Enjoyable	Attractiveness	-0.794
Declarative Tool: Impractical or Practical	Efficiency	-0.778
Declarative Tool: Bad or Good	Attractiveness	-0.751
Declarative Tool: Distracted or Concentrated	Efficiency	-0.749
Declarative Tool: Inefficient or Efficient	Efficiency	-0.74
Declarative Tool: Unproductive or Productive	Efficiency	-0.726
Programmatic Tool: Exhausting or Smooth	Stimulation	0.701
Programmatic Tool: Inefficient or Efficient	Efficiency	0.71
Programmatic Task: Task was easy to understand	n/a	0.736
Programmatic Tool: Lack of Control or In Control	Dependability	0.765
Programmatic Tool: Unfriendly or Friendly	Attractiveness	0.771
Programmatic Tool: Slow or Fast	Efficiency	0.775

Table 6 shows that there is a significant correlation between the years of programming experience of the participant and their ratings of the declarative and programmatic tools. Participants with more years of programming experience considered the declarative tool duller, more uncontrollable, inferior, annoying, and unproductive. An experienced junior said about the declarative tool in the interview that ”I can see that they try to make it simplistic, but it almost feels too simplistic”, and another said, ”I do not consider this programming - maybe very high-level programming at best”. On the other hand, participants with more years of experience considered the programmatic tool to be fast, friendly, controllable, efficient, and smooth. An experienced junior said about the programmatic tool that ”up until the last task I had a really good flow” and another said, ”I was definitely excited and it felt good since I got into a flow very fast”.

The interviews support the correlations from the other end as well: participants with fewer years of programming experience found the declarative tools more creative and controllable and enjoyable and find the programmatic tool slow, unfriendly, and exhausting. Several participants with under one year of programming experience mentioned the steep learning curve, difficulty, and confusing error messages of the programmatic tool, making comments like ”it was challenging in a negative way”, ”it gives me anxiety because when something goes wrong, you do not know where or how it happened”, and ”you feel distracted by many small details”. On a general level, these observations can be summarized in Finding 2:

Finding 2: There is a significant correlation between the number of years of programming experience and the tool preference, such that the more experience the developer has, the more they will prefer the programmatic tool over the declarative tool.

Two descriptive variable pairs are present in both declarative and programmatic tools in Table 6: *Lack of Control or In Control* and *Inefficient or Efficient*, meaning that developers with more experience felt less in control when using the declarative tool and found it less efficient, while they felt more in control of the programmatic tool and found it more efficient, and vice versa for developers with less experience. The notion of control was mentioned by multiple participants in the interviews as well. One experienced junior said ”I would want to have more details and be able to have more control over the logic I am building” and another wondered what type of code the declarative tool is actually generating and regretted the inability to check the code's complexity and efficiency and have control over its optimization. Another experienced developer had difficulties with the declarative tool, as they were trying to create the needed variables before implementing their logic, unaware that variable assignment is automatically handled in the background by the tool. It appears that more experienced developers viewed the declarative tool as a barrier between themselves and the actual code, constraining their control over the code and causing inefficiencies.

In contrast to the experienced juniors, the inexperienced juniors strongly preferred the declarative tool over the programmatic one, finding themselves more in control when using it and considering it more efficient. A participant with no previous programming experience said "for me, it was super interactive, super friendly, ... and very efficient" and another said, "I felt like I was in control of the tool, ... with APEX I could not get it to do what I wanted". Several inexperienced juniors mentioned the helpfulness of visual and human-friendly feedback messages provided by the declarative tool and said that they were struggling to understand and decipher compiler errors that the programmatic tool gave when their code failed to execute, explaining that debugging was more efficient with the declarative tool. None of the experienced juniors mentioned this, so this issue may be fixed once the participants become more comfortable with reading compiler errors and debugging code. The findings related to the *Lack of Control or In Control* and *Inefficient or Efficient* descriptive variable pairs lead to Findings 3 and 4:

Finding 3: There is a significant correlation between the number of years of programming experience and the feeling of control, such that developers with more experience feel more in control of the programmatic tool, and those with less experience feel more in control of the declarative tool.

Finding 4: There is a significant correlation between the number of years of programming experience and the notion of efficiency, such that developers with more experience consider the programmatic tool more efficient, and those with less experience consider the declarative tool more efficient.

In addition to a significant correlation between the increasing years of experience and the tendency to find the programmatic tool more efficient than the declarative one, as shown in Table 7, there is also a significant negative correlation between *DEC_Efficiency* and *PRO_Efficiency*, meaning that the participants either found the declarative tool efficient or the programmatic one, but not both. This is the only significant correlation found across the declarative and programmatic attributes. The rest of the correlations are either between two attributes of the same tool or between an attribute and years of programming experience. The full table of grouped bivariate correlations can be found in Appendix F. Table 7 also shows that regardless of the years of programming experience, the more familiar the participants are with LCDP tools, the more they consider them to be efficient, implying that with more exposure and training developers may begin to consider the declarative tool to be more efficient. The findings in Table 7 lead to Findings 5 and 6:

Finding 5: Participants either find the programmatic tool efficient or the declarative one, but not both.

Finding 6: There is a significant positive correlation between past LCDP experience and finding the declarative tool efficient.

Table 7: Bivariate correlation of efficiency-related variable pairs where |p|>0.7. All correlations are significant at the 0.001 level (2-tailed). N = 18.

First variable	Second variable	Pearson Correlation
DEC_Efficiency	PRO_Efficiency	-0.780
Past Programming experience in years	DEC_Efficiency	-0.944
Past Programming experience in years	PRO_Efficiency	0.760
LCDP_XP	Declarative Task: Inefficient or Efficient	0.727

Some Salesforce-specific terms in the task description (Appendix A) were difficult to understand for some of the participants. As none of the participants had been regular users of Salesforce products in the past, several reported having difficulties understanding the meanings of object relationships (for example, ”Opportunity’s Account’s Owner’s Email” versus ”Opportunity’s Owner’s Email”). As seen in Table 3, experienced juniors found the task description more understandable than inexperienced ones. This observation is supported by the bivariate correlations in Table 8, where there is a statistically significant positive correlation at the 0.05 level (2-tailed) between the years of programming experience and the understandability of the task for the participant, which also may have impacted their performance in completing the tasks.

Table 8: Bivariate correlation of the participant’s reported years of programming experience against task understandability. N = 18.

Variable	Pearson Correlation	Significance (2-tailed)
Declarative task was easy to understand	0.736	<0.001
Programmatic task was easy to understand	0.532	0.023

Only one statistically significant correlation was found between the participants’ task performance and their developer experiences: there was a significant negative correlation between the programmatic task performance and the *DEC_Novelty* score given by the participants, with a Pearson coefficient of -0.783 and two-tailed significance of under 0.001. This means that participants who performed better at the programmatic task found the declarative tool less creative, inventive, leading edge, and innovative. In conjunction with Findings 1 and 7, this correlation implies that participants with more experience who performed better at the programmatic task found the declarative tool less novel, which is supported by the negative correlation between the years of experience and the *Dull or Creative* descriptive variable pair found in Table 6. The lack of statistically significant correlations otherwise implies that the participants’ success (or lack thereof) in the tasks did not significantly influence their responses to the Post-Challenge Questionnaire about their episodic experiences.

Seven participants noted in their interviews that a major limitation of the declarative tool may be in its scalability. When asked whether they see themselves using the tool in the future, one inexperienced junior said that "I may occasionally use it for prototyping, but in the end, I would write my logic in code" and another said, "maybe I would use it for simple requirements or if there is a tight schedule, but the tool can become very complicated when the logic becomes more complex". An experienced junior brought up the problem of code optimization, saying that they are not convinced that any LCDP would be able to generate code as optimized as an experienced developer could manually create. Another experienced junior said "it is harder to see the big picture of the code base when dealing with Flows.", further raising the issue of scalability. Even though participants of varying experience levels have mentioned this as a potential issue, there is no quantitative data to show a significant correlation between years of programming experience and concern for scalability.

4.2.3 Multivariate General Linear Model

Participants in general performed better in the declarative task compared to the programmatic task. The paired samples t-test of the declarative task grade to the programmatic task grade gives a t-value of 3.676 at a two-sided significance of 0.002, indicating that there is a statistically significant difference between participants' performance in the two tasks. Table 2 shows that even though all participants performed worse in the programmatic task, experienced juniors performed better than inexperienced juniors on average. This proposition is further supported through statistical analysis by the multivariate general linear model, using the declarative and programmatic task grades as dependent variables against the years of programming experience. The F test of the Wilks' Lambda shows that there is a statistically significant relationship ($p < 0.001$) across the dependent variables (participant performance in both tasks) as a function of the independent variable (years of programming experience). The tests of between-subjects effects in Table 9 show that for the declarative task's grade there is no statistically significant difference as a function of the years of programming experience ($p = 0.135$), but there is a significant difference for the programmatic task's grade ($p = 0.003$).

As the mean grade of experienced juniors in the programmatic task is higher than that of inexperienced juniors, it seems that the years of programming experience

Table 9: Tests of between-subject effects with programmatic XP in years as the independent variable. Wilks' Lambda 0.048 with F test 5.911 and $p < 0.001$. N = 18.

Source	**Dependent Variable**	**Type III Sum of Squares**	**df**	**Mean Square**	**F**	**Sig.**
Corrected Model	Declarative Task: Grade	9.561_a	6	1.594	2.104	0.135
	Programmatic Task: Grade	75.161_b	6	12.527	6.738	0.003
Programming experience in years	Declarative Task: Grade	9.561	6	1.594	2.104	0.135
	Programmatic Task: Grade	75.161	6	12.527	6.738	0.003

[a] R squared = 0.534 (Adjusted R Squared = 0.280), [b] R squared = 0.786 (Adjusted R Squared = 0.669)

directly improves the participants' performance in the programmatic task, but has no significant effect on their performance in the declarative task, leading to Finding 7:

Finding 7: There is a significant statistical difference in the performance in the programmatic task as a function of years of programming experience, but there is no significant statistical difference in the performance in the declarative task as a function of years of programming experience.

5 Discussion

The main goal of this thesis is to compare different components of developer experience in traditional and low-code development platforms. The Salesforce platform was used to conduct an experiment on the performance and experience of developers when performing a series of tasks using Salesforce Flow (a LCDP provided by Salesforce) and the APEX programming language (a traditional programming language supported by Salesforce) while having no prior experience with either tool. This section revisits the research questions originally posed in Section 1.2 and attempts to address each question based on the analysis performed in Section 4. This section also includes observations on a number of topics relevant to the subject, such as optimization and scalability, and addresses the limitations of the study.

5.1 Addressing the Research Questions

All four research questions involve a comparison between traditional (programmatic) and lowcode (declarative) development platforms. For RQ1, it appears that the developer experience depends greatly on the developer's prior experience with programming, LCDPs, and years of programming experience. The experiment results show that developers with more programming experience tended to prefer the programmatic tool over the declarative one and those with less programming experience tended to prefer the declarative tool, as described in Finding 1. This was visible in multiple attribute categories at different levels, as shown in Tables 5 and 6 and Figure 6. Developers with more programmatic experience clearly preferred coding over low-code development, with one experienced developer not considering the LCDP as programming at all.

RQ2 focuses on the **cognition** developer experience category. Looking at the tool itself, Finding 2 suggests that participants with more years of programming experience did not find either tool particularly novel, while participants with less experience found the declarative tool more novel. Finding 4 suggests that more experienced developers consider the programmatic tool more efficient, while those with less experience consider the declarative tool more efficient. This dichotomy in opinion is further elaborated in

Finding 5, which states the developers either find the programmatic tool efficient or the declarative one, but not both. Furthermore, as highlighted in Finding 6, developers with past LCDP experience find the declarative tool more efficient, implying that prior experience with declarative tools may help with learning how to work with new declarative tools more effectively in a shorter time period.

RQ3 focuses on the **affect** developer experience category. Finding 3 suggests that developers with more programming experience felt more in control of the programmatic tool, while those with less experience felt more in control of the declarative tool. Table 6 shows that the fewer years of experience a developer had, the more enjoyable they found the declarative tool, implying that developers with more traditional programming experience found using the declarative tool less satisfying. Finding 7 suggests that there is a significant statistical difference in the developer's performance using the programmatic tool as a function of years of programming experience, while there is no significant difference in the performance in the declarative task as a function of the same. This implies that while prior years of programming experience may improve the developer's performance with the programmatic tool, it may not directly result in better low-code performance. However, while the means calculated in Table 3 show that experienced developers in general found the tasks using both tools easier, the only statistically significant correlation found between the developer's performance and how they felt about each tool was that developers with better programmatic task performance felt that the declarative tool was less novel.

RQ4 focuses on the **conation** developer experience category. According to the interview data, multiple participants with varying years of programming experience considered the declarative tool as less scalable than the programmatic one in terms of the complexity of the programs that can be implemented using the tool, the deployment of declarative programs, and their maintainability. One experienced junior expressed concern over the inability to see the overall "code base" of the declarative tool and questioned whether changes to a declarative solution could ever be audited and reviewed using a distributed version control tool such as Git. A few participants considered the declarative tool as a prototyping tool, with one participant saying that they would eventually write their solutions in code after testing them in the declarative tool, implying that they consider their contribution more valuable in the programmatic tool compared to the declarative tool. One participant mentioned that they felt happier and more excited every time they succeeded in a task using the programmatic tool. However, no statistically significant correlation was found between developer satisfaction and their success in the hands-on challenges.

5.2 Further Observations

The issues of code efficiency and lack of control over the LCDP tool were mentioned by several participants in the experiment. One participant with several years of programming experience expressed concern over the quality of the code that is being generated by the tool based on the visual programming done by the user. LCDPs offer an unprecedentedly high level of abstraction. With the advent of high-level programming languages and rapid improvements in computer processing power and file storage options, software optimization is sometimes sacrificed to speed up software development speed or improve code readability. Knuth (1974) has argued that ”We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%”. As Pereira et al. (2017) have pointed out, a computer program may become faster by improving its source code, but also by optimizing its compiler and used libraries. As Knuth (1971) observed already during the reign of FORTRAN, for which the first ever optimizing compiler was released in 1957, “programmers should be strongly influenced by what their compilers do; a compiler writer ... may in fact know what is really good for the programmer and would like to steer him towards a proper course.” However, Knuth also points out that this may steer programmers towards making unnatural constructions to benefit from the compiler's optimization features. St-Amour et al. (2012), in their study of code optimization software that communicates with the programmers, have observed that expert programmers learn to reverse-engineer compiler logic and follow style guides (e.g. Fog (2012)) to write code that would be best optimized by the compiler, which can lead to unnatural constructions. Therefore, an efficient compiler is an important tool for writing efficient software in high-level programming languages and LCDPs create yet another level of abstraction between the programmer and the equipment they are ultimately programming for.

Optimization becomes even more important when the software needs to be scalable. LCDP adoption has been on the rise in the past decade. According to PMD Group's survey on rapid application development (2017), 74% of IT professionals considered low-code as a key to rapid application development while 56% expected the number of low-code solutions in enterprises to increase in the near future. Despite this rise in popularity and prevalence, scalability still remains a point of mistrust. According to Forrester (2019), even though 84% of enterprises have implemented a LCDP to handle part of their enterprise application development needs, 31% of enterprises still have not used low-code to build and deliver any high-value or critical application.

The experiment's results also indicated that participants with more development experience acknowledged the scalability and limitations of LCDP tools compared to programming languages. In order for LCDPs to truly become a viable alternative to traditional programming in high-level and high-complexity scenarios, the issue of scalability needs to be addressed by the platform developers. Furthermore, in order to gain the trust and support of experienced developers, more control needs to be given to the developers, so they would be able to study and modify the code that is being generated by the LCDP as needed. Lethbridge (2021) explains that most low-code tools currently in the market focus their attention on the experience of developers that create small amounts of code, while scalability remains a key requirement in many enterprise applications. Therefore, Lethbridge (2021) argues that LCDP designers should emphasize developer experience on all scales. A question for future research is posed here: what are the main causes of the apparent resistance from experienced developers and their cynicism towards the capabilities of LCDPs? To what extent are they influenced by the cognitive skills that the developers have built towards writing code as opposed to, for example, fear of change and concern for one's skills becoming obsolete in the future?

5.3 Limitations

A major limitation of this study and a direct threat to its external validity was the number of participants. Having only eighteen participants poses a direct threat to the validity of the findings and it is recommended to do similar experiments in the future with more participants. Furthermore, the small number of participants likely contributed to the inability to find more statistically significant correlations in the data and prevented the use of several types of regression models that require a larger sample size in order to be effective. Another limitation is the fact that the exact same task was used in both declarative and programmatic hands-on sessions. This was necessary in order to control the effect of the task difficulty and understandability on the participants' performance. However, this also meant that by the time of the programmatic hands-on challenge, the participants were already familiar with the task description from the declarative hands-on challenge that they had done earlier. This issue could have been avoided by splitting the participants into two groups, where one group would do the programmatic task first and the other would start with the declarative task. However, in that case, the small number of participants may have further skewed the results. Additionally, this study focuses solely on traditional software developers. While this does not necessarily constitute a sampling bias as the population under study consists of software developers, future studies can explore the differences in pro-code and low-code DX among citizen developers and other types of end users as well. Furthermore, a subset of the participants was recently hired employees of the same company, so their responses to some of the questions, such as the understandability and easiness of the tasks, may have been upwardly biased due to the Hawthorne effect, as they were aware that they were being tested.

The threats to the internal validity include the participant selection and the instrumentation. While the selection criteria ensured that none of the participants had prior Salesforce development experience and the amount of training the participants received for each tool prior to each hands-on challenge was controlled in the study, the nature and extent of their programming experience were not carefully controlled otherwise during participant selection. Therefore, the past programming experience had to be generalized into the years of experience and a few specific technology categories, namely multi-paradigm, object-oriented, and LCDP. Better control of the programmatic backgrounds of the participants may improve the validity of the study. Regarding the instrumentation, while the final questionnaires used in this study were not separately validated, several parts of the questionnaires had previously been validated elsewhere, namely as part of the UEQ (Laugwitz 2008) and the Survey for Measuring DX of LCDPs (Gao 2022).

Lastly, the scope of this study was limited, as it compares one programmatic tool with one LCDP. Future studies can compare several programmatic tools with several LCDPs in order to better control the effect of a single tool on experience items and evaluate generic correlations between programmatic versus declarative tools more reliably. Conducting a study using multiple tools may also provide additional insights into the differences in the learning curves of programmatic and declarative tools. Furthermore, using multiple tools would also allow the evaluation of the correlation between experience items and the tool itself, which is one aspect that this study was unable to address due to its limited scope.

6 Conclusion

This thesis explored the differences in developer experience among software developers when using low-code development platforms against traditional development platforms by devising and conducting an experiment. Salesforce Flow and the APEX programming language were used as example low-code and pro-code tools respectively and, after a training session on each tool, the experiment's participants were asked to develop a series of requirements using each tool and report on their thoughts and experiences afterward. The quantitative and qualitative data collected were then analyzed to identify patterns and correlations against different experiential categories as well as variables such as existing skills, past programming experience, and task performance.

The results show that prior programming experience both in terms of years and familiarity with different technologies appears to have a strong influence on the developers' performance, experiences, and tool preferences. Furthermore, the results indicate that, regardless of prior programming experience, most participants appeared skeptical of the scalability of LCDP tools, considering programmatic tools more scalable and more suitable for implementing complex requirements. While this study only focused on the episodic experience of Salesforce technologies and only involved participants with IT backgrounds, it can serve as a starting point for future research on the differences in developer experience in low-code and pro-code environments.

Future research can explore immediate and long-term experiences in addition to episodic experience, use a wider array of low-code and pro-code tools, and drill into specific developer experience sub-categories under conation, affect, and cognition. Further research may aid the development of more efficient and better-optimized low-code development platforms that require less training and have a larger and faster adoption rate among users in order to further facilitate the democratization of software development and support the ever-growing needs of digitalization.

7 References

- Accenture Techvision 2019 Tech Trends Report. https://www.accenture.com/_acnmedia/pdf-94/accenture-techvision-2019-tech-trends-report.pdf. Accessed 24.08.2022
- Baltes, S., Diehl, S. (2018). Towards a Theory of Software Development Expertise. Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018, 187–200.
- Besker, T., Ghanbari, H., Martini, A., Bosch, J. (2020). The influence of Technical Debt on software developer morale. Journal of Systems and Software, 167.
- Boynton, P. M., Greenhalgh, T. (2004). Selecting, designing, and developing your questionnaire. BMJ?: British Medical Journal, 328(7451), 1312–1315.
- Bratincevic, J. Koplowitz, R. (2017). The Forrester Wave™: Low-Code Development Platforms For Professional Developers, Q2 2021. Cambridge: Forrester. https://whitepaperseries.com/wp-content/uploads/2021/07/95830_TheForresterWave_LowCodeDevelopmentPlatformsForProfessionalDevelopersQ22021.pdf. Accessed 26.08.2022
- Burnett, M. M., McIntyre, D. W. (1995). Visual Programming. Encyclopedia of Electrical and Electronics Engineering (John G. Webster edition), John Wiley & Sons Inc., New York.
- Capretz, L. F. (2014). Bringing the human factor to software engineering. IEEE Software, 31 (2), 104–104.
- Cleo et al. (2019) "Understanding the Need for CRM ERP Systems Integration." Team Cleo, https://www.cleo.com/blog/knowledge-base-erp-and-crm-integration. Accessed 27.08.2022
- Codabux, Z., Williams, B. (2013). Managing technical debt: an industrial case study. In: Proceedings of the 4th International Workshop on Managing Technical Debt, pp. 8–15.

- Collin, J., Hiekkanen, K., Korhonen, J. J., Halén, M., Itälä, T., Helenius, M.(2015). IT Leadership in Transition - The Impact of Digitalization on Finnish Organizations. (eds.) IT Leadership in Transition, Aalto University. 29-34.

- CompTIA (2020). IT Industry Outlook. https://comptiacdn.azureedge.net/webcontent/docs/default-source/research-reports/comptia-it-industry-outlook-2020.pdf?sfvrsn=8869ad68_0. Accessed 23.08.2022

- Dahlberg, D. (2020). Developer Experience of a Low-Code Platform: An exploratory study (MSc Thesis, Umeå University).

- Dourish, P. (2003). The appropriation of interactive technologies: Some lessons from placeless documents. Computer Supported Cooperative Work, 12, 465–490

- Fagerholm, F. (2015). Software developer experience: Case studies in lean-agile and open source environments. (Doctoral dissertation, University of Helsinki).

- Fagerholm, F., Münch, J. (2012). Developer experience: Concept and definition. 2012 International Conference on Software and System Process (ICSSP), 73–77.

- Fog, A. (2012) Software optimization resources. http://www.agner.org/optimize/. Accessed 28.09.2022

- Forrester. (2019). Large Enterprises Succeeding with Low-Code, a Forrester Consulting Thought Leadership Paper Commissioned by Appian. 4–5.

- Gao, D. (2022). Measuring Developers' Episodic Experience of Low-Code Development Platforms. (MSc Thesis, Aalto University)

- Gartner (2016). Gartner IT glossary: digitalization. http://www.gartner.com/it-glossary/digitalization/. Accessed 22.08.2022

- Gillham B. (2000). Developing a questionnaire (real world research). London: Continuum

- Gomes, P., Brito, M. (2022). Low-Code Development Platforms: A Descriptive Study. 17th Iberian Conference on Information Systems and Technologies (CISTI). 1-4.

- Graziotin, D., Fagerholm, F., Wang, X., Abrahamsson, P. (2017a). Consequences of Unhappiness While Developing Software. 2017 IEEE/ACM 2nd International Workshop on Emotion Awareness in Software Engineering (SEmotion), 42–47.
- Graziotin, D., Fagerholm, F., Wang, X., Abrahamsson, P. (2017b). Unhappy Developers: Bad for Themselves, Bad for Process, and Bad for Software Product. 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), 362– 364.
- Graziotin, D., Wang, X., Abrahamsson, P. (2015). How do you feel, developer? An explanatory theory of the impact of affects on programming performance. PeerJ Computer Science, vol. 1, no. 1, p. E18.
- Green, T. R. G., Petre, M. (1996). Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. Journal of Visual Languages Computing, Volume 7, Issue 2, 131-174.
- Greig, J. (2020). COVID-19 triggering a massive shift in adoption of low-code platforms. TechRepublic. https://www.techrepublic.com/article/covid-19-triggering-a-massive-shift-in-adoption-of-low-code-platforms. Accessed 26.08.2022
- ISO - International Organization for Standardization. (2019). ISO 9241- 210 Ergonomics of human-system interaction – Part 210: Human-centred design for interactive systems.
- Khorram, F., Mottu, J. M., Sunyé, G. (2020). Challenges opportunities in low-code testing. Proceedings - 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS-C 2020 - Companion Proceedings, 490–499.
- Kinnula, A. (1999). Software Process Engineering in a Multi-Site Environment: An architectural design of a software process engineering system. (Doctoral dissertation, Acta Universitas Ouluensis - A333)
- Knuth, D. (1974). Structured Programming with Go-To Statements. ACM Computing Surveys. 6 (4): 268.

- Kruger, J., Dunning, D., (1999). Unskilled and unaware of it: how difficulties in recognizing one's own incompetence lead to inflated self-assessments. Journal of personality and social psychology 77, 1121–1134.

- Kuusinen, K. (2015). Software Developers as Users: Developer Experience of a Cross-Platform Integrated Development Environment. In P. Abrahamsson, L. Corral, M. Oivo, B. Russo (Eds.), Product-Focused Software Process Improvement (Vol. 9459, pp. 546–552). Springer International Publishing.

- Kuusinen, K., Petrie, H., Fagerholm, F., Mikkonen, T. (2016). Flow, Intrinsic Motivation, and Developer Experience in Software Engineering. In H. Sharp T. Hall (Eds.), Agile Processes, in Software Engineering, and Extreme Programming (Vol. 251, pp. 104–117). Springer International Publishing.

- Laugwitz, B., Held, T., Schrepp, M. (2008). Construction and evaluation of a user experience questionnaire. A. Holzinger (Ed.), HCI and Usability for Education and Work, Vol. 5298 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, Berlin, Heidelberg, 63–76.

- Law, E. L. C., Roto, V., Hassenzahl, M., Vermeeren, A. P., Kort, J. (2009). Understanding, scoping and defining user experience: a survey approach. In Proceedings of the SIGCHI conference on human factors in computing systems (pp. 719-728).

- Lethbridge, T. C. (2021) Low-Code Is Often High-Code, So We Must Design Low-Code Platforms to Enable Proper Software Engineering. Leveraging Applications of Formal Methods, Verification and Validation. ISoLA 2021. Lecture Notes in Computer Science, vol 13036. Springer, Cham. 202–212.

- Lieberman, H., Paternò, F., Klann, M., Wulf, V. (2006). End-User Development: An Emerging Paradigm. In: Lieberman, H., Paternò, F., Wulf, V. (eds) End User Development. Human-Computer Interaction Series, vol 9. Springer, Dordrecht, 1-8.

- Morales, J., Rusu, C., Botella, F., Quinones, D. (2019). Programmer experience: A systematic literature review. IEEE Access, vol. 7, 71079-71094.

- Mohapatra, S. Singh, R (2012). Information strategy design and practices. Springer, Chapter 1, 1-23.
- Nylund, A. (2020). A multivocal literature review on developer experience. (Master's thesis, Aalto University)
- Oliveira, R., França, C. (2019). Agile Practices and Motivation: A quantitative study with Brazilian software developers. Proceedings of the Evaluation and Assessment on Software Engineering, 365–368.
- Oppenheim AN. Questionnaire design, interviewing and attitude measure- ment. London: Continuum, 1992.
- Pereira, R., Carção, T., Couto, M., Cunha, J., Fernandes, J. P., and Saraiva, J. (2017). Helping Programmers Improve the Energy Efficiency of Source Code. In Proc. of the 39th Int. Conf. on Soft. Eng. Companion. ACM.
- Peters, L. (2014). Technical debt: the ultimate antipattern - The Biggest costs may be hidden, widespread, and long term. In: Managing Technical Debt (MTD)
- PMG Group. (2017). IT and the Rise of the RAD Enterprise Survey. https://www.pmg.net/wp-content/uploads/2019/07/rad.jpg. Accessed 21.10.2022.
- Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018, 187–200.
- Richardson, C, Rymer, J. R. (2014). New Development Platforms Emerge For Customer-Facing Applications; Forrester: Cambridge, MA, USA.
- Roto, V., Lee, J., Law, L. L., and Zimmerman, J. (2021). The Overlaps and Boundaries Between Service Design and User Experience Design. In Designing Interactive Systems Conference 2021, 1915–26. New York, NY, USA: Association for Computing Machinery.
- Sahay, A., Indamutsa, A., Di Ruscio, D., and Pierantonio, A. (2020). Supporting the understanding and comparison of low-code development platforms.

- Salesforce Developers. (2010). What Is Apex?: Apex Developer Guide: Salesforce Developers, http://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apex code/apex_intro_what_is_apex.htm. Accessed 14.08.2022

- Salesforce Trailhead. (2022). Guide Users Through Your Business Processes with Flow Builder. https://trailhead.salesforce.com/content/learn/modules/business_pro cess_automation/flow. Accessed 28.08.2022

- SHRM (2019). The Global Skills Shortage - Bridging the Talent Gap with Education, Training, and Sourcing. https://www.shrm.org/hr-today/trends-and-forecasting/research-and-surveys/Documents/SHRM%20Skills%20Gap%202019.pdf. Accessed 23.08.2022

- Spínola, R. O., Vetrò, V., Zazworka, N., Seamna, C., Shull, F. (2013). Investigating technical debt folklore: shedding some light on technical debt opinion. 4th International Workshop on Managing Technical Debt (MTD), 1–7

- St-Amour, V., Tobin-Hochstadt, S., Felleisen, M. (2012). Optimization Coaching: Optimizers Learn to Communicate with Programmers. In Proc. of the ACM Int. Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA '12). ACM, 163–166

- Tanimoto, S. (1990). VIVA: a visual language for image processing. Journal of Visual Languages Computing 2(2), 127-139.

- Tchounikine, P. (2017). Designing for Appropriation: A Theoretical Account. Human-Computer Interaction, 32(4), 155–195.

- Tisi, M., Mottu, J. M., Kolovos, D., De Lara, J., Guerra, E., Di Ruscio, D., Pierantonio, A., Wimmer, M. (2019). Lowcomote: Training the Next Generation of Experts in Scalable Low-Code Engineering Platforms. STAF 2019 Co-Located Events Joint Proceedings: 1st Junior Researcher Community Event, 2nd International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems, and 1st Research Project Showcase Workshop co-located with Software Technologies: Applications and Foundations, Eindhoven, Netherlands.

- Tom, E., Aurum, A., Vidgen, R. (2013). An exploration of technical debt. Journal of Systems and Software, 86 (6), 1498–1516.
- Turban, E. Volonino, L. (2011) Information Technology for Management: Improving Strategic and Operational Performance, 8th Edition. (O'Reilly) Chapters 9 - 11
- Urbach, N., Ahlemann, F. (2018). IT management in the digital age—a roadmap for the IT department of the future. Springer, Heidelberg
- Urbach, N., Drews, P., Ross, J. (2017). Digital business transformation and the changing role of the IT function. MIS Q Exec 16(2):ii–iv
- Urbach, N., Ahlemann, F., Böhmann, T., Drews, P., Brenner, W., Schaudel, F., and Schütte, R. (2018). The Impact of Digitalization on the IT Department. Business Information Systems Engineering.
- Wilburn, Kathleen M., Wilburn, H. Ralph. (2018). The Impact of Technology on Business and Society. Global Journal of Business Research, v. 12 (1), 23-39.
- Wong, J., Iijima, K., Leow, A., Jain, A., Vincent, P. (2021). Magic quadrant for enterprise low-code application platforms; Gartner Reports: Stamford, CT, USA, https://www.gartner.com/doc/reprints?id=1-275QSBDLct=210813st=sb. Accessed 20.10.2022
- Woo, M. (2020) The Rise of No/Low Code Software Development-No Experience Needed?. Journal of Engineering (Beijing).
- Yli-Huumo, J., Maglyas, A., Smolander, K. (2014). The sources and approaches to management of technical debt: a case study of two product lines in a middle-size Finnish software company. In: Conference on Product-Focused Software Process Improvement, 93–107.

A Hands-on Task Description

Whenever a new **Opportunity** record is created or updated such that the value of the **Amount** field is greater than **10000**, the following actions need to automatically happen:

1. A Task record should be created and assigned to the **Opportunity's Account owner** with the following details:
   - Subject = New Major Opportunity
   - Description = A major opportunity with the amount of ***Opportunity's Amount*** was added on ***Opportunity's Last Modified Date***. Please follow up with the branch executive.
   - Priority = High
   - Related Record (WhatId) = ***Opportunity's Id***
2. The Opportunity record itself should be updated such that:
   - Next Step = Account Executive to contact Branch Executive
3. An email should be automatically sent from the ***Opportunity's owner's email*** to the ***Opportunity's Account's owner's email***. Its subject line and body should say "Major opportunity project incoming!"

In 60 minutes, create the automations using only the tool allowed. Note that the solution should be able to handle large amounts of data as well.

B Pre-Experiment Questionnaire

Pre-study questionnaire

Please answer the following questions about your own background and estimate of your own knowledge and skills at this moment.

* Required

1. Your name *

2. How many years of professional programming experience do you have? (please write in numbers) *

3. How familiar are you with programming? *

Mark only one oval.

	1	2	3	4	5	6	7	
Not familiar at all	◯	◯	◯	◯	◯	◯	◯	Very familiar

4. How familiar are you with low-code programming? (meaning creating logic through a graphical user interface instead of via writing code) *

Mark only one oval.

	1	2	3	4	5	6	7	
Not familiar at all	◯	◯	◯	◯	◯	◯	◯	Very familiar

How familiar are you with each of the following technologies?

5. Python: *

Mark only one oval.

	1	2	3	4	5	6	7	
Not familiar at all	⬭	⬭	⬭	⬭	⬭	⬭	⬭	Very familiar

6. JavaScript: *

Mark only one oval.

	1	2	3	4	5	6	7	
Not familiar at all	⬭	⬭	⬭	⬭	⬭	⬭	⬭	Very familiar

7. Scratch *

Mark only one oval.

	1	2	3	4	5	6	7	
Not familiar at all	⬭	⬭	⬭	⬭	⬭	⬭	⬭	Very familiar

8. Java *

Mark only one oval.

	1	2	3	4	5	6	7	
Not familiar at all	⬭	⬭	⬭	⬭	⬭	⬭	⬭	Very familiar

9. C++ *

Mark only one oval.

	1	2	3	4	5	6	7	
Not familiar at all								Very familiar

10. PHP *

Mark only one oval.

	1	2	3	4	5	6	7	
Not familiar at all								Very familiar

11. Salesforce Flow Builder *

Mark only one oval.

	1	2	3	4	5	6	7	
Not familiar at all								Very familiar

12. Salesforce Process Builder *

Mark only one oval.

	1	2	3	4	5	6	7	
Not familiar at all								Very familiar

13. APEX *

Mark only one oval.

	1	2	3	4	5	6	7	
Not familiar at all								Very familiar

14. Is there any other technology/language you are familiar with?

15. What would you consider as your strongest programming language? *

16. For how many years have you been using Salesforce products? *

C Post-Challenge Questionnaire

Hands-on Challenge - Post-session feedback

Please select the most appropriate descriptions of the experience you just had during the hands-on challenge.

* Required

1. *

Mark only one oval.

	1	2	3	4	5	6	7	
Exhausting	⬭	⬭	⬭	⬭	⬭	⬭	⬭	Smooth

2. *

Mark only one oval.

	1	2	3	4	5	6	7	
Concentrated	⬭	⬭	⬭	⬭	⬭	⬭	⬭	Distracted

3. *

Mark only one oval.

	1	2	3	4	5	6	7	
Lack of control	⬭	⬭	⬭	⬭	⬭	⬭	⬭	In control

4. *

Mark only one oval.

	1	2	3	4	5	6	7	
Limited	⬭	⬭	⬭	⬭	⬭	⬭	⬭	Free to explore

5. *

Mark only one oval.

	1	2	3	4	5	6	7	
Unproductive	◯	◯	◯	◯	◯	◯	◯	Productive

6. *

Mark only one oval.

	1	2	3	4	5	6	7	
Annoying	◯	◯	◯	◯	◯	◯	◯	Enjoyable

7. *

Mark only one oval.

	1	2	3	4	5	6	7	
Not understandable	◯	◯	◯	◯	◯	◯	◯	Understandable

8. *

Mark only one oval.

	1	2	3	4	5	6	7	
Creative	◯	◯	◯	◯	◯	◯	◯	Dull

9. *

Mark only one oval.

	1	2	3	4	5	6	7	
Easy to learn	◯	◯	◯	◯	◯	◯	◯	Difficult to learn

10. *

Mark only one oval.

	1	2	3	4	5	6	7	
Valuable	◯	◯	◯	◯	◯	◯	◯	Inferior

11. *

Mark only one oval.

	1	2	3	4	5	6	7	
Boring	◯	◯	◯	◯	◯	◯	◯	Exciting

12. *

Mark only one oval.

	1	2	3	4	5	6	7	
Not interesting	◯	◯	◯	◯	◯	◯	◯	Interesting

13. *

Mark only one oval.

	1	2	3	4	5	6	7	
Unpredictable	◯	◯	◯	◯	◯	◯	◯	Predictable

14. *

Mark only one oval.

	1	2	3	4	5	6	7	
Fast	◯	◯	◯	◯	◯	◯	◯	Slow

15. *

Mark only one oval.

	1	2	3	4	5	6	7	
Inventive	⬭	⬭	⬭	⬭	⬭	⬭	⬭	Conventional

16. *

Mark only one oval.

	1	2	3	4	5	6	7	
Obstructive	⬭	⬭	⬭	⬭	⬭	⬭	⬭	Supportive

17. *

Mark only one oval.

	1	2	3	4	5	6	7	
Good	⬭	⬭	⬭	⬭	⬭	⬭	⬭	Bad

18. *

Mark only one oval.

	1	2	3	4	5	6	7	
Complicated	⬭	⬭	⬭	⬭	⬭	⬭	⬭	Easy

19. *

Mark only one oval.

	1	2	3	4	5	6	7	
Unlikable	⬭	⬭	⬭	⬭	⬭	⬭	⬭	Pleasing

20. *

Mark only one oval.

	1	2	3	4	5	6	7	
Usual	⬭	⬭	⬭	⬭	⬭	⬭	⬭	Leading edge

21. *

Mark only one oval.

	1	2	3	4	5	6	7	
Unpleasant	⬭	⬭	⬭	⬭	⬭	⬭	⬭	Pleasant

22. *

Mark only one oval.

	1	2	3	4	5	6	7	
Secure	⬭	⬭	⬭	⬭	⬭	⬭	⬭	Not secure

23. *

Mark only one oval.

	1	2	3	4	5	6	7	
Motivating	⬭	⬭	⬭	⬭	⬭	⬭	⬭	Demotivating

24. *

Mark only one oval.

	1	2	3	4	5	6	7	
Meets expectations	⬭	⬭	⬭	⬭	⬭	⬭	⬭	Does not meet expectations

25. *

Mark only one oval.

	1	2	3	4	5	6	7	
Inefficient								Efficient

26. *

Mark only one oval.

	1	2	3	4	5	6	7	
Clear								Confusing

27. *

Mark only one oval.

	1	2	3	4	5	6	7	
Impractical								Practical

28. *

Mark only one oval.

	1	2	3	4	5	6	7	
Organized								Cluttered

29. *

Mark only one oval.

	1	2	3	4	5	6	7	
Attractive								Unattractive

30. *

Mark only one oval.

	1	2	3	4	5	6	7	
Friendly	⬭	⬭	⬭	⬭	⬭	⬭	⬭	Unfriendly

31. *

Mark only one oval.

	1	2	3	4	5	6	7	
Conservative	⬭	⬭	⬭	⬭	⬭	⬭	⬭	Innovative

How much do you agree with the following statements regarding **the task**?

32. The task was easy to understand *

Mark only one oval.

	1	2	3	4	5	6	7	
Completely agree	⬭	⬭	⬭	⬭	⬭	⬭	⬭	Completely disagree

33. The task was easy to complete *

Mark only one oval.

	1	2	3	4	5	6	7	
Completely agree	⬭	⬭	⬭	⬭	⬭	⬭	⬭	Completely disagree

D Task Grading Criteria

D.1 Test Case 1: New Opportunity With Amount Greater Than 10000

D.1.1 Steps

1. Go to the Salesforce Account tab.
2. Click on *New Account*.
3. Fill the form with the following details:
   - Name: Test Account
4. Click on "Save".
5. Go to the Salesforce Opportunity tab.
6. Click on *New Opportunity*.
7. Fill the form with the following details:
   - Name: Major Project Opportunity
   - Amount: 15000
   - Owner: *any active user except the current user*
8. Click on "Save".

D.1.2 Expected Results

- A new Opportunity is created. The user is redirected to the new Opportunity page. **(1 point)**
- A Task is created under the Opportunity with the following details:
  - Subject: New Major Opportunity
  - Description: A major opportunity with the amount of 15000 was added on *today's date*. Please follow up with the branch executive.
  - Priority: High
  - Related Record (WhatId): *Opportunity id from step 8* **(1 point)**

- The Opportunity record contains the following details:
  - Name: Major Project Opportunity
  - Amount: 15000
  - Next Step: Account Executive to contact Branch Executive
  - Owner: *any active user except the current user* **(1 point)**
- An email is sent from the *Opportunity's owner's email* to the tester's email address. Its subject line and body read "Major opportunity project incoming!" **(1 point)**

D.2 Test Case 2: New Opportunity With Amount Less Than 10000

D.2.1 Steps

1. Go to the Salesforce Account tab.
2. Click on *New Account*.
3. Fill the form with the following details:
   - Name: Test Account
4. Click on "Save".
5. Go to the Salesforce Opportunity tab.
6. Click on *New Opportunity*.
7. Fill the form with the following details:
   - Name: Major Project Opportunity
   - Amount: 5000
   - Owner: *any active user except the current user*
8. Click on "Save".

D.2.2 Expected Results

- A new Opportunity is created. The user is redirected to the new Opportunity page. The ”Next Step” field on the Opportunity is empty. **(1 point)**
- No task is created under the Opportunity. **(1 point)**
- No email is sent to the tester’s email address from Salesforce. **(1 point)**

D.3 Test Case 3: Updating Existing Opportunity to Amount Greater Than 10000

D.3.1 Steps

1. Go to the record page of the Opportunity created in Test Case 2 (Appendix D.2.1).
2. Click on *Edit*.
3. Change the value of Amount to 12000.
4. Click on ”Save”.

D.3.2 Expected results

- The Opportunity is updated. The user is redirected to the Opportunity page. The ”Next Step” field now reads ”Account Executive to contact Branch Executive”. **(1 point)**
- A Task is created under the Opportunity with the following details:
  - Subject: New Major Opportunity
  - Description: A major opportunity with the amount of 12000 was added on *today’s date*. Please follow up with the branch executive.
  - Priority: High
  - Related Record (WhatId): *Opportunity id from step 4* **(1 point)**

- An email is sent from the *Opportunity's owner's email* to the tester's email address. Its subject line and body read ”Major opportunity project incoming!” **(1 point)**

E Descriptive Questions From the Post-challenge Questionnaire and Their Attribute Categories

Attribute category	**Negative term**	**Positive term**
Attractiveness	Annoying	Enjoyable
	Bad	Good
	Unlikable	Pleasing
	Unpleasant	Pleasant
	Unattractive	Attractive
	Unfriendly	Friendly
Dependability	Unpredictable	Predictable
	Obstructive	Supportive
	Not Secure	Secure
	No Expectations	Meets Expectations
	Lack of control	In control
	Limited	Free to Explore
Efficiency	Slow	Fast
	Inefficient	Efficient
	Impractical	Practical
	Cluttered	Organized
	Distracted	Concentrated
	Unproductive	Productive
Novelty	Dull	Creative
	Conventional	Inventive
	Usual	Leading Edge
	Conservative	Innovative
Perspicuity	Not Understandable	Understandable
	Difficult to Learn	Easy to Learn
	Complicated	Easy
	Confusing	Clear
Stimulation	Inferior	Valuable
	Boring	Exciting
	Not Interesting	Interesting
	Demotivating	Motivating
	Exhausting	Smooth

F Grouped Bivariate Correlation Table

Correlations

		Past programming experience in years	Multiparadigm_XP	ObjectOriented_XP	LCDP_XP	DEC_Attractiveness	PRO_Attractiveness	DEC_Dependability	PRO_Dependability	DEC_Efficiency	PRO_Efficiency	DEC_Novelty	PRO_Novelty	DEC_Perspicuity	PRO_Perspicuity	DEC_Stimulation	PRO_Stimulation
Past programming experience in years	Pearson Correlation	1	-.196	-.158	-.435	-.738**	.626**	-.744**	.702**	-.944**	.760**	-.779**	.168	-.551*	.628**	-.844**	.698**
	Sig. (2-tailed)		.436	.531	.071	<.001	.005	<.001	.001	<.001	<.001	<.001	.505	.018	.005	<.001	.001
	N	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
Multiparadigm_XP	Pearson Correlation	-.196	1	.436	.655**	.038	.053	.161	-.004	.316	-.236	.045	.159	-.063	.078	.207	-.211
	Sig. (2-tailed)	.436		.071	.003	.881	.834	.523	.987	.202	.346	.859	.529	.803	.760	.410	.401
	N	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
ObjectOriented_XP	Pearson Correlation	-.158	.436	1	.207	.076	.041	-.110	.019	.106	-.168	-.210	.335	.241	.186	.424	-.036
	Sig. (2-tailed)	.531	.071		.409	.764	.871	.665	.941	.676	.506	.404	.175	.335	.459	.079	.888
	N	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
LCDP_XP	Pearson Correlation	-.435	.655**	.207	1	.263	-.385	.398	-.448	.620**	-.521*	.433	-.258	.141	-.369	.398	-.426
	Sig. (2-tailed)	.071	.003	.409		.291	.114	.102	.062	.006	.027	.073	.302	.577	.132	.102	.078
	N	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
DEC_Attractiveness	Pearson Correlation	-.738**	.038	.076	.263	1	-.090	.753**	-.220	.727**	-.310	.633**	.276	.836**	-.129	.768**	-.126
	Sig. (2-tailed)	<.001	.881	.764	.291		.722	<.001	.380	<.001	.211	.005	.267	<.001	.610	<.001	.618
	N	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
PRO_Attractiveness	Pearson Correlation	.626**	.053	.041	-.385	-.090	1	-.215	.958**	-.598**	.872**	-.511*	.775**	-.024	.937**	-.382	.907**
	Sig. (2-tailed)	.005	.834	.871	.114	.722		.392	<.001	.009	<.001	.030	<.001	.923	<.001	.117	<.001
	N	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
DEC_Dependability	Pearson Correlation	-.744**	.161	-.110	.398	.753**	-.215	1	-.263	.807**	-.350	.755**	-.020	.464	-.261	.661**	-.333
	Sig. (2-tailed)	<.001	.523	.665	.102	<.001	.392		.291	<.001	.154	<.001	.938	.053	.296	.003	.177
	N	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
PRO_Dependability	Pearson Correlation	.702**	-.004	.019	-.448	-.220	.958**	-.263	1	-.690**	.917**	-.571*	.633**	-.185	.912**	-.452	.896**
	Sig. (2-tailed)	.001	.987	.941	.062	.380	<.001	.291		.002	<.001	.013	.005	.462	<.001	.060	<.001
	N	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
DEC_Efficiency	Pearson Correlation	-.944**	.316	.106	.620**	.727**	-.598**	.807**	-.690**	1	-.780**	.814**	-.185	.530*	-.607**	.814**	-.699**
	Sig. (2-tailed)	<.001	.202	.676	.006	<.001	.009	<.001	.002		<.001	<.001	.463	.024	.008	<.001	.001
	N	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
PRO_Efficiency	Pearson Correlation	.760**	-.236	-.168	-.521*	-.310	.872**	-.350	.917**	-.780**	1	-.522*	.529*	-.213	.794**	-.569*	.935**
	Sig. (2-tailed)	<.001	.346	.506	.027	.211	<.001	.154	<.001	<.001		.026	.024	.396	<.001	.014	<.001
	N	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
DEC_Novelty	Pearson Correlation	-.779**	.045	-.210	.433	.633**	-.511*	.755**	-.571*	.814**	-.522*	1	-.192	.526*	-.546*	.547*	-.547*
	Sig. (2-tailed)	<.001	.859	.404	.073	.005	.030	<.001	.013	<.001	.026		.445	.025	.019	.019	.019
	N	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
PRO_Novelty	Pearson Correlation	.168	.159	.335	-.258	.276	.775**	-.020	.633**	-.185	.529*	-.192	1	.425	.679**	.133	.630**
	Sig. (2-tailed)	.505	.529	.175	.302	.267	<.001	.938	.005	.463	.024	.445		.079	.002	.598	.005
	N	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
DEC_Perspicuity	Pearson Correlation	-.551*	-.063	.241	.141	.836**	-.024	.464	-.185	.530*	-.213	.526*	.425	1	.024	.641**	-.003
	Sig. (2-tailed)	.018	.803	.335	.577	<.001	.923	.053	.462	.024	.396	.025	.079		.924	.004	.990
	N	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
PRO_Perspicuity	Pearson Correlation	.628**	.078	.186	-.369	-.129	.937**	-.261	.912**	-.607**	.794**	-.546*	.679**	.024	1	-.413	.850**
	Sig. (2-tailed)	.005	.760	.459	.132	.610	<.001	.296	<.001	.008	<.001	.019	.002	.924		.088	<.001
	N	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
DEC_Stimulation	Pearson Correlation	-.844**	.207	.424	.398	.768**	-.382	.661**	-.452	.814**	-.569*	.547*	.133	.641**	-.413	1	-.442
	Sig. (2-tailed)	<.001	.410	.079	.102	<.001	.117	.003	.060	<.001	.014	.019	.598	.004	.088		.066
	N	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
PRO_Stimulation	Pearson Correlation	.698**	-.211	-.036	-.426	-.126	.907**	-.333	.896**	-.699**	.935**	-.547*	.630**	-.003	.850**	-.442	1
	Sig. (2-tailed)	.001	.401	.888	.078	.618	<.001	.177	<.001	.001	<.001	.019	.005	.990	<.001	.066	
	N	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18

**. Correlation is significant at the 0.01 level (2-tailed).

*. Correlation is significant at the 0.05 level (2-tailed).