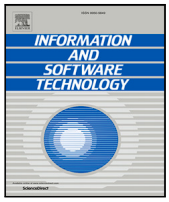
Information and Software Technology 139 (2021) 106640

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



A decision model for programming language ecosystem selection: Seven industry case studies

Siamak Farshidi [a,*], Slinger Jansen [b,c], Mahdi Deldar [d]

[a] Informatics Institute, University of Amsterdam, Amsterdam, The Netherlands
[b] Department of Information and Computer Science at Utrecht University, Utrecht, The Netherlands
[c] Department of Computer Science at Lappeenranta University of Technology, Lappeenranta, Finland
[d] Data Kavosh Software, Tehran, Iran

ARTICLE INFO

Keywords:
Programming language ecosystem selection
Decision model
Industry case study
Software production
Multi-criteria decision-making
Decision support system

ABSTRACT

Context: Software development is a continuous decision-making process that mainly relies on the software engineer's experience and intuition. One of the essential decisions in the early stages of the process is selecting the best fitting programming language ecosystem based on the project requirements. A significant number of criteria, such as developer availability and consistent documentation, in addition to the number of available options in the market, lead to a challenging decision-making process. As the selection of programming language ecosystems depends on the application to be developed and its environment, a decision model is required to analyze the selection problem using systematic identification and evaluation of potential alternatives for a development project.

Method: Recently, we introduced a framework to build decision models for technology selection problems in software production. Furthermore, we designed and implemented a decision support system that uses such decision models to support software engineers with their decision-making problems. This study presents a decision model based on the framework for the programming language ecosystem selection problem.

Results: The decision model has been evaluated through seven real-world case studies at seven software development companies. The case study participants declared that the approach provides significantly more insight into the programming language ecosystem selection process and decreases the decision-making process's time and cost.

Conclusion: With the decision model, software engineers can more rapidly evaluate and select programming language ecosystems. Having the knowledge in the decision model readily available supports software engineers in making more efficient and effective decisions that meet their requirements and priorities. Furthermore, such reusable knowledge can be employed by other researchers to develop new concepts and solutions for future challenges.

1. Introduction

Software engineers make a sequence of design decisions while developing a software product [1]. Each design decision can be analyzed as an episode of complex problem solving [2] that relies on a substantial amount of knowledge and rationale. Design decisions in the software development lifecycle are significantly constrained by former decisions and lead to additional constraints on future decisions [3]. Making informed design decisions in different phases of the software development lifecycle has critical impacts on the success of a software product.

Over the last decades, thousands of programming languages belonging to several programming paradigms have been introduced. Despite the significant number of programming languages, only a few fundamental programming concepts and languages have survived for more than ten years [4]. Some languages have grown into extensive software ecosystems [5], while others have failed to grow beyond their niche or disappeared altogether [6].

No unique programming language is the *best option* for all potential scenarios. Judging the suitability of a programming language for a software product, as an application or a customized component, is a non-trivial task. For instance, a purely functional language like *Haskell* is the best fit for writing parallel programs that can, in principle, efficiently exploit huge parallel machines working on large data sets [7]. However, while developing a dynamic website, a software engineer

* Corresponding author.
E-mail addresses: s.farshidi@uva.nl (S. Farshidi), slinger.jansen@uu.nl (S. Jansen), m.deldar@datakavosh.com (M. Deldar).

https://doi.org/10.1016/j.infsof.2021.106640
Received 14 May 2020; Received in revised form 5 May 2021; Accepted 18 May 2021
Available online 27 May 2021

might consider *ASP.net* as the best alternative, and others might prefer using *PHP* or a similar scripting language. It is interesting to highlight that successful projects have been built with both: StackOverflow is built-in *ASP.net*, whereas Wikipedia is built-in *PHP*. Furthermore, a software engineer might prefer particular criteria, such as scalability in enterprise applications, whereas other criteria, such as technology maturity level, might have lower priorities.

Selecting and employing multiple programming languages in one development project is quite common [8]. For example, a software engineer might code the back-end of a website using *PHP* or *C#* and then use the combination of *HTML*, *CSS*, and *JavaScript* to design layouts and styles of the front-end. Furthermore, leading popular software available in the market are developed in multiple languages. For example, some essential components of the Linux operating system[1] are designed and implemented in *C*; however, the majority of its utilities and applications are built-in *C++*, *Perl*, and *Python*. Comparably, OpenCV4,[2] an open-source computer vision and machine learning software library, is developed utilizing an assemblage of programming languages such as *C++*, *C*, *Python*, *Java*, and *JavaScript*.

Acquiring and expanding knowledge about programming languages is a highly complex process, as significant numbers of criteria and alternatives exist in the market [9]. Various factors need to be taken into account, of which not all are obvious. Simultaneously, the choice of programming languages can have repercussions on the implementation cost, quality of the result, and maintenance cost of the application [10]. Some of these consequences may not be felt for years after the initial programming language choice decision has been made.

Each programming language has different characteristics, communities, and ecosystems that should be considered. The selection process is mainly based on surrounding ecosystems and communities. Third-party libraries play an essential role as many software applications are built by gluing together plenty of existing libraries in the market, so such libraries increase language growth. Additionally, communities generate wikis, forums, and tutorials to improve the learnability and understandability of languages.

Nowadays, the development of software products, systems, and services typically results in complex decision models and decision-making processes [11]. Selecting the best fitting programming language ecosystem(s) for a software project can be modeled as a multi-criteria decision-making (MCDM) problem that deals with the evaluation of a set of alternatives and takes into account a set of decision criteria [12] (e.g., features of the programming language ecosystems). In this study, we focus on *programming language ecosystems*, and for the sake of brevity, we use *programming languages* to refer to them.

Recently, we developed a theoretical framework [13] to assist software engineers with a set of MCDM problems in software production. The framework provides a guideline for software engineers to systematically capture knowledge from different knowledge sources to build decision models for MCDM problems in software production. Knowledge has to be collected and organized when it is needed to be employed. The framework and a Decision Support System (DSS) [14,15] were introduced in our previous studies for building MCDM decision models in software production [13,16–19].

The DSS is a platform[3] for capturing MCDM decision models based on the framework. Decision models can be uploaded to the DSS's knowledge base to facilitate software-producing organizations' decision-making process according to their requirements and preferences. The DSS provides a discussion and negotiation platform to enable decision-makers at software-producing organizations to make group decisions. Furthermore, the DSS can be used over the entire lifecycle and co-evolve its advice based on evolving requirements. A broad study has been carried out based on qualitative and quantitative research to evaluate the efficiency and effectiveness of the DSS and the decision models inside its knowledge base to support software engineers with their decision-making process in software production.

Knowledge about programming languages is scattered among a wide range of literature, documentation, and software engineers' experience. This study's main motive is to build a decision model to capture knowledge about programming languages and concepts systematically and make it available in a reusable and extendable format. Accordingly, we have followed our framework [13] to build such a decision model for the programming language selection problem.

The rest of this study is outlined as follows: Section 2 describes our research method, which is based on design science and exploratory theory-testing case studies. This study has the following contributions:

- Section 3 explains the integration of the captured tacit knowledge of software engineers through interviews and the explicit knowledge that is scattered in an extensive list of websites, articles, and reports. Acquired knowledge is presented in the form of reusable knowledge that software engineers in their decision-making process can use.
- Section 4 describes seven conducted case studies, performed in the Netherlands and Iran, to evaluate the effectiveness and usefulness of the decision model.
- Section 5 analyzes the results of the DSS and compares them with the case study participants' ranked shortlists of feasible programming languages. The results show that the DSS recommended nearly the same solutions as the case study participants suggested to their companies after extensive analysis and discussions and do so more efficiently.

Section 6 highlights barriers to the knowledge acquisition and decision-making process, such as motivational and cognitive biases, and argues how we have minimized these threats to the validity of the results. Section 7 positions the proposed approach in this study among the other programming language selection techniques in the literature. Finally, Section 8 summarizes the proposed approach, defends its novelty, and offers directions for future studies.

2. Research method

Research methods are classified based on their data collection techniques (interview, observation, literature, etc.), inference techniques (taxonomy, protocol analysis, statistics, etc.), research purpose (evaluation, exploration, description, etc.), units of analysis (individuals, groups, process, etc.), and so forth [20]. Multiple research methods can be combined to achieve a fuller picture and a more in-depth understanding of the studied phenomenon by connecting complementary findings that conclude from the methods from the different methodological traditions of qualitative and quantitative investigation [21].

Knowledge acquisition is the process of capturing, structuring, and organizing knowledge from multiple sources [22]. Human experts, discourse, internal meetings, case studies, literature studies, or other research methods are the primary sources of knowledge. The rest of this section outlines the research questions and elaborates on a mixed research method based on design science research, expert interviews, documentation analysis, and case study research to capture knowledge regarding the programming languages, to answer the research questions, and to build a decision model for the programming language selection problem.

2.1. Research questions

We formulated the following research questions, each one with the goal of capturing the knowledge required for creating a decision model for programming language selection [13]:

[1] https://www.linux.com/

[2] https://opencv.org/opencv-4-0/

[3] The decision studio is available online on the DSS website: https://dss-mcdm.com.

RQ_1: Which programming languages should be considered in the decision model?
RQ_2: Which programming concepts should be considered as the programming language features in the decision model?
RQ_3: Which software quality attributes can be utilized to evaluate the programming languages?
RQ_4: What are the impacts of the programming language features on the quality attributes of the programming languages?
RQ_5: Which programming languages currently support the programming language features?

2.2. Design science

Design Science is an iterative process [23], has its roots in engineering [24], is broadly considered a problem-solving process [25], and attempts to produce generalizable knowledge about design processes and design decisions. Similar to a theory, the design process is a set of hypotheses that can eventually be proven only by creating the artifact it describes [26]. However, a design's feasibility can be supported by a scientific theory to the extent that the design comprises principles of the theory.

Recently, we designed a framework [13] and implemented a DSS [14,15] for supporting software engineers (decision-makers) with their MCDM problems in software production. Knowledge engineering theories have been employed to design and implement the DSS and the framework. The framework provides a guideline for decision-makers to build decision models for MCDM problems in software production following the six-step of the decision-making process [27]: (1) identifying the objective, (2) selection of the features, (3) selection of the alternatives, (4) selection of the weighing method, (5) applying the method of aggregation, and (6) decision-making based on the aggregation results.

In this study, we applied the framework to build a decision model for the programming language selection problem. The research approach for creating the decision model is Design Science, which addresses research through the building and evaluation of artifacts to meet identified business needs [24]. We carried out seven industry case studies in the context of seven software development companies to evaluate the decision model.

2.3. Expert interviews

The primary source of knowledge to build a valid decision model for this work is domain experts. We followed Myers and Newman guidelines [28] to conduct a series of qualitative semi-structured interviews with senior software engineers to explore expert knowledge regarding the programming language selection problem. We developed a role description before contacting potential experts to ensure the right target group. We contacted the experts through email using the role description and information about our research topic. The experts were pragmatically and conveniently selected according to their expertise and experience mentioned on their *LinkedIn* profile. We considered a set of expert evaluation criteria (including *Years of experience*, *Expertise*, *Skills*, *Education*, and *Level of expertise*) to select the experts.

Each of the interviews followed a semi-structured interview protocol (see Appendix) and lasted between 60 and 90 min. We used open questions to elicit as much information as possible from the experts minimizing prior bias. All interviews were done in person, recorded with the interviewees' permission, and then coded for further analysis.

Fifteen experts (fourteen senior software engineers and one business consultant) participated in this research to answer the research questions and build a decision model for the programming language selection problem. Acquired knowledge during each interview is typically propagated to the next to validate the captured knowledge incrementally. Finally, the findings were sent to the interview participants afterward for final confirmation. Note, for the validity of the results, the research's data collection phases were not affected by the case study participants; moreover, none of the researchers were involved in the case studies.

2.4. Documentation analysis

Document analysis is one of the analytical methods in qualitative research that requires data investigation and interpretation to elicit meaning, gain understanding, and develop empirical knowledge [29]. To build a decision model for the programming language selection problem, we reviewed web pages, whitepapers, scientific articles, fact sheets, technical reports, product wikis, product forums, product videos, and webinars to collect data. Accordingly, we reviewed 489 unique resources (including webpages, whitepapers, and scientific articles) to map the programming language features to the programming language.

A structured coding procedure is employed to extract knowledge from the selected sources of knowledge. Structured coding captures a conceptual area of the research interest [30]. The extracted knowledge has been classified into five categories: *quality attributes*, *programming languages*, *programming language features*, *impacts of the programming language features on the quality attributes*, and *supportability of the programming language features by the programming languages*. Afterward, the extracted knowledge was employed to build a decision model for the programming language selection problem. Then, the decision model was uploaded to the knowledge base of the DSS.

2.5. Case study

Case study research is an empirical research method [31] that investigates a phenomenon within a particular context in the domain of interest [32]. Case studies can describe, explain, and evaluate a hypothesis. A case study can be employed to collect data regarding a particular phenomenon, apply a tool, and evaluate its efficiency and effectiveness using interviews. Note, we followed the guidelines outlined by Yin [33] to conduct and plan the case studies.

Objective: Building a valid decision model for the programming language selection problem was the main goal of this research.

The cases: The analysis units were seven industry case studies, performed in the Netherlands and Iran, in the context of seven software development.

Methods: We conducted multiple expert interviews with the case study participants to collect data and identify their requirements and preferences regarding the programming language selection problem.

Selection strategy: In this study, we selected *multiple case study* [33] to analyze the data both within each situation and across situations, to more extensive exploring the research questions and theoretical evolution, and to create a more convincing theory.

Theory: The proposed decision model is a valid reference model to support software engineers with the programming language selection problem.

Protocol: To conduct the case studies and evaluate the proposed decision model, we followed the following protocol:

Step 1. Requirements elicitation: The participants defined their programming language feature requirements and prioritized them based on the *MoSCoW* prioritization technique [34]. Furthermore, they identified a set of programming languages as potential solutions for their software projects.

Step 2. Results and recommendations: We defined seven separate cases on the DSS portal according to the case studies' requirements and priorities. Next, the DSS suggested a set of feasible solutions per case individually. Then, the outcomes were discussed with the case study participants.

Step 3. Analysis: We compared the DSS feasible solutions with the experts' solutions at the case study companies had suggested. Moreover, we analyzed the outcomes and our observations and then reported them to the case study participants.

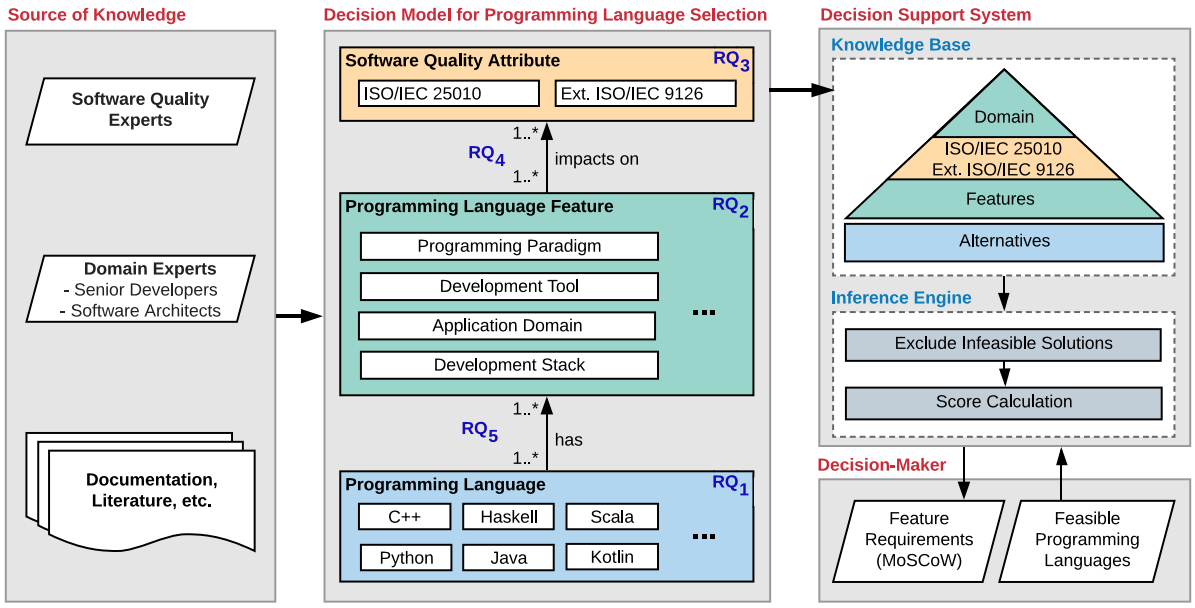


Fig. 1. Illustrates the main building blocks of the framework, adapted from our previous study [13]. On the left, the sources of knowledge, in the middle, the proposed decision model for the programming language selection problem, and on the right, the decision support system is modeled.

Note, based on the framework, we built decision models for database management systems [18], cloud service providers [13], software architecture patterns [15,19], model-driven platforms [16], and blockchain platforms [17].[4] Several case studies were conducted to evaluate the DSS's effectiveness and usefulness to address these MCDM problems. The results showed that the decision models could reduce decision-making time and support the decision-makers' decision-making process.

3. Multi-criteria decision-making for programming language selection

Decision theories are widely applied in many disciplines, such as e-learning [35] and software production [36–38]. In literature, *decision-making* is typically defined as a process or a set of ordered activities concerning stages of problem identifying, data collection, defining alternatives, selecting a shortlist of alternatives as feasible solutions with the ranked preferences [39–44].

In the software production domain, software producing organizations need to decide whether to use their internal development resources (in-house), buying commercial off-the-shelf components, do subcontracting (outsourcing), or whether to use open-source software [11]. A decision problem in software production is not addressed in the same way by all software engineers. Each software engineer has her priorities, tacit knowledge, and decision-making policy [45]; consequently, one software engineer's judgment is expected to differ. Addressing such issues in building decision models in software production forms the focal point of interest in multiple-criteria decision-making (MCDM).

MCDM is both an approach and a set of techniques to provide an overall ranking of alternative solutions, from the most preferred to the least preferred solution [46]. Alternative solutions may differ in how they achieve several objectives, and no one alternative solution will be best in achieving all objectives. Besides, some conflict or trade-off is usually evident amongst the objectives; alternative solutions that are more beneficial are usually more costly. Costs and benefits typically conflict, but so can short-term benefits compared to long-term ones, and risks may be higher for the otherwise more beneficial options.

MCDM problems consist of a finite set of alternative solutions, explicitly known at the beginning of the solution process [47]. In multi-criteria design problems (multiple objective mathematical programming problems), alternative solutions are unknown. An alternative solution can be found by solving a mathematical decision model. Typically, the number of alternatives is either infinite or not countable when variables are continuous or very large if countable when variables are discrete. However, both kinds of problems are considered sub-classes of MCDM problems.

Each decision-making problem in software production can be modeled as an MCDM problem that deals with evaluating a set of alternatives and considering a set of decision criteria [13,16–19]. The challenge consists of evaluating and selecting the most suitable alternatives for software engineers (decision-makers) according to their preferences and requirements [27]. In this study, we formulate the programming language selection problem as an MCDM problem in software production:

Let $Languages = \{l_1, l_2, \dots l_{|Languages|}\}$ be a set of programming languages in the market (i.e., *C++*, *Ruby*, and *Python*), and $Features = \{f_1, f_2, \dots t_{|Features|}\}$ be a set of programming language features (i.e., Supporting threading and Multi-platform) of the programming languages. Each language l, where $l \in Languages$, supports a subset of the set $Features$. The goal is finding the best fitting programming languages as solutions, where $Solutions \subset Languages$, that support a set of programming language feature requirements, called *Requirements*, where $Requirements \subseteq Features$.

An MCDM approach for the selection problem receives $Languages$ and their $Features$ as its input, then applies a weighting method to prioritize the $Features$ based on the decision-makers' preferences to define the $Requirements$, and finally employs a method of aggregation to rank the $Languages$ and suggests $Solutions$. Accordingly, an MCDM approach can be formulated as follows:

$$MCDM : Languages \times Features \times Requirements \rightarrow Solutions$$

Typically, a unique optimal solution for an MCDM problem does not exist, and it is necessary to employ decision-makers' preferences to differentiate between solutions [27]. Fig. 2 visualizes MCDM approach

[4] The decision models and modeling studio are available on the DSS website: https://dss-mcdm.com.

Table 1
Shows the programming languages that were mentioned on at least three sources of knowledge, including experts and well-known websites. This list has been considered as the programming language alternatives in the decision model.

	Agreement	Initial Hypothesis	GitHut Ranks	stackoverflow Ran	Language Migratio	hired.com	infoq.com	codingame.com	jetbrains.com	TIOBE Index	PYPL	hackernoon	Coding infinite	statista	stackify	dzone	Redmonk	IEEE	Domain Expert 1	Domain Expert 2	Domain Expert 3	Domain Expert 4	Domain Expert 5	Domain Expert 6	Domain Expert 7	Domain Expert 8	Domain Expert 9	Domain Expert 10	Domain Expert 11	Domain Expert 12
Python	100.00%	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
C	100.00%	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
C#	100.00%	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Java	100.00%	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
C++	96.55%	✓	✓	✓	✓	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
JavaScript	96.55%	✓	✓	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
PHP	96.55%	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Visual Basic .NET	72.41%	✓	×	✓	✓	×	×	✓	✓	✓	✓	×	×	✓	✓	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Ruby	68.97%	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	✓	×	✓	✓	×	×	✓	×	×	×	×	×	✓	✓
R	65.52%	✓	✓	✓	✓	✓	×	×	✓	✓	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	×	×	×	×	×	✓	✓
Swift	65.52%	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	✓	✓	✓	✓	×	×	×	×	✓	×	×	×	×	×	✓	✓
Go	62.07%	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	×	×	×	×	✓	×	×	×	×	×	×	✓
HTML / CSS	58.62%	×	✓	✓	×	×	×	×	✓	×	×	×	×	✓	×	×	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Objective-C	58.62%	✓	✓	✓	✓	✓	×	✓	✓	✓	✓	×	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×	×	×	✓
SQL	55.17%	✓	×	✓	×	×	×	×	✓	✓	×	×	×	✓	✓	×	×	×	✓	✓	×	✓	✓	×	✓	✓	✓	✓	✓	✓
Matlab	48.28%	✓	✓	×	✓	×	×	×	✓	✓	×	×	×	✓	✓	✓	×	✓	✓	✓	✓	×	×	✓	×	×	×	×	×	✓
Kotlin	44.83%	✓	×	✓	✓	✓	×	✓	✓	✓	✓	×	✓	✓	×	×	✓	×	×	×	×	×	×	×	✓	×	×	×	×	✓
Scala	44.83%	✓	✓	✓	✓	✓	×	✓	✓	✓	✓	×	✓	✓	×	×	✓	×	×	×	×	×	×	×	×	×	×	×	×	✓
TypeScript	41.38%	✓	✓	✓	×	✓	✓	×	✓	✓	✓	✓	×	✓	×	×	✓	×	×	×	×	×	×	×	×	×	×	×	×	✓
Rust	37.93%	✓	✓	✓	✓	×	✓	✓	✓	✓	✓	×	✓	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
Assembly language	37.93%	✓	✓	✓	×	×	×	×	✓	✓	×	×	×	✓	✓	×	×	×	✓	×	×	×	✓	×	×	×	×	×	✓	✓
Perl	34.48%	✓	✓	×	✓	×	×	✓	✓	✓	×	×	×	✓	×	×	✓	×	×	×	×	×	×	×	×	×	×	×	×	✓
Clojure	31.03%	✓	✓	✓	✓	×	×	✓	✓	✓	×	×	✓	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
Visual-Basic	31.03%	✓	×	×	✓	✓	×	×	✓	✓	×	×	×	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓
Haskell	31.03%	✓	✓	×	✓	×	×	✓	✓	✓	×	×	×	×	×	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	✓
shell	27.59%	✓	✓	✓	×	×	×	×	✓	×	×	×	✓	✓	×	×	✓	×	×	×	×	×	×	×	×	×	×	×	✓	×
Dart	27.59%	✓	✓	✓	×	×	✓	✓	✓	✓	×	×	×	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
F#	27.59%	✓	✓	✓	×	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
PowerShell	24.14%	✓	✓	✓	×	×	×	×	✓	×	×	×	×	✓	×	×	✓	×	×	×	×	×	×	×	×	×	×	×	×	×
Lua	24.14%	✓	✓	×	✓	×	×	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
Julia	20.69%	✓	✓	×	×	×	×	×	✓	✓	×	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
Groovy	20.69%	✓	✓	×	×	×	×	✓	✓	×	×	×	×	×	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
Elixir	17.24%	✓	✓	✓	×	×	×	×	✓	×	×	×	×	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
Bash	17.24%	✓	×	✓	×	×	×	✓	×	×	×	×	✓	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
ASP.net	17.24%	✓	✓	×	×	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓
Delphi	13.79%	×	×	×	×	×	×	✓	✓	×	✓	×	×	×	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
Fortran	13.79%	✓	✓	×	✓	×	×	×	×	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
Erlang	13.79%	✓	×	✓	×	×	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	×
D	10.34%	✓	✓	×	×	×	×	×	×	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
CoffeeScript	10.34%	✓	✓	×	×	×	×	×	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
ActionScript	10.34%	✓	✓	×	×	×	×	×	×	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
WebAssembly	10.34%	✓	×	✓	×	×	×	×	×	×	×	×	×	×	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
Common Lisp	10.34%	✓	✓	×	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
COBOL	10.34%	✓	×	×	✓	×	×	×	×	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
Logo	10.34%	✓	✓	×	×	×	×	×	×	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
Scheme	10.34%	✓	✓	×	×	×	×	×	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
OCaml	10.34%	✓	✓	×	×	×	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
Object Pascal	10.34%	✓	×	×	×	×	×	×	×	✓	×	×	×	×	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×

for the programming language selection problem in a 3D space. It shows that the degree of satisfaction of the decision-makers with a suggested solution is fuzzy, which means that the satisfaction degree from a decision-maker perspective may range between completely true (best fit) and completely false (worst fit) [48], which is represented by a range of colors from red to dark green.

As aforementioned, we follow the framework [13] as modeled in Fig. 1 to build a decision model for the programming language selection problem. Generally speaking, a decision model for an MCDM problem contains decision criteria, alternatives, and relationships among them. Fig. 1 represents the main building blocks of the framework, including the source of knowledge, the proposed decision model, and the decision support system.

3.1. Programming language alternatives (RQ_1)

This study only focuses on programming languages used in computer programming to develop software-intensive applications or implement algorithms. Accordingly, we are not interested in domain-specific programming languages such as *Arduino*,[5] DOT,[6] or *CFML*.[7]

To answer the first research question, we identified a set of alternatives, including 594 programming languages, based on a variety of programming language websites and related forums as our initial hypothesis. Next, we reviewed the published surveys and reports

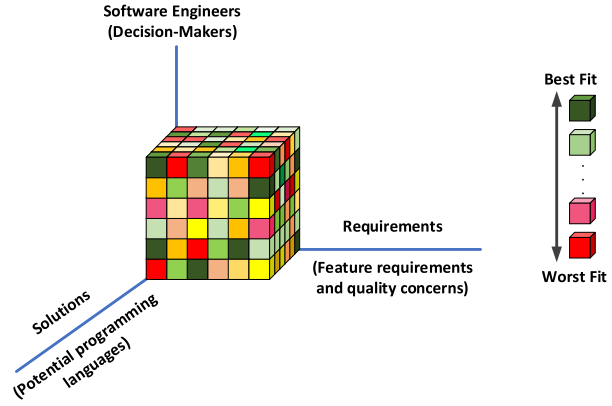


Fig. 2. Is an MCDM approach for the programming language selection problem in a 3-dimensional space. The degree of the decision-makers' satisfaction with solutions (potential programming languages) according to their priorities and preferences (requirements) ranges between the best and worst fit solutions, represented by a range of colors from red to dark green. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

from well-known knowledge bases, including GitHut [75], StackOverflow [78], Language Migration [76], Hired.com [77], Infoq.com [83], Codingame.com [79], Jetbrains.com [86], TIOBE Index [90], PYPL [84], Hackernoon [80], Coding infinite [89], Statista [82], Stackify [81], Dzone [88], Redmonk [85], and IEEE [87]. Afterward, we conducted a set of expert interviews with twelve experts to gain more insight into the popular and applicable programming languages and to evaluate our findings. It is interesting to highlight that most of the domain experts were familiar with a limited number of the list's

[5] Arduino is mainly used to program micro-controllers.

[6] DOT is a development tool that is optimized for the processing of graph-structured data.

[7] The ColdFusion Markup Language (CFML) is a set of tags used in ColdFusion pages to interact with data sources, manipulate data, and display output.

programming languages (See the twelve Domain Experts' columns on Table 1). To prevent potential biases, we only considered the programming languages mentioned on at least three resources. Finally, we analyzed the data and ended with 47 alternative programming languages mentioned on at least three resources. Table 1 shows the complete list of the programming languages that we have selected in the decision model.

3.2. Programming language features (RQ_2)

Domain experts were the primary source of knowledge to identify the right set of programming language features, even though documentation and literature study of programming languages can be employed to develop an initial hypothesis about the programming language feature set. Each programming language feature has a data type, such as *Boolean* and *non-Boolean*. For example, the data types of programming language features, such as the *popularity in the market* and supportability of *Object-oriented programming*, can be considered as *non-Boolean* and *Boolean*, respectively.

The initial set of programming language features was extracted from online documentation of programming languages. Then, a list of essential programming language features was identified during twelve domain expert interviews. Finally, 94 Boolean and 13 non-Boolean programming language features[8] were identified and confirmed by the domain experts.

3.3. Software quality attributes (RQ_3)

Quality attributes are characteristics of a software product that are intrinsically non-functional. One of the primary concerns of software engineers in the implementation phase of a software product is to satisfy the quality requirements. In other words, the quality of a system is the degree to which the system meets its requirements (functionality, performance, security, maintainability, etc.). It is necessary to find quality attributes widely recommended by other researchers to measure the system's characteristics.

The literature study results confirmed that researchers do not agree upon a set of standard criteria, including quality attributes and features, to evaluate the programming languages (See Table 6). Additionally, we realized that the suggested criteria were mainly applied to specific domains to address different research questions. Thus, a set of generic and domain-independent criteria is required to assess programming languages.

The ISO/IEC 25010 [50] provides best practice recommendations on the foundation of a quality evaluation model. The quality model determines which quality characteristics should be taken into account when evaluating a software product's properties. A set of quality attributes should be defined in the decision model [13]. In this study, we employed the *ISO/IEC 25010* standard [50] and *extended ISO/IEC 9126* standard [51] as two domain-independent quality models to analyze programming language features based on their impact on quality attributes of programming languages. The key rationale behind using these software quality models is that they are a standardized way of measuring a software product. Moreover, they describe how easily and reliably a software product can be used.

The last four columns of Table 6 show the results of our analysis regarding the common criteria and alternatives of this study with the selected publications. Let us define the coverage of the ith selected study as follows: $Coverage_i = \frac{CQ_i + CF_i}{C_i} \times 100$ where CQ_i and CF_i denote the numbers of common quality attributes (column #CQ) and features (column #CF) of the ith selected study with this study, respectively. Furthermore, C_i signifies its number of suggested criteria. The last column (*Cov.*) of Table 6 indicates the percentage of the coverage of the considered criteria within the selected studies. On average, 80% of those criteria are already considered in this study.

3.4. Impacts of the programming language features on the software quality attributes (RQ_4)

The mapping between the sets *software quality attributes* and *programming language features* has been determined based on domain experts' knowledge. Three domain experts participated in this phase of the research to map the programming language features ($Features$) to the quality attributes ($Qualities$) based on a Boolean adjacency matrix[9] ($Qualities \times Features \rightarrow Boolean$). For instance, *support threading* as a programming language feature influences the *Time behavior* quality attribute. The experts believed that about 68% percent of the programming language features have impacts on the following quality aspects of the programming languages:

- **Usability** defines the degree to which a programming language can be used to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use. Moreover, it embraces quality attributes such as *Learnability*, *Operability*, *User error protection*.
- **Cost** denotes the amount of money that a company spends on implementing a software product using a programming language. It includes quality attributes such as *Implementation Cost*, *Platform Cost*, and *Licensing Costs*.
- **Product** defines a set of quality attributes regarding the state or fact of exclusive rights and control over the property. For instance, *Stability*, *Ownership*, and *Guarantees* are part of this characteristic.
- **Supplier** includes a set of quality attributes such as *Reputation* and *Support* of the programming languages.
- **Maintainability** is the degree to which a programming language can be effectively and efficiently modified without introducing defects or degrading existing product quality. It can be supported by a programming paradigm, such as *object-oriented programming*, that encourages *Modularity*, *Reusability*, *Analyzability*, *Modifiability*, and *Testability*.

The acquired knowledge regarding the impacts of the programming language features on the quality attributes was used to calculate the Impact Factors [13] that apply in the score calculation of the DSS. The framework does not enforce a programming language feature to present in a single quality attribute; programming language features can be part of many quality attributes. For example, *object-oriented programming* as a feature might connect to multiple quality attributes such as *Functional completeness* and *Interoperability*.

In this study's knowledge extraction phase, we realized some inconsistencies regarding the programming language features' impacts on the quality aspects. For example, one of the experts asserted that when a programming language supports object-oriented programming as one of its paradigms, the programming language's operability and User error protection will be increased. However, the other asserted that object-oriented programming does not have any impact on these two quality aspects. Thus, we used the fuzzy Delphi technique to reach a consensus among experts regarding the impacts of the programming language features on the quality aspects.

[8] The entire lists of the programming language features and their mapping with the considered programming languages are available and accessible on the *Programming Language Selection* website (https://dss-mcdm.com).

[9] The final Boolean adjacency matrix is available on Mendeley Data [49]. It is made publicly available to enable other researchers to use it for their research purposes.

Table 2
Shows a subset of the Boolean Features ($Feature^B$), the programming languages ($Languages$), and the *BFL* mapping, where $BFL : Feature^B \times Languages \rightarrow \{0,1\}$. Note, we reviewed 489 unique resources (including webpages, whitepapers, and scientific articles) to map the programming language features to the programming language. The entire list of the features and mappings is available on Mendeley Data [49].

Boolean Features / Programming Languages	Coverage	Java	Python	C++	C#	C	Scala	Swift	Perl	D	Object Pascal	Ruby	Common Lisp	F#	PHP	OCaml	Lua	JavaScript	Erlang	Go	Fortran	Visual Basic .NET	Clojure	HTML / CSS	Objective-C	TypeScript	Elixir	Visual-Basic	Kotlin	ASP.net	Delphi	Rust	ActionScript	Groovy	Dart	R	Haskell	Julia	Assembly language	Scheme	Matlab	Bash / Shell	CoffeeScript	SQL	COBOL	PowerShell	Logo	WebAssembly
Application Domain																																																
Interactive System	80.85%	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	0	1	0	1	1	0	1	1	1	0	1	1	1	0
Compiler Design	31.91%	1	1	1	1	1	0	0	0	1	1	1	0	1	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0
Commercial-Off-The-Shelf (COTS)	8.51%	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Data Base Systems	27.66%	1	1	1	1	1	1	0	0	0	1	0	0	0	0	0	0	1	1	1	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Distributed Systems	29.79%	1	0	1	1	1	1	0	1	1	0	0	1	0	0	1	0	0	1	1	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
Cloud Computing Applications	36.17%	1	1	1	1	1	0	0	0	1	1	0	0	1	1	1	1	0	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0
Mobile Applications	65.96%	1	1	1	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	0	1	0	1	1	1	0	1	0	0	0	0	0	1	0	0	0	0	0
Web-Based Systems	70.21%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1	1	1	1	1	1	1	1	0	1	1	1	1	1	0	0	0	0	0	1	0	0	0	0	0
Web Services	48.94%	1	1	1	1	1	0	1	1	0	0	1	0	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	0	1	0	1	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0
Service-Based Systems	70.21%	1	1	1	1	1	1	0	1	1	1	1	0	1	1	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	0	1	0	0	0	0	0
Plug-and-Play Environment	17.02%	1	1	1	1	0	0	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Real-Time Systems	23.40%	1	1	1	0	1	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	1	0	1
File-Sharing Applications	31.91%	1	1	1	1	1	0	1	1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
Exchange Data And Information	27.66%	0	1	1	0	1	1	0	0	1	1	0	0	1	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0	1	0	0	0	0
Banking System	44.68%	1	1	1	1	0	1	1	1	0	0	1	0	1	1	1	0	0	1	0	1	0	0	1	0	1	0	1	1	0	0	0	0	0	0	1	0	0	0	1	0	0	0	1	1	0	0	0
Platforms																																																
Cross platform	89.36%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0
Windows	89.36%	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	1	1	0
Linux	82.98%	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0
macOS	68.09%	1	0	1	1	1	1	1	1	1	1	0	0	1	0	1	1	0	0	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	0	1	1	1	0	1	1	0	1	0	0	0	0
Docker	72.34%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	0	1	0	1	1	0	1	0	0	1	0	0	0	1	0	0	0	0
AWS	61.70%	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Microsoft Azure	55.32%	1	1	0	1	0	1	1	1	1	1	0	1	1	1	1	0	1	1	0	0	1	1	1	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0
IBM Cloud or Watson	34.04%	1	0	0	1	0	0	1	0	0	0	1	1	0	1	1	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
Kubernetes	38.30%	0	0	0	0	0	1	1	1	1	0	0	1	1	0	1	1	0	1	0	1	0	1	1	0	1	1	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Android	57.45%	1	1	1	1	1	1	0	0	1	1	0	1	1	1	1	1	0	1	1	1	1	1	1	0	1	1	0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
iOS	57.45%	1	1	1	1	0	1	1	0	1	1	0	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	0	1	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Web-based	72.34%	1	1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	0	1	0	0	0	0	0

Table 3
Shows the Non-Boolean Features ($Feature^N$), the programming languages ($Languages$), and the *NFL* mapping, where $NFL : Features^N \times Languages \rightarrow \{H,M,L\}$. Note, this table is mainly the result of the expert interviews and reviewing the sources of knowledge indicated in the *source of knowledge* column.

Non Boolean Features / Programming Languages	Java	Python	C++	C#	C	Scala	Swift	Perl	D	Object Pascal	Ruby	Common Lisp	F#	PHP	OCaml	Lua	JavaScript	Erlang	Go	Fortran	Visual Basic .NET	Clojure	HTML / CSS	Objective-C	TypeScript	Elixir	Visual-Basic	Kotlin	ASP.net	Delphi	Rust	ActionScript	Groovy	Dart	R	Haskell	Julia	Assembly language	Scheme	Matlab	Bash / Shell	CoffeeScript	SQL	COBOL	PowerShell	Logo	WebAssembly	Source of Knowledge
Popularity in the market	H	H	H	H	H	H	H	L	L	L	M	L	L	H	L	L	H	L	H	L	M	L	H	M	M	L	L	L	H	L	H	L	M	L	H	L	L	M	M	H	L	L	H	L	M	L	L	Domain Experts
TIOBE Index (January 2020)	16.90%	9.70%	5.57%	5.34%	15.77%	0.37%	1.80%	0.74%	0.82%	0.99%	1.06%	0.00%	0.23%	2.40%	0.00%	0.23%	2.45%	0.00%	0.90%	0.31%	5.28%	0.00%	0.00%	0.92%	0.18%	0.00%	0.83%	0.29%	0.00%	0.99%	0.40%	0.00%	0,61%	0.63%	0.80%	0.19%	0.18%	0.87%	0.20%	0.73%	0.00%	0.00%	1.50%	0.38%	0.00%	0.00%	0.00%	https://www.tiobe.com
LinkedIn (Jobs)	12K+	9K+	4K+	11K+	3K+	8K+	781	543	-	-	1K+	-	6	10K+	2	45	17K+	50	11K+	97	46	65	7K+	459	2K+	123	299	625	5K+	316	6K+	18	473	100	5K+	90	529	1K+	2K+	1K+	103	75	20K+	56	2K+	-	4	https://www.linkedin.com/
GitHub (Repositories)	808K+	833K+	171K+	121K+	114K+	59K+	126K+	27K+	838	14	240K+	4K+	4K+	348K+	7K+	19K+	362K+	11K+	224K+	3K+	454	25K+	218K+	16K+	72K+	26K+	156	66K+	3K+	6K+	62K+	1K+	6K+	14K+	112K+	39K+	9K+	2K+	3K+	42K+	49K+	3K+	5K+	777	25K+	365	189	https://github.com/
Google Trends (mean of the last 5 years)	58.15	66.02	79.48	83.73	78.58	76.24	76.44	62.57	30.67	54.44	75.85	47.97	31.25	75.06	56.64	74.03	84.18	66.69	64.89	70.30	63.16	69.93	73.47	67.19	55.53	60.46	60.76	39.57	65.47	34.27	43.82	32.31	82.16	31.53	81.56	65.84	31.16	74.87	67.66	71.44	16.76	40.11	80.08	68.34	81.48	14.35	11.42	https://trends.google.com/
Reusability	H	H	H	H	H	H	H	H	H	H	H	H	M	H	M	M	H	M	M	M	M	M	H	M	M	M	M	M	M	M	M	M	M	L	M	M	L	M	M	L	L	M	L	L	M	L	L	Domain Experts
Application domains	34	33	34	33	30	19	19	18	20	23	19	20	16	22	16	15	18	16	16	11	12	11	21	17	14	13	15	13	8	10	10	13	7	7	13	9	6	11	12	4	9	10	9	4	9	2	3	BFL mapping
Modular paradigms	5	5	4	4	3	3	3	5	3	3	5	5	3	4	3	4	4	2	3	2	2	2	0	3	2	2	1	2	3	2	3	4	5	2	4	2	4	0	1	4	1	1	0	3	2	1	0	BFL mapping
Maturity level	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	M	M	M	H	M	H	M	H	M	M	M	H	M	M	M	M	M	H	M	M	L	M	H	L	L	L	L	L	L	L	Domain Experts
Boolean Features' coverage	96%	92%	88%	91%	80%	70%	69%	69%	71%	70%	67%	67%	67%	78%	64%	64%	69%	59%	59%	58%	65%	59%	60%	58%	64%	56%	59%	56%	59%	56%	55%	54%	52%	51%	51%	51%	45%	40%	42%	40%	40%	39%	33%	35%	28%	25%	11%	BFL mapping
Comprehensive documentation	H	H	H	H	H	M	H	M	L	L	L	M	M	H	L	L	H	M	H	H	H	M	H	H	L	L	H	H	H	M	H	M	M	H	H	M	M	M	M	H	M	L	H	M	L	M	L	Domain Experts
Google hits	3M+	7M+	1M+	500K+	4M	100K+	400K+	255K	272K	12K+	313K	21K	16K+	371K	8K+	79K+	283K	35K+	527K	76K+	1M+	23K+	458K	208K	10K+	18K+	323K	108K	104K	63K+	179K	59K+	24K+	180K	750K	76K+	45K+	69K+	54K+	141K	20K+	1K+	205K+	61K+	5K+	61K+	199	https://www.google.com/
GitHub (Topics)	3K	2K	25	25	25	411	715	181	1	3	611	6	3	3K	47	402	1K	141	5K	60	1	103	1K	2	393	160	16	501	2	80	392	8	31	122	2	165	75	20	192	156	212	12	989	13	155	2	12	https://github.com/
GitHub (Wikis)	252K	241K	511K	511K	511K	23K	13K	24K	13K	2K	80K	2K	210K	189K	3K	23K	238K	7K	510K	6K	4K	13K	408K	15K	13K	3K	21K	4K	12K	1K	15K	2K	12K	3K	240K	9K	3K	6K	46K	15K	65K	9K	91K	976	12K	1K	651	https://github.com/
Active community	H	H	H	H	H	H	H	H	M	M	H	L	H	H	H	H	H	M	H	M	L	M	M	H	H	M	L	M	M	L	H	L	H	H	H	H	H	M	H	M	H	L	H	L	L	L	L	Domain Experts
stackoverflow	500	500	500	500	500	500	500	500	450	471	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	https://stackoverflow.com/
GitHub (Commits)	22M	19M	336M+	362M+	362M+	1M	1M	6M	47K	40K	4M	53K	76M+	18M	727K	3M	3M	252K	150M+	955K	4K	189K	48M+	982K+	709K	202K	34K	380K	242K	67K	964K	33K	723K	1M	244M+	1M	23M	165K+	17M	373K	6M+	130K	6M	15K	278K	476	327K	https://github.com/
GitHub (Issues)	2M	2M	272K	377K	1M	221K	146K	227K	3K	10K	761K	2K	12K	1M	33K	84K	1M	61K	1M	6K	59	33K	613K	30K	374K	46K	238	48K	98	14K	210K	6K	24K	102K	138K	51K	96K	305	3K	17K	76K	11K	913+	155	39K	1K	2K	https://github.com/
Github (Marketplace)	28	49	603	603	603	5	21	1	-	-	26	-	270	53	3	5	31	1	58	1	-	4	13	2	7	7	-	3	-	-	15	-	1	3	382	1	2	-	1	-	7	1	3	-	-	-	-	https://github.com/
GitHub (Packages)	219	117	5K	5K	5K	21	10	8	5K	-	103	-	1K	156	1	12	884	3	488	-	11	2	751	6	13K	3	15	50	47	1	23	-	3	7	3K	4	-	6	16	-	492	4	94	1	16	3	15	https://github.com/
Availability of developers	H	H	H	H	H	M	M	M	M	L	H	L	L	H	L	L	H	L	H	M	M	L	H	M	M	L	H	L	H	M	L	M	L	M	M	L	L	M	M	H	M	L	H	M	M	L	L	Domain Experts
GitHub (Users)	27K	22K	0	7K	3K	866	2K	3K	7	46	5K	42	0	12K	57	375	47K	804	3K	39	0	368	4K	8	486	310	0	382	25	221	689	13	75	209	2K	395	209	10	12	248	353	9	13K	12	652	2	0	
LinkedIn (People)	9M+	5M+	5M+	3M+	11M+	241K+	574K+	761K+	430K+	24K+	682K+	9K+	13K+	3M+	18K+	85K+	6M+	24K+	4M+	205K+	365K+	20K+	4M+	517K+	258K+	39K+	1M+	99K+	1M+	334K+	95K+	134K+	90K+	235K+	763K+	69K+	24K+	320K+	120K+	3M+	124K+	30K+	10M+	341K+	295K+	94K+	2K+	https://www.linkedin.com/
Scalability	H	H	M	H	M	M	L	L	L	M	H	L	L	H	L	L	H	L	H	L	L	M	H	L	H	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	H	L	L	L	L	L	L	Domain Experts
Distributed Systems	5K	1K+	891	324	805	214	-	-	-	-	173	-	-	-	-	-	813	-	1K+	-	-	-	304	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	https://github.com/
Cloud native systems	652	123	-	51	-	-	-	-	-	49	-	-	-	-	-	-	2K+	-	312	-	-	-	250	-	58	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	176	-	-	-	-	-	-	https://github.com/
Service based systems	202	69	34	34	-	-	-	-	-	-	18	-	-	77	-	-	71	-	-	-	-	-	45	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	25	-	-	-	-	-	-	https://github.com/
Microservices	15K	3K	-	2K	-	-	-	-	-	-	1K+	-	-	1K+	-	-	11K	-	3K	-	-	-	2K	-	1K+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1K	-	-	-	-	-	-	https://github.com/
Component based systems	67	33	71	41	23	-	-	-	-	-	-	-	-	36	-	-	79	-	-	-	-	12	20	-	19	-	-	-	-	-	-	-	-	-	-	-	-	-		-	-	-	-	-	-	-	-	https://github.com/

Table 4
Shows the feature requirements, based on the MoSCoW prioritization technique (Must-Have (M), Should-Have (S), and Could-Have (C)). Note, The entire list of the features requirements is available on Mendeley Data [49].

	Oceaneering	Dooman ltd.	Saanaa DP	ASC	FinanceComp	SecureSECO	ENVRI-FAIR	
Debugger	M	M	M	M	M	M	M	R01
Object-oriented programming (OOP)	S	M	M	M	M	M	M	R02
Testing tools	M	C	M	M	M	M	M	R03
Socket programming	M	M	C	M	M	M	M	R04
Support threading	M	S	M	M	S	M	M	R05
Scalability	M	M	M	M	S	M	S	R06
Popularity in the market	M	S	M	M		M	M	R07
Maturity level	M	M	M	S	M		M	R08
Open source compiler or Interpreter	M	M	C	C	M	M	M	R09
Web-based		M	M	M	M	C	M	R10
General-Purpose PL	M	M	S	M	M	S	S	R11
Web-Based Systems	S	M	M	M	M	S	S	R12
Web Services	S	M	C	M	M	S	M	R13
Windows	S	M	M	M	M			R14
Free implementation of the core libraries	M	C	C	C	M	M	M	R15
Cross platform / Multiplatform	M	M	C			M	M	R16
Open source	M	M	C			M	M	R17
Back-end		M	C	M	M	M		R18
Full-stack		M	C	M	M		M	R19
Event-driven programming	M	M	M	S	S	S	S	R20
Comprehensive consistent documentation	M	M		M	S	S	S	R21
Software Architecture Patterns	M	M	M	S	C	S	C	R22
Package Manager	S	S		M	C	M	M	R23
Database Systems	S	M	M	M		S	C	R24
Toolchain	M	M	C	S		M	C	R25
Reusability	M	S		M			M	R26
Compiler	M	M	M	C		C	C	R27
Code coverage	M	C		M		M		R28
Front-end		M	C	M	M			R29
Imperative programming	M			M		M	S	R30
Maintainability	M	S	S	S	M		S	R31
Software Architecture Design Patterns	M	M	S	S	C	S	C	R32
Static code analysis	M		C	M	S	S	C	R33
Generic programming	S	M		M	S			R34
Human Resource Availability (Developers)		M			M	S	S	R35
Profiler	M	C	C	M		S	C	R36
Compiler Design	M	M				S		R37
Linux	M	M	C	C				R38
Interactive System	M	M	C	C			S	R39
Easy to write new code	S	S	M	S		S	S	R40
Easy to read existing code	S	M	S	S		S	S	R41
Easy to reuse existing code	S	M	S	S		S	S	R42
Service-Based Systems	C	C	S	M		S	S	R43
Functional programming	M		C	S	S	S		R44
Accessible and friendly community	M	C		S		S	S	R45
Docker	M	C	C	S		C		R46
Code refactoring	C	C		M		S		R47

	Oceaneering	Dooman ltd.	Saanaa DP	ASC	FinanceComp	SecureSECO	ENVRI-FAIR	
Reflective programming	C	M		S			S	R48
Network and Communication Systems		M				S		R49
Banking System			S		M			R50
Amazon Web Services (AWS)	C	C		M		C	C	R51
Declarative programming	C			M				R52
Licensed				M				R53
Operating Systems		M						R54
ORM	C	S	C	S	S	S	S	R55
Self-documenting (or self-describing) syntax	S	S	C	S		S	S	R56
Simple and concise syntax	S	S		S		S	S	R57
Procedural programming	C			S	S	S	S	R58
Distributed Systems	S	C				S	S	R59
Mobile Applications	C	S	S				S	R60
Cloud Computing Applications	C	S	C	C		S	C	R61
Aspect-Oriented Programming (AOP)	C					S	S	R62
Data-Dominant Software						S	S	R63
Data-driven programming	C		C			S		R64
Commercial-Off-The-Shelf (Cots)		C		C			S	R65
GUI builder	C	C	C	C		C	C	R66
Interpreter	C	C	C	C		C	C	R67
Dynamic programming	C			S				R68
macOS	S	C						R69
Information Management and DSSs				C			S	R70
Pattern Recognition	C					S		R71
Bytecode	C		C	C		C	C	R72
Array programming							S	R73
Embedded Systems	S							R74
Real-Time Systems	S							R75
Microsoft Azure	C	C				C	C	R76
IBM Cloud or Watson	C	C				C	C	R77
Kubernetes	C	C		C		C		R78
Constraint programming	C			C		C		R79
Google Cloud Platform	C					C	C	R80
Android	C	C	C					R81
iOS	C	C	C					R82
Flow-Based Programming (FBP)	C			C				R83
Heroku	C					C		R84
Plug-and-Play Environment		C	C					R85
File-Sharing Applications	C			C				R86
Exchange Data And Information	C			C			C	R87
Metaprogramming	C							R88
Non-structured programming	C							R89
Arduino	C							R90
Multi-Processors Environment		C						R91
Expert System				C				R92
Management Information Systems				C				R93
Internet Of Things (Iots)	C							R94

3.5. Supportability of the programming language features by the programming languages (RQ_5)

A Programming language has a set of programming language features that can be either Boolean ($Feature^B$) or non-Boolean ($Feature^N$). A Boolean programming language feature is a feature that is supported by the programming language; for example, supporting the *socket programming*. Additionally, a non-Boolean programming language feature assigns a non-Boolean value to a particular programming language; for example, the *maturity level* of a programming language can be *high*, *medium*, or *low*. Therefore, the programming language features in this study are a collection of Boolean and non-Boolean features, where $Features = Feature^B \cup Feature^N$.

The mapping $BFL : Feature^B \times Languages \rightarrow \{0,1\}$ defines the supportability of the Boolean programming language features by the programming languages. So that $BFL(f,l) = 0$ means that the programming language l does not support the programming language feature f and $BFL(f,l) = 1$ signifies that the language supports the feature. The mapping *BFL* is defined based on documentation of the programming languages and expert interviews. One of the principal challenges is the lack of standard terminology among programming languages. Sometimes different programming languages refer to the same concept by different names, or even worse, the same name might stand for different concepts in different programming languages. Discovering conflicts is essential to prevent semantic mismatches throughout the programming language selection process. Table 2 shows a subset of the Boolean Features that we have considered in the decision model.

We defined thirteen non-Boolean programming language features, such as Scalability and Popularity in the market, and developer availability. The assigned values to these non-Boolean programming language features for a specific programming language is a 3-point Likert scale (**H**igh, **M**edium, and **L**ow), where $NFL : Features^N \times Languages \rightarrow \{H, M, L\}$, based on several predefined parameters. For instance, the *popularity in the market* of programming languages was defined based on the following four parameters: *TIOBE Index* [90], *LinkedIn (Jobs)*, *GitHub (Repositories)*, and the mean of the last five years of *Google Trends*. Table 3 shows a subset of the non-Boolean programming language features, their parameters, and sources of knowledge.

3.6. Programming language feature requirements

The DSS [14,15] receives the programming language feature requirements based on the *MoSCoW* prioritization technique [34]. Decision-makers should prioritize their feature requirements using a set of weights ($W_{MoSCoW} = \{w_{Must}, w_{Should}, w_{Could}, w_{Willnot}\}$) according to the definition of the *MoSCoW* prioritization technique. programming language feature requirements with *Must-Have* or *Won't-Have* priorities act as hard constraints and programming language feature requirements with *Should-Have* and *Could-Have* priorities act as soft constraints. The DSS excludes all infeasible programming languages which do not support programming language features with *Must-Have* and support programming language features with *Won't-Have* priorities. Then, it assigns non-negative scores to feasible programming languages according to the number of programming language features with *Should-Have* and *Could-Have* prioritizes [13].

Decision-makers specify desirable values, from their perspectives, for non-Boolean programming language feature requirements. For example, a decision-maker could be interested in prioritizing programming languages with the *Maturity level* above average. Therefore, the *Maturity level* above average is considered as a *Should-Have* feature.

4. Empirical evidence: The case studies

Seven industry case studies at seven software development companies have been conducted to evaluate and signify the decision model's usefulness and effectiveness to address the programming language selection problem. We selected the case study companies from seven

Table 5
Presents the context of the case study companies (Context), the feature requirements (Requirements), the case study participants' ranked shortlists (CP ranked shortlists), and the outcomes of the DSS for the case studies based on their requirements and priorities (DSS Solutions). Moreover, the numbers of features requirements (#Feature Req) and the percentages of the MoSCoW priorities are shown in the table. Note, the numbers in percentages beside the solutions signify the calculated scores by the DSS. For instance, the score of the *C* programming language for *Oceaneering* is 78% (see [13] for the details).

		Oceaneering	ASC	Dooman ltd.	Saanaa DP	FinanceComp	SecureSECO	ENVRI-FAIR
Context	App. Domain	Control systems	CMS	Helpdesk systems	Booking systems	Financial systems	Distributed ledgers	Search engines
	#Employees	800-1000	101-250	50-80	20-50	4500-5000	20-50	101-250
	Country	Netherlands	Netherlands	Iran	Iran	Iran	Netherlands	Netherlands
Requirements	Must-Have	38.96%	44.62%	50.77%	30.61%	60.00%	27.12%	27.59%
	Should-Have	20.78%	27.69%	16.92%	16.33%	30.00%	50.85%	44.83%
	Could-Have	40.26%	27.69%	32.31%	53.06%	10.00%	22.03%	27.59%
	#Feature Req.	77	65	65	49	30	59	58
CP ranked Shortlists	1.	C++	C#	C#	C#	C#	Python	PHP
	2.	C	VB.net	PHP	-	Java	Rust	Python
	3.	Java	TypeScript	Python	-	-	Java	Java
	4.	Python	JavaScript	-	-	-	C#	C#
	5.	-	-	-	-	-	C++	-
DSS Solutions	1.	Java 99%	C# 99%	C# 99%	C# 99%	C# 100%	Java 90%	Java 94%
	2.	C# 92%	Java 99%	Python 99%	Python 99%	Java 100%	C# 89%	C# 93%
	3.	C++ 84%	Python 99%	Java 99%	Java 99%	Python 100%	Python 88%	PHP 92%
	4.	Python 84%	JavaScript 78%	PHP 85%	Object Pascal 87%	PHP 100%	PHP 84%	Python 91%
	5.	C 78%	VB.Net 72%	- -	Go 87%	- -	C++ 71%	JavaScript 75%
	6.	PHP 64%	TypeScript 65%	- -	- -	- -	JavaScript 65%	Ruby 72%

different application domains for increasing diversity in our evaluation, including control systems, content management systems (CMS), helpdesk systems, booking systems, financial systems, distributed ledgers, and search engines. Moreover, the selected case study companies were located in two different countries, namely Iran and the Netherlands.

The case study participants have identified a shortlist of ranked feasible programming languages (Table 5), as their potential solutions, for the **backend** of their projects through multiple internal expert meetings and investigation into programming languages before participating in this research. The experts at the case study companies specified their programming language feature requirements based on the MoSCoW prioritization technique (Table 4), so seven industry cases were defined and stored in the knowledge base of the DSS.[10] Next, the Inference Engine of the DSS generated feasible solutions for each case. The rest of the section describes the case study companies' ranked shortlists and analyzes the DSS outcomes.

4.1. Case study 1: Oceaneering

Oceaneering AGV Systems work on logistics and Automated Guided Vehicle (AGV) technology, widely used for transporting materials in industry and commerce. An AGV system is a portable robot that follows along marked long lines or wires on the floor or uses radio waves, vision cameras, magnets, or lasers for navigation.

One of the Oceaneering's branches is located in the Netherlands and is mainly active in developing, implementing, and marketing AGV Systems. They specialize in providing mission-critical mobile robotics solutions for material handling applications involving mixed fleets deployed globally in the automotive and manufacturing sectors.

The Oceaneering experts designed a centralized control system that collects data from several field devices and transmits control instructions. The system is responsible for fleet and traffic management and various logistics functions, including order fulfillment. The system is customized to meet specific requirements related to defining vehicle traffic rules, presenting performance data, and optimizing battery consumption.

[10] The industry cases are available on the DSS website: https://dss-mcdm.com.

4.1.1. Requirements

The case study participants defined the following subset of requirements of the control system (for more detail, see Table 4):

- The system is expected to coordinate multiple AGVs to guarantee that no collisions occur while tasks are performed. The system must dispatch AGVs so that the quickest cycle time is achieved. Accordingly, Socket programming (R04) and support threading (R05) are two Must-Have features from their perspective.
- The system's main architecture design is based on a centralized management system to coordinate AGVs from a single point. Thus, the potential programming languages must have support predefined Software Architecture Patterns (R22) and Design Patterns (R32). Moreover, supporting functional programming (R44), free implementations of the core libraries (R15), and a wide range of Package Managers (R23) facilitate the development phase of a centralized management system.
- A real-time or semi-real-time data processing unit is required for data streams, such as collection, classification, storage, and analysis of various event messages output from multiple sources (R20, R74, R94, and R75).
- A mobile app. can be used for real-time handling of on-site tasks related to composition, opening, malfunction, and tests for home site response (R39, R60, R81, and R82).
- The system architecture supports security monitoring and behavioral analysis. Additionally, it supports the development of data-driven systems based on collecting and processing security-related data to assess risks, identify and visualize threats, and produce alerts, among other security services (R13, R64, R43, and R61).
- As maintainability (R31), reusability (R26), and scalability (R06) were part of the quality concerns of the experts, they were looking for highly popular (R07) and mature programming languages (R08).

4.1.2. Results

The case study participants at Oceaneering have considered four potential programming languages (including *C++*, *C*, *Java*, and *Python*) to implement the centralized control system based on 77 programming

language feature requirements (see Table 5). Although more than 60% of the feature requirements for this case study were Could-Have and Should-Have features (soft constraints), a significant portion of their features prioritized as Must-Have features (almost 39%). Accordingly, the DSS excluded 41 infeasible solutions and ranked the rest of the programming languages (including *Java*, *C#*, *C++*, *Python*, *C*, and *PHP*) based on the soft constraints.

The experts were looking for programming languages that can be used in *Web Services* and *Web-Based Systems*, so they prioritized these features as Should-Have. The DSS suggested *Java* and *C#* as two better alternative solutions for these application domains. The experts at Oceaneering were looking for programming languages that can be employed in *Embedded Systems*, so they considered *C* and *C++* as two alternatives. As it was a Should-Have feature for them, the DSS did not exclude *PHP* as an infeasible solution but scored it lower than the other potential solutions (64%).

4.2. Case study 2: Author-it software corporation (ASC)

Author-it Software Corporation (ASC) is enterprise software for authoring, content management, publishing, and localization. ASC centralizes the content creation process, writing in components, and storing the pure content information in a database. Author-it supports the assembly and generation of this information into various documents to be published to an array of outputs. Author-it can be used for documenting Pharma & Biotech, Medtech, Technical Publications and Training & eLearning. One of the branches of ASC is located in the Netherlands.

The experts at this case study design and implement cloud-based component authoring solutions for collaborative content development and multi-channel publishing. The platform enables organizations to author, share, and reuse information across multiple forms of content for critical business needs. Author-it Honeycomb is the latest version of ASC's responsive *HTML5* output for delivering eLearning, mobile learning, and assessments on desktops, tablets, and smartphones.

4.2.1. Requirements

The experts at ASC specified the following subset of requirements of their system (for more detail, see Table 4):

- The system architecture is designed based on the service-oriented and architecture (R43) and object-oriented design (R02), so potential programming languages should support software architecture design patterns (R32) and predefined software architecture patterns (R22).
- The system is a Software-as-a-Service solution and enables content creation in a web browser (R61, R10, and R12).
- Author-it enables users to publish single-source content to other content formats, such as PDF, Word, and PPT (R86 and R87).
- Author-it supports translations with its Localization Manager (R67 and R70).
- The content manager has various editors, such as image and text editors (R65).
- The system supports permission management and version control (R93 and R24).
- Declarative programming (R52), Dynamic programming (R68), Imperative programming (R30), and Functional programming (R44) are essential programming paradigms for implementing the system.
- The case study participants were looking for languages that can be used to code, build, run, test, and debug software for cloud platforms, such as Amazon Web Services (R51).
- The system requires a highly integrated and consumerized user experience with consistency across all devices that can be provided by a web-based user interface (R12, R39, and R66).
- The popularity in the market (R07), Reusability (R26), Maturity level (R08), and Scalability (R06) are the main quality concerns of the experts at ASC when they want to select potential programming languages.

4.2.2. Results

The case study participants at ASC came up with *C#*, *VB.Net*, *TypeScript*, and *JavaScript* as four main programming language alternatives to building the content management system. The experts defined 65 programming language feature requirements and assigned Must-Have priority to almost 45% percent of them. The DSS results were similar to the case study participants and suggested *Java* and *Python* as two equal alternatives to *C#*.

The "popularity in the market", "Reusability", "Maturity level", and "Scalability" of the suggested solutions had differed from each other, so the DSS scored them differently. The experts were looking for programming languages that were employed mainly in *Cloud computing applications*. Accordingly, the DSS ranked *C#*, *Java*, and *Python* higher than *JavaScript*, *Visual Basic.net*, and *TypeScript*.

4.3. Case study 3: Dooman ltd.

Dooman ltd. is an Iranian software development company implementing and maintaining a help desk system called *Gamma*. Gamma is a suite of tools that enables organizations to provide information and support customers with concerns, complaints, or inquiries about their products or services. Gamma unifies queries from various customer-facing support channels, such as live chat, email integration, web contact forms, phone, mobile, and social media.

The experts at this case study company designed and implemented the system based on the Multitier and Model–View–Controller (MVC) software architecture patterns. Additionally, the system is deployed and maintained on a private cloud. The experts are mainly interested in the Microsoft ecosystem, so currently, Gamma is developed based on Microsoft technologies, such as *.NET Framework* and *SQL Server*.

4.3.1. Requirements

The experts at this company indicated the following subset of requirements of their system (for more detail, see Table 4):

- It should be possible for a user to log an incident by sending an email which will generate a new incident reference and be added to the queue. This will ensure that all questions sent by email to the Help Desk inbox, no matter how small, will be logged and tracked, and no one will have to monitor the Service Desk inbox manually (R13, R39, and R49).
- Gamma converts all emails from customers to tickets and facilitates ticket management (R37).
- Gamma automates the ticket assignment process and guarantees all customer requests will be replied to within one business day (R20).
- The system needs a reporting system to track team performance, customer satisfaction, and identify potential bottlenecks (R65).
- Gamma must quickly generate ad hoc reports, reporting on any fields, including text strings (R24).
- The system should be able to differentiate between (1) a new incident and (2) an update to an existing incident (e.g., by auto-detecting the incident reference number in the subject line/body of the email). If the latter, it should be added as a note to the relevant incident (R48).
- It must be possible for end-users to submit suggestions via the portal, which can then be reviewed and added to the frequently asked questions' list if appropriate (R12).
- The experts modeled their system based on object-oriented design, so Object-oriented programming is an essential paradigm for them (R02).
- Gamma needs to link an article in the knowledge base to an incident (e.g., to indicate that all actions in that article were attempted), possibly with a method to indicate with a single-click (e.g., tick/cross) which actions were successful/unsuccessful (R67 and R39).

- The system must be accessible through a dedicated web client (i.e., with no client software installed on the PC, even behind the scenes) (R10).
- Gamma must be integrated with Change Management software (for generating Change Requests, etc.) (R04).
- In order to implement their web-based solution, the experts preferred to employ popular and mature enough programming languages (R07 and R08).
- The experts believed that such programming languages have consistent, comprehensive documentation, and typically their communities are more friendly and accessible (R21 and R45).
- The experts mentioned that simplicity in writing, reading, and reusing codes are vital factors (R40, R41, R42).

4.3.2. Results

The case study participants selected *C#*, *PHP*, and *Python* as their potential solutions before participating in this research. Next, they identified 65 feature requirements based on the MoSCoW prioritization technique. More than half of those features have been prioritized as Must-Have features, so the DSS excluded 43 alternatives and suggested top-4 alternative solutions.

Table 5 shows that the DSS offered *Java* as an alternative that was scored equal to *C#* and *Python*. According to our research documenting analysis phase, we realized that *PHP* was not used as a tool to implement Commercial-Off-The-Shelf (COTS) components, and it cannot be used to develop MacOS-based applications. As both of these features prioritized as Could-Have, the DSS did not exclude *PHP* from the set of feasible solutions; however, *PHP* gained the lowest score among the other feasible solutions.

4.4. Case study 4: Saanaa DP

Saanaa DP is specialized in designing and building web-based systems for a variety of customers in Iran. The experts at Saanaa DP mainly determine their clients' requirements and goals and then provide an estimate of the cost to create web applications. They are also active in hosting their websites and debugging any problems. They re-design websites for pre-existing clients, as well as new clients. One of their customers requested an online ticket reservation and hotel booking system, so they looked for the best fitting programming language to implement a web-based solution.

The backbone of the booking system architecture is designed based on the MVC and Client-Server software architecture patterns. Typically, the experts at Saanaa DP deploy their web applications on public cloud providers. The majority of their websites are currently developed based on Microsoft technology, so their initial choice for selecting a programming language ecosystem is *C#.net*.

4.4.1. Requirements

The experts at Saanaa DP defined the following subset of requirements for the booking system (for more detail, see Table 4):

- The booking system needs to have a user-friendly interface (R66 and R39).
- The system should show up-to-date availability and immediate price quotations, request any information on the booking form, handle cancellations, modifications, and set up automatic confirmations (R24 and R20).
- The system should be able to handle the creation and delivery of invoices to clients. Additionally, the system should accept deposits or full payments, whether optional or obligatory, processed securely by one of the partner payment gateways (R43 and R50).
- The system must send a booking confirmation email after successful payment (R13).
- The case study participants stated that object-oriented programming (OOP) and data-driven programming are mainly considered as programming paradigms at their company to model ticket reservation systems (R02 and R64).
- The potential programming languages for implementing the system have to support multithreading to simultaneously handle multiple tasks (R05).
- The system should support almost all popular web browsers, such as Internet Explorer, Safari, Chrome, and Firefox (R39, R19, and R29).
- The case study participants preferred to select programming languages that have been employed to implement web-based and transaction processing systems (R12 and R50).
- The potential programming languages must be mature enough and trendy in the market because they have comprehensive documentation and friendly communities (R07, R08, R21, and R45).

4.4.2. Results

As the software engineers of Saanaa DP depend heavily on Microsoft technology, they have selected *C#* as their primary choice for their customers. The case study participants defined 49 feature requirements and prioritized almost 70% of them as soft constraints (Should-Have and Could-Have features). Thus, the DSS had to suggest more feasible solutions; however, they assigned the Must-Have priority to several particular features, such as web-based systems and supporting predefined software architecture patterns supported by a limited list of programming languages (see Table 4).

The DSS offered *C#*, *Python*, and *Java* as three almost equal alternatives, besides *Object Pascal* and *Go* as two potential solutions with lower scores. During the data collection phase to build the decision model, we did not find any evidence that shows *Go* and *Object Pascal* can be employed in *transaction-based systems (banking Systems)*.

4.5. Case study 5: FinanceComp

FinanceComp is an Iranian financial institution that provides personal loans, commercial loans, and mortgage loans; moreover, it allows financial transactions at its branches. Additionally, it provides an internet banking system to help customers view and operate their respective accounts through the internet.

The software architecture of the system is based on the Multitier and Client-Server architecture patterns. Security is one of the main quality concerns of the software engineers in the case study; accordingly, they employed multi-tenant databases to store financial transactions. Furthermore, the system is deployed and maintained on a private cloud.

4.5.1. Requirements

The experts at FinanceComp defined the following subset of requirements for the system (for more detail, see Table 4):

- Standard data classifications (definition and formats) should be established and used for recording financial events (R50).
- Internal controls over data entry, transaction processing, and reporting should be applied consistently (R20 and R50).
- The system must provide timely and useful financial reports to support managers (R13).
- The system should define, maintain and execute the posting and editing rules for processed transactions (R50 and R44).
- The system should provide and maintain online queries and reports on balances separately for the current and prior months (R50).
- The case study participants highlighted that *Object-oriented programming* and *Event-driven programming* are the essential programming paradigms that they have used to model the system to incorporate the advantages of modularity and reusability (R02 and R20).

- The system should generate an audit log that identifies all document additions, changes, approvals, and deletions by users (R18).
- The experts mentioned that the potential programming languages have to support *socket programming*, enabling them to exchange information between processes across the network (R04).
- The programming languages should facilitate their software architecture implementation and assist them with employing software architecture design patterns (R22 and R32).
- *Maintainability*, *scalability*, and *security* were part of their quality concerns, so that they preferred to hire highly mature programming languages (R08, R06, and R31).
- The developers' availability, besides their current programming knowledge and experience, is a vital factor that has profoundly impacted their decision-making process (R35).

4.5.2. Results

The case study participants at FinanceComp stated that their system's former implementation was based on *Java*, and now they want to consider *C#* as another alternative solution for implementing the system. They defined 30 programming language feature requirements and assigned Must-Have priority to 60% percent of them. Thus, they were looking for a limited set of programming languages with unique capabilities.

The complexity of programming languages was not an issue for them; however, they wanted to select a language that can hire enough senior developers to work with. Finally, the DSS concluded that *C#*, *Java*, *Python*, and *PHP* could be used as the most suitable alternatives to their case.

4.6. Case study 6: SecureSECO

SecureSECO is a research organization in the Netherlands. The researchers at SecureSECO work in close collaboration with Utrecht University and the Delft University of Technology. The goal of the research group is to secure and increase trust in the software ecosystem by using distributed ledger technology and empirical software engineering research.

The Software Heritage Graph (SHG) is a database that contains 8 billion source code files that have been collected from the worldwide software ecosystem. This archive is a treasure trove, but it is a big challenge to extract value from the SHG. The researchers at SecureSECO propose the SearchSECO, a hash-based index for code fragments that enables searching source code at the method level in the worldwide software ecosystem. They want to create a set of parsers that extract fragments (methods) from the code files and make them findable.

4.6.1. Requirements

The experts at SecureSECO defined the following subset of requirements for developing the SearchSECO (for more detail, see Table 4):

- An extensible data structure, a meta-model representing the relevant entities for SecureSECO, is needed (R55).
- The system needs a parsing and extraction architecture that enables the rapid extraction of code file methods, including the call graph in a project (R67).
- The system will initially use parsers for *Java*, *C*, *JavaScript*, and *Python* (R44).
- SearchSECO requires a project data extractor to collect data about the code fragments, such as author data and version data (R24).
- The system needs several repository spiders that collect source code and project data from different repositories (R43).
- SearchSECO has multiple generic extraction techniques to extract code from languages for which it does not have parsers available individually (R63).
- The system needs smart hashing techniques for hashing the abstract syntax tree of code clones and for hashing code fragment meta-data (R64).
- SearchSECO requires a search API that enables one to search through the SecureSECO ledger rapidly (R59).
- a Job distribution architecture, Worker nodes that can perform the SecureSECO maintenance jobs, is required (R13 and R59).
- SearchSECO needs a data collection dashboard that shows the current state of the SearchSECO platform's growth (R71 and R20).

4.6.2. Results

Recently, the SecureSECO organization experts designed the SearchSECO architecture, so it has not been implemented yet. The case study participants identified 59 programming language feature requirements and prioritized more than 70% of them as soft constraints (Could-Have and Should-Have) features based on their assumptions at the current stage of the software development life cycle. Accordingly, they have not limited themselves to a specific technology or third-party vendor. The SecureSECO experts indicated *Python*, *Rust*, *Java*, *C#*, and *C++* as their top-5 potential programming languages.

We did not find any evidence showing that *Rust* supports a toolchain and a code coverage tool during this study's data collection phase. Accordingly, the DSS excluded *Rust* from the ranked shortlist of solutions because the case study participants prioritized these two features as two Must-Have feature requirements. Based on the feature requirements, the DSS suggested to employ *Java*, *C#*, *Python*, *PHP*, *C++*, and *JavaScript* as potential solutions. As aforementioned, each software-intensive project can be implemented by employing a set of programming languages; for instance, the case study participants, after getting feedback from us, asserted that SearchSECO could be developed using a combination of *Python* and *JavaScript* programming languages.

4.7. Case study 7: ENVRI-FAIR

The Environmental Research Infrastructure (ENVRI) community is a community of Environmental Research Infrastructures, projects, networks, and other diverse stakeholders interested in environmental Research Infrastructure matters. The overarching goal of ENVRI-FAIR is for all participating Research Infrastructures to improve their FAIRness and prepare the connection of their data repositories and services to the European Open Science Cloud. With the development of FAIR implementations from the participating Research Infrastructures and integrated services among the environmental subdomains, these data and services will be brought together at a higher level (for the entire cluster), providing more efficient researchers' services and policymakers.

One of the deliverable projects in ENVRI FAIR context is Open Semantic Search Engine (OSSE), which is a platform for building own Search Engine, Explorer for Discovery of extensive document collections based on Apache Solr or Elasticsearch open-source enterprise-search and Open Standards for Linked Data, Semantic Web, and Linked Open Data integration. In this case study, the experts were interested in various OSSE functionality, such as importing and indexing linked data from semantic knowledge graphs for full-text search and faceted search.

4.7.1. Requirements

The experts of the ENVRI-FAIR project defined the following subset of requirements for developing the OSSE (for more detail, see Table 4):

- As the most common type for knowledge storage, representation, reasoning, RDF's support is the core requirement in the design and development of the knowledge base of the search engine. This requirement can include the following specific options, such as RDF import/export, RDF storage, owl import, and SPARQL support (R10, R24, R30, and R70).

– An interface for search and discovery of knowledge base content should be provided. This could be the conventional keyword-based search or faceted search. Rather than strict adherence to a single controlled vocabulary or keyword set, a semantic search function is further expected to permit search based on 'similar' or 'related' terms (R39 and R61).
– Due to the variance of source types in the ENVRI community, various methods should be supported for knowledge acquisition, like form-based manual RDF ingestion, Questionnaire-based RDF triple generation, existing RDF integration, structured and unstructured information transformation, etc. Specific measures should be considered to facilitate non-technical users adding knowledge in a straightforward way (R48, R59, and R87).
– A graph/network analysis view can provide a visualization of the direct and indirect relations, connections, and networks between named entities like persons, organizations, or main concepts which occur together in your content, data sources, and documents or are connected in your Linked Data Knowledge Graph (R65, R66, and R19).
– Considering the typical case where multiple users contribute to the knowledge base, provenance is of fundamental importance. This primarily refers to tracking individual additions, deletions, and updates and their administration, i.e., approval, rejection, reversion (R05 and R20).

4.7.2. Results

The case study participants stated that after analyzing the requirements of the OSSE, they investigated the potential open-source tools that can be used and customized to meet the requirements. After performing an extensive evaluation, they selected an open-source tool called open semantic search. The backend of the tool was implemented in *PHP* and *Python*. Accordingly, the first two solutions for the case study participants were these programming languages. However, they assumed that *Java* and *C#* could be employed in developing additional components of the OSSE.

More than 70% of the feature requirements were prioritized as soft constraint features so that the DSS could offer a broader list of alternative solutions. The DSS results showed that besides the case study participants' solutions, *JavaScript* and *Ruby* could be considered as two more options. Scalability, popularity, and maturity of the programming languages were prioritized as three Must-Have features, so the DSS prioritized *Java* and *C#* higher than the offered solutions.

5. Analysis of the results

The validity metric is defined as the degree to which an artifact works correctly. There are two ways to measure validity: (1) the results of the DSS compared to the predefined case-study participant shortlist of potentially feasible programming languages, and (2) according to the domain experts' opinion.

Concerning effectiveness, the case study participants asserted that the updated and validated version of the decision model is useful and valuable in finding the shortlist of feasible programming languages. Moreover, the DSS reduces the time and cost of the decision-making process. The case study participants expressed that the DSS enabled them to meet more detailed programming language feature requirements. Furthermore, they were surprised to find their primary concerns, especially when different experts' opinions are combined.

The DSS suggests that *C#*, *Java*, and *Python* can be feasible solutions for all seven case studies (see Table 5), which means that these programming languages support all of the features with *Must-have* priority. It makes sense as these programming languages are in the top-5 list of popular solutions in the market (see Table 3); moreover, their maturity levels are relatively high, as they support most of the programming language features that we have considered in this study (see Tables 2).

Scalability and maturity of the programming languages were two key quality concerns of the case study participants (see Table 4) so that they considered at least one of the top-5 programming languages as their potential solutions. Table 5 represents that the DSS can come up with more feasible programming languages than human experts.

Table 4 shows that supporting *Debugger*, *Object-Oriented Programming*, *Testing tools*, *Socket programming, threading*, *Scalability*, *Open source compiler or Interpreter*, *General-Purpose PL*, *Web-Based Systems*, *Web Services*, *Free implementation of the core libraries*, *Event-driven programming*, *Software Architecture Patterns*, *Software Architecture Design Patterns*, and *Object-Relational Mapping (ORM)* were programming language features that all of the case studies assigned priorities to them and defined them as their programming language feature requirements. All of the case study participants somehow declared that the Object-Oriented Programming paradigm leads to cheaper and faster development. They mainly preferred to employ a programming language, or a set of languages, that supports debugging, tracking, and testing tools.

It is not surprising that *socket programming* and *threading* were prioritized as two essential features, as all of the case studies were mainly involved with network programming and web-based applications. In other words, Client-Server was one of the software architecture patterns of the backbone of their systems.

Easy to reuse and read existing code and write new code have considered three programming language features in the decision model. These features were prioritized as Must-Have and Should-Have features at four case study companies so that the case study participants were looking for programming languages that facilitate the development process and increase the reusability and readability of their source code.

Besides the free implementation of their core libraries, open-source programming languages, as two programming language features, arouse almost all of the case study participants' attention. We realized that the development teams' budget at the case study companies was impactful on their decisions to select either licensed or open-source programming languages. In other words, the budget constraint can be led to selecting the cheapest feasible solution as the best fitting solution.

Table 5 shows that the case study participants who indicated the feature requirement with more confidence were advised a limited set of alternative solutions. Hence, the higher number of hard-constrained feature requirements (Must-Have) on unique programming language features leads to fewer alternative solutions. For instance, *Saanaa DP* and *FinanceComp* have prioritized their feature requirements according to their current main solutions (*C# and Java*), so they have assigned Must-Have priority to the particular features, such as Supporting *Web Services*, *Windows platform*, and *Socket programming*. In other words, their feature requirements were biased to the features that their shortlist of programming languages supported them. Thus, the DSS results did not surprise them.

The results show that flexibility on the feature requirements leads to a higher number of alternative solutions. For instance, the DSS suggested a broader list of alternative solutions to the case study participants at *SecureSECO* and *ENVRI-FAIR* as they did not emphasize particular feature requirements and defined more soft-constrained (Should-Have and Could-Have) features.

The decision-makers have a different perspective on their feature requirements in the different software development life cycle phases. Typically, they consider generic features in the early phases of the life cycle (just like the case study participants from *SecureSECO* and *ENVRI-FAIR*). In contrast, they are interested in more technical features as their development process matures (similar to the case study participants at *Saanaa DP* and *FinanceComp*).

6. Discussion

This section highlights the case study participants' and experts' viewpoints on the decision model. Moreover, it explains the lessons learned and our observations while researching to build and evaluate the decision model. Finally, this section discusses the limitations and threats to the validity of this study.

6.1. Case study participants

Almost all of the case study participants asserted that for evaluating a programming language from its usability point of view, they should be familiar with the language first, essentially when it is a cutting-edge programming language. It is a time-consuming process depending on the language's complexity, making it more challenging to evaluate. Thus, they require technical knowledge to comprehend the programming language and its features to create suitable test setups and compare them with other potential solutions. Typically, they need to focus on particular parts of the programming language — assessing the entire language at once is nearly impossible because it would take much time. Additionally, to evaluate the programming language, the case study company's experts need to answer the following question: which programming language features are the most important ones? Can a language's complexity be evaluated in a questionnaire or a usability test, or do we need test users evaluating the language over a more extended period? What are the right set of criteria to declare a language is usable? How can we assume a language is *better* than another one? What should be measured to find this out? Several measurements could be more practical, such as learnability, understandability, or consistency — What is most substantial? The case study participants confirmed that the decision model, including its programming language features and languages, can address such questions and reduces the decision-making process's time and cost.

One of the participants mentioned that *in contrast to licensed programming languages, in which programmers are limited to use them for a few days only, open-source programming languages are available to evaluate before actual implementation. However, after selecting an open-source programming language as the primary language, sometimes multiple modifications are made in the source code of an open-source programming language by unknown developers. Eventually, this leaves programmers questioning the current version of the code they are using. In the case of licensed programming languages, the changes are made systematically by authorized developers of the source code, and they notify the version of the code.* Accordingly, a development team should be able to organize the source code and increase its reusability.

One of the case participants stated that *code reuse is the practice of using existing code for a new function or component. The existing code may be reused to perform the same function or to do a similar but slightly modified function providing for efficiencies, cost savings, and improved overall quality. In order to reuse code, that code should be high-quality, so it should be secure and reliable. Code reusable in practice means that developers have to build libraries that other projects requiring that same functionality can utilize. So the developers should identify the core competence of each module.*

Biases, such as motivational and cognitive [52], arise because of shortcuts or heuristics that decision-makers use to solve problems and perform tasks. The Hawthorne effect [53], which is the tendency for decision-makers to change their behavior when observed, is a form of cognitive bias. The case study participants might have been more careful in the observational setting than in the real setting because they are being observed by scientists judging their selected programming language feature requirements and priorities. Moreover, the Bandwagon effect [54], which is the tendency to do or believe things because many other decision-makers do or believe the same, is another form of cognitive bias. The Bandwagon effect typically shows up in group decisions. To mitigate the Hawthorne and Bandwagon effects, individual and group interviews have been conducted.

6.2. Experts

One of the experts asserted that *some programming languages are certainly better options for solving different problems, making the selection process more straightforward. For instance, if you are integrating heavily with Microsoft products, then you have to consider .NET on your list of potential alternatives. If you perform functions, such as scientific modeling, Python can be a better choice because of its well-defined math libraries and the ability to scale well with Hadoop. If enterprise-level security for a business-to-business application is critical, then Java can be the best fitting language. Thus, some factors, including the problem domain, business case, and the types of customers, are impactful in the evaluation process.*

The experts expressed that programming languages' supported programming language features play a significant role in the programming language selection process. For instance, one of the experts expressed that *we are a developer-centric company and need a comprehensive set of software development kits for various well-known programming languages, including Java, Python, Ruby, and PHP. Each programming language has different features, communities, support, and ecosystems to consider when making your choice.*

Almost all of the experts mentioned that their companies continuously improve and reevaluate their technologies, including the used programming languages. They mainly consider a limited set of programming language features and languages in their selection process; thus, a decision model, such as the one in this study, can ease their evaluation process.

6.3. Lessons learned

General-purpose programming languages, such as *C#* and *Java*, are nearly at the same level of maturity and support almost the same feature set, so programming language selection is majorly concerned with the ecosystem, the community, and the availability of programmers. The ecosystem can be considered as libraries, tools, and frameworks that support programming languages. The community is typically the one who maintains the ecosystem. Even when backed by a company, the community is the main drive for improvements, often implementing them. The availability of programmers is self-explanatory. Some programming languages, such as *Haskell*, are appreciated in theory, but fewer programmers are available than *Java*.

When a software development company selects a popular programming language, it can count on sourcing numerous well-qualified developers who are available to work at the company. Such programming languages are surrounded by huge communities that provide developers with samples and solutions to solve critical tasks and problems much quicker.

Experience in using technology provides invaluable knowledge when selecting suitable technology. In other words, software engineers typically prefer to select the programming languages they have employed before and have some experience. The main factor is the cost of adding a new programming language. Hiring new developers, changing the infrastructure, and learning the best practices are costly for many companies. Therefore, the answer to the best programming language question usually is what the company was already using. There are many risks associated with this, as an organization could be stuck in a legacy technology for which there is no longer a demand [55]. It is thereby advisable for these companies to primarily use the DSS to avoid innovation stasis.

For small projects, selecting a programming language is much faster and more straightforward, as the brevity of the lines of code is essential. In other words, software engineers prefer to select a programming language requiring fewer code lines to develop the project in such projects. The intention is to get the solution out first, next to be worried about the performance. However, for large organizational projects, programming language selection is a different story. Various development teams will develop different modules and services expected to interact and interconnect with another to address a particular problem. In this case, the programming language selection might involve the high level of portability of the language to run on different platforms or the exchangeability of data and information.

6.4. The decision model

The case study participants confirm that the DSS provides programming languages to help software development companies in their initial decisions for selecting programming languages. In other words, the DSS recommended nearly the same programming languages as the case study participants suggested to their companies after extensive analysis and discussions. However, the DSS offers a shortlist of feasible programming languages; therefore, software development companies should perform further investigations, such as performance testing, to find the best fitting programming language for their software products.

The case study participants confirm that the updated and validated version of the DSS is useful and valuable in finding the shortlist of feasible solutions. Finally, it reduces the time and cost of the decision-making process. Our website[11] is up and running to keep the decision support system's knowledge base up-to-date and valid. The supported programming language features are going to change due to technological advances. As such, the decision model must be updated regularly. We envision a community of users of the DSS who maintain and curate the system's knowledge and consider building such a community as future work.

Decision support systems can be employed to make decisions quicker and more efficiently; however, they suffer from adoption problems [56]. A DSS supports rational decision-making by recommending alternative solutions basis the objectivity. Although limited rationality plays a crucial role in a decision-making process, subjectivity should not be discarded. A DSS promotes objectivity and dismisses subjectivity, which can have a drastic consequence on the decisions' reliability.

The DSS provides a discussion and negotiation platform to enable requirement engineers to make group decisions. It detects and highlights the conflicts in the assigned priorities to decision-makers' programming language feature requirements and asks them to resolve disagreements. Thus, the DSS supports requirements engineers in the requirements verification and validation activity by avoiding conflict between programming language feature requirements and generating feasible solutions according to the programming language feature requirements. Moreover, the DSS can be considered as a communication tool among the decision-makers to facilitate the requirements specification activity [17].

We have recorded the times it took to collect the data about the different programming languages with the goal of showing the efficiency gain for programming language selection projects. However, this data foregoes the inherent efficiency gain from doing the data collection work for multiple selection projects. The actual performance indicator for our project is not the number of programming languages in our system or the number of features but instead the number of projects done using the DSS. For this reason, we are anonymously tracking end-user behavior on the system. For instance, we observed that in a one and half year time window (from November 2019 to April 2020), after creating the decision mode, 524 users visited the decision model within the DSS and then downloaded the collected data, such as the lists of the programming languages, the features and their definitions, and the mappings. Additionally, 23 users as the decision-makers at different software development companies employed the decision model in their decision-making process to compare their shortlists of feasible solutions with the DSS suggestions.[12]

It is essential to highlight that sometimes unpopular programming languages have unique features that the popular ones do not support. For instance, *OptiML* is an embedded domain-specific language for machine learning. *OptiML* enables software developers to run statistical inference algorithms expressible by the statistical query model to be easy to express and execute quickly [57]. Alternatively, there are many command-line tools on UNIX-like operating systems (such as Linux, Mac, and BSD), each one accepting instructions in their format. This format can be considered a domain-specific language that allows defining the tasks to be executed. For example, *SED* executes the text transformations indicated using its domain-specific language [58]. Consequently, we need a decision model for selecting such programming languages. We believe that the decision model can be extended in the future to consider such programming languages as its alternative solutions.

6.5. Limitations and threats to validity

The validity assessment is an essential part of any empirical study. Validity discussions typically involve Construct Validity, Internal Validity, External Validity, and Conclusion Validity.

Construct validity refers to whether an accurate operational measure or test has been used for the concepts being studied. To mitigate the threats to the construct validity, we followed the MCDM theory and the six-step of a decision-making process [27] to build the decision model for the programming language selection problem. Moreover, we employed document analysis and expert interviews to capture knowledge regarding programming languages as two different knowledge acquisition techniques. Additionally, the DSS and the decision model have been evaluated through seven real-world case studies at seven different real-world software development companies in the Netherlands and Iran.

A challenge for this study is that the qualities and features that we have identified with the support of twelve experts can vary wildly with the expert's perception. Some of the experts, for instance, indicated that most features could quickly be built using the language, although it is not necessarily included in the language's standard libraries. While we are convinced that the twelve experts have added a significant amount of extra knowledge to the model, one might argue we need a large number of experts per programming language to reach a consensus on each feature. We should also be aware of the strong opinions surrounding programming languages, making it somewhat more complicated to find consensus in the data. A potential solution to this validity threat is the introduction of the community, as mentioned earlier.

Internal validity attempts to verify claims about the cause–effect relationships within the context of a study. In other words, it determines whether the study is sound or not. To mitigate the threats to the decision model's internal validity, we define DSS success when it, in part, aligns with the case study participants' shortlist and when it provides new suggestions that are identified as being of interest to the case study participants. Emphasis on the case study participants' opinion as a measurement instrument is risky, as they may not have sufficient knowledge to make a valid judgment. We counter this risk by conducting more than one case study, assuming that the case study participants are handling their interest and applying the DSS to other problem domains, where we find similar results [13–15,17–19].

External validity concerns the domain to which the research findings can be generalized. External validity is sometimes used interchangeably with generalizability (feasibility of applying the results to other research settings). We evaluated the decision model in the context of Dutch enterprises. To mitigate threats to the research's external validity, we captured knowledge from different knowledge sources without any regional limitations to define the constructs and build the decision model. Accordingly, we hypothesize that the decision model can be generalized to all software development companies worldwide who face uncertainty in the programming language selection problem. Another question is whether the framework and the DSS can be applied to other problem domains as well. The problem domains [13,17–19] were selected opportunistically and pragmatically, but we are convinced that there are still many decision problems to which the framework and the DSS can be applied. The categories of problems to which the framework

[11] https://dss-mcdm.com

[12] Note, the results of our analysis, besides the instances of the decision model in the form of XML files, are available on GitHub [91].

and the DSS can be applied successfully can be summed up as follows: (1) the problem regards a technology decision in system design with long-lasting consequences, (2) there is copious scientific, industry, and informal knowledge publicly available to software engineers, and (3) the (team of) software engineer(s) is not knowledgeable in the field but very knowledgeable about the system requirements. We believe that the framework can be employed as a guideline to build decision models for MCDM problems in software production.

Conclusion validity verifies whether the methods of a study, such as the data collection method, can be reproduced, with similar results. We captured knowledge systematically from the sources of knowledge following the MCDM framework [13]. The accuracy of the extracted knowledge was guaranteed through the protocols that were developed to define the knowledge extraction strategy and format (See Appendix). A review protocol was proposed and applied by multiple research assistants, including bachelor and master students, to mitigate the research's conclusion validity threats. By following the framework and the protocols, we keep consistency in the knowledge extraction process and check whether the acquired knowledge addresses the research questions. Moreover, we crosschecked the captured knowledge to assess the results' quality, and we had at least two assistants extracting data independently.

7. Related work

Decision analysis, which is the study of decision-making for problems with multiple objectives, has been developed and widely employed in solving complex decision-making problems. In literature, decision-making is typically defined as a process or a set of ordered activities concerning stages of problem identifying, data collection, defining alternatives, selecting a shortlist of alternatives as feasible solutions with the ranked preferences. Fitzgerald et al. [39] define decision-making as a process that consolidates critical assessment of evidence and a structured process that requires time and conscious effort. Kaufmann et al. [40] state that the decision-making process encourages decision-makers to establish relevant decision criteria, recognize a comprehensive collection of alternatives, and assess the alternatives accurately. Over the past few years, various methods and underlying theories have been introduced for solving decision-making problems in software production, such as programming language selection.

In this research, Snowballing was the primary method to investigate the existing literature regarding techniques that address the programming language selection problem. Table 6 summarizes a subset of selected studies that discuss the problem.

7.1. Benchmarking and statistical analysis

Some studies employed Benchmarking and Statistical Analysis to evaluate and compare a collection of programming languages against each other in literature. For instance, Meyerovich et al. [6] conducted survey research to identify the factors that lead to language adoption. They concluded that only a limited number of programming languages were used for most applications, but the programming market supports many programming languages with niche user bases. Furthermore, essential programming language features have only secondary importance in adoption. Open-source libraries, existing code, and experience strongly influence developers when selecting a project's programming language.

Holtz and Rasdorf [10] introduced and discussed various attributes of programming languages that can positively or negatively affect the computer-aided design and computer-aided engineering software. Four programming languages, *Fortran*, *C*, *Pascal*, and *Modula-2*, were compared using the attributes.

Batdalov et al. [9] introduced a methodology for quantifying the impact of programming language on software quality and developer productivity. They formulated four hypotheses that investigated whether using *C++* leads to better software than using *C*. Then, they tested their hypotheses on large data sets to ensure statistically significant results.

Mannila and Raadt [67] suggested a set of criteria, including learnability, suitability, and availability, of programming languages. Next, they compared eleven programming languages (including *Eiffel*, *Haskell*, *Java*, *JavaScript*, *Logo*, *Pascal*, *Python*, and *Scheme*) according to the criteria. Finally, they assigned scores to the languages based on the number of criteria that they support.

Ray et al. [64] collected an extensive data set from GitHub to study the effect of programming language features such as static versus dynamic typing, strong versus weak typing on software quality. By triangulating findings from different methods and controlling for confounding effects such as team size, project size, and project history, they reported that programming language design does have a significant but modest effect on software quality.

Bissyande et al. [66] investigated a large number of open-source projects from GitHub to measure the *popularity*, *interoperability* and *impact* of various programming languages in terms of lines of code, development teams, issues, etc.

Feraud and Galland [62] compared five agent-based programming languages according to a number of criteria, such as Code extensibility and Debugging tools.

Costanza et al. [60] performed performance testing to analyze and compare the performance of three programming languages (*Go*, *Java*, and *C++*). Based on their benchmark results, the authors selected *Go* as their implementation tool and recommended considering *Go* as a valid candidate for developing other bioinformatics applications.

Studies based on benchmarking and statistical analysis are typically time-consuming approaches and mainly applicable to a limited set of alternatives and criteria, as they require a thorough knowledge of programming languages and concepts. Decision-making based on such analysis can be challenging as decision-makers cannot assess all their requirements and preferences at the same time, especially when the number of requirements and alternatives is significantly high. Furthermore, benchmarking and statistical analysis are likely to become outdated soon and should be kept up to date continuously, which involves a high-cost process.

7.2. MCDM approaches

As aforementioned, selecting the best fitting programming language(s) for a software project is a decision-making process that evaluates several alternatives and criteria. The selected programming language(s) should address the concerns and priorities of the decision-makers.

Conversely to MCDM approaches, studies based on *Benchmarking* and *Statistical Analysis* principally offer generic results and comparisons and do not consider individual decision-maker needs and preferences.

The tools and techniques based on MCDM are mathematical decision models aggregating criteria, points of view, or features [47]. Support is a fundamental concept in MCDM, indicating decision models are not developed following a process where the decision maker's role is passive [48]. Alternatively, an iterative process is applied to analyze decision-makers' priorities and describe them as consistently as possible in a suitable decision model. This iterative and interactive modeling procedure forms the underlying principle of decision support tendency of MCDM, and it is one of the main distinguishing characteristics of the MCDM as opposed to statistical and optimization decision-making approaches [70].

A variety of MCDM approaches have been introduced by researchers recently. A subset of selected MCDM methods is presented as follows: The *Analytic Hierarchy Process (AHP)* is a structured and well-known method for organizing and analyzing MCDM problems based on mathematics and psychology. Parker et al. [68] presented a set of criteria for selecting a programming language for use in an introductory programming course. Next, they applied the AHP to evaluate the

Table 6
Compares a subset of selected studies from the literature that addresses the programming language selection problem. The first and second columns (Studies and Years) refer to the considered studies and their publication years. The third column (DMA) indicates the decision-making approach that the studies have employed to address the problem. The fourth column (MCDM) denotes whether the corresponding decision-making technique is an MCDM approach. The fifth column (PC) indicates whether the MCDM approach applied pairwise comparison as a weight calculation method or not. The sixth column (QA) determines the type of quality attributes. The seventh and eighth columns (#C and #A) signify the number of criteria and alternatives considered in the selected studies. The following three columns indicate the numbers of common quality attributes (#CQ), features (#CF), and alternatives (#CA) of this study (the first row) with the selected studies. The last column (Cov.) shows the percentage of the coverage of the considered criteria (quality attributes and features).

Studies	Years	DMA	MCDM	PC	QA	#C	#A	#CQ	#CF	#CA	Cov.
This study		DSS	Yes	No	ISO/IEC 25010 EX. ISO/IEC 9216	164	47	57	107	47	100%
[59]	2020	Fuzzy Logic	Yes	No	Domain Specific	7	8	5	2	7	100%
[60]	2019	Benchmarking	No	N/A	Domain Specific	2	3	2	0	3	100%
[61]	2018	TOPSIS	Yes	Yes	Domain Specific	18	4	8	8	4	89%
[62]	2017	Benchmarking	No	N/A	Domain Specific	14	5	5	8	0	92%
[63]	2016	FDM	Yes	No	Domain Specific	29	6	3	12	3	52%
[64]	2014	Statistical Analysis	No	N/A	Domain Specific	26	17	2	12	17	82%
[65]	2014	FAHP	Yes	Yes	Domain Specific	8	5	3	3	5	75%
[6]	2013	Statistical Analysis	No	N/A	Domain Specific	14	33	3	7	26	71%
[66]	2013	Statistical Analysis	No	N/A	Domain Specific	3	30	2	1	17	100%
[9]	2011	Statistical Analysis	No	N/A	Domain Specific	4	2	2	0	2	50%
[67]	2006	Benchmarking	No	N/A	Domain Specific	17	11	9	7	9	94%
[68]	2006	AHP	Yes	Yes	Domain Specific	23	7	9	12	7	91%
[69]	2005	Fuzzy Logic	Yes	No	Domain Specific	36	3	9	2	1	31%
[10]	1988	Benchmarking	No	N/A	Domain Specific	23	4	18	3	3	91%

seven programming languages they considered potential alternatives. This MCDM approach considers a hierarchical structure of objectives, criteria, and alternatives to make complex decisions.

The *Technique for Order Preference by Similarity to Ideal Solution (TOPSIS)* is an MCDM approach that employs information entropy to assess alternatives.

Yıldızbaşı et al. [61] employed the TOPSIS method to evaluate a set of programming languages based on seven criteria such as ease of use and ability of programming languages. This approach aims to come up with an ideal solution and a negative ideal solution and then identify a scenario that is nearest to the ideal solution and farthest from the negative ideal solution.

Fuzzy logic is an approach to computing based on *degrees of truth* rather than the usual Boolean logic. Cochran et al. [69] and Mishra et al. [59] employed fuzzy set theory and fuzzy operations to address the programming language selection problem using based on weighted scores. Sometimes combinations of fuzzy logic with other MCDM approaches, such as AHP, are employed to solve MCDM problems. For instance, Rouyendegh [65] introduced a FAHP-based approach to evaluate five object-oriented programming languages against eight decision criteria.

The *Fuzzy Delphi Method (FDM)* is a more advanced version of the Delphi Method in that it utilizes triangulation statistics to determine the distance between the levels of consensus within the expert panel. Yoon et al. [63] identified a set of key factors for educational programming language selection and then applied the Delphi method based on a 20-expert panel to evaluate six programming languages.

The majority of the MCDM techniques in literature define domain-specific quality attributes to evaluate the alternatives. Such studies are mainly appropriate for specific case studies. Furthermore, the results of MCDM approaches are valid for a specified period; therefore, the results of such studies, by programming language advances, will be outdated. Note that, in our proposal, this is also a challenge, and we propose a solution for keeping the knowledge base up to date in Section 6.

Additionally, the pairwise comparison is typically considered as the main method to assess the weight of criteria in MCDM techniques. For a problem with n number of criteria $\frac{n(n-1)}{2}$ number of comparisons are needed [71]. It means that the pairwise comparison is a time-consuming process and gets exponentially more complicated as the number of criteria increases [72]. A subset of MCDM approaches, such as TOPSIS and AHP, are not scalable [73,74], so in modifying the list of alternatives or criteria, the whole process of evaluation should be redone. Therefore, these methods are costly and applicable to only a small number of criteria and alternatives. In this study, we have considered 164 criteria and 47 alternatives to building a decision model for the programming language selection problem.

In contrast to the named approaches, the cost of creating, evaluating, and applying the proposed decision model is not penalized exponentially by the number of criteria and alternatives because it is an evolvable and expandable approach that splits down the decision-making process into four maintainable phases [18]. Moreover, we introduce several parameters to measure non-Boolean criteria' values, e.g., the maturity level and popularity of programming languages. The proposed decision model addresses the main knowledge management issues, such as capturing, sharing, and maintaining knowledge. Moreover, it uses the ISO/IEC 25010 [50] as a standard set of quality attributes. This quality standard is a domain-independent software quality model and provides reference points by defining a top-down standard quality model for software systems.

8. Conclusion and future work

The development of software systems is well recognized as an engineering activity, hence the term *software engineering*. As with all engineering activities, supervisors and practitioners must have a firm understanding of the fundamental principles of their domain. The complexity of software engineering has increased dramatically in the past decade. With the continuing increase in the variety, functionality, and complexity of software engineering, more attention must be paid to programming language suitability to make rational decisions regarding language selection.

In this study, the programming language selection process is modeled as a multi-criteria decision-making problem that deals with evaluating a set of alternatives and considering a set of decision criteria [12]. We presented a decision model for the programming language selection problem based on the technology selection framework [13]. The novelty of the approach provides knowledge about programming languages to support uninformed decision-makers while contributing a sound decision model to knowledgeable decision-makers. Furthermore, it incorporates deeply embedded requirements engineering concepts (such as the ISO software quality standards and the MoSCoW prioritization

technique) and knowledge engineering theories to develop the decision model.

We conducted seven industry case studies to evaluate the decision model's usefulness and effectiveness to address the decision problem. We find that while organizations are typically tied to particular ecosystems by extraneous factors, they can benefit significantly from using the DSS.

The case studies show that this article's decision model also provides a foundation for future work on MCDM problems. We intend to build trustworthy decision models to address the *Programming Language Framework* and *Decentralized Autonomous Organization Software as a Service Platform* selection problem as our (near) future work.

CRediT authorship contribution statement

Siamak Farshidi: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing - original draft, Writing - review & editing, Visualization, Supervision, Project administration. **Slinger Jansen:** Conceptualization, Methodology, Validation, Investigation, Data curation, Writing - review & editing, Supervision, Project administration, Funding acquisition. **Mahdi Deldar:** Investigation, Data curation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We thank the twelve experts that participated in this research. Furthermore, we thank the excellent support we have received from the journal editors and reviewers. This work is part of AMUSE Project and funded by NWO [project number 628.006.001].

Appendix. Expert interview protocols

[Evaluation of the programming languages and their features]

Step 1- A brief description of the project, the decision model, the DSS, and the main goal of the interview.

Step 2- Introductory questions:
- *How long have you worked as a developer?*
- *Could you describe what your current position encompasses?*
- *How is your position related to software development?*

Step 3- Decision-making questions:
- *How do software producing organizations typically select programming languages?*
- *What are the essential features from your perspective for selecting the best fitting programming languages?*
- *Which programming languages are typically considered as alternative solutions by software producing organizations?*

Step 4- Evaluation of the sets of [programming language features] / [programming languages]:
- *What do you think about these [programming language features] / [programming languages]?*
- *Which [programming language features] / [programming languages] should be excluded from the list?*
- *Which programming language features/languages should be added to the list?*

Step 5- Closing
- *What do you think about our work?*
- *May we contact you if we have any further questions?*
- *Can we use the name of your company in the scientific paper, or do you prefer an anonymous name?*
- *Can we use your name in the scientific paper, or do you prefer an anonymous name?*
- *Do you have any questions?*

[Mapping between the programming language features and the quality attributes]

Step 1- A brief description of the project, the decision model, the DSS, and the main goal of the interview.

Step 2- Introductory questions:
- *How long have you worked as a developer?*
- *Could you describe what your current position encompasses?*
- *How is your position related to software development?*
- *Are you familiar with the ISO/IEC quality models?*

Step 3- Mapping between the programming language features and the quality attributes: (Note: this step will be repeated for all of the programming language features and quality attributes).
- *Does the programming language feature [X] have a positive impact on the quality attribute [Y]? For instance, if a programming language supports Object-oriented programming (OOP) means that it has positive impacts on Functional completeness and Analyzability.*

Step 4- Closing
- *What do you think about our work?*
- *May we contact you if we have any further questions?*
- *Can we use the name of your company in the scientific paper, or do you prefer an anonymous name?*
- *Can we use your name in the scientific paper, or do you prefer an anonymous name?*
- *Do you have any questions?*

References

[1] G. Ruhe, Software engineering decision support–a new paradigm for learning software organizations, in: International Workshop on Learning Software Organizations, Springer, 2002, pp. 104–113.
[2] R.S. Pressman, Software Engineering: A Practitioner's Approach, Palgrave macmillan, 2005.
[3] J.E. Burge, J.M. Carroll, R. McCall, I. Mistrik, Rationale-based Software Engineering, Springer, 2008.
[4] M. Vujošević-Janičić, D. Tošić, The role of programming paradigms in the first programming courses, Teach. Math. 2 (21) (2008) 63–83.
[5] S. Jansen, S. Brinkkemper, A. Finkelstein, Business network management as a survival strategy, in: Software Ecosystems, Edward Elgar Publishing, 2013, pp. 29—42.
[6] L.A. Meyerovich, A.S. Rabkin, Empirical analysis of programming language adoption, in: ACM SIGPLAN Notices, Vol. 48, ACM, 2013, pp. 1–18.
[7] S. Peyton Jones, R. Leshchinskiy, G. Keller, M.M. Chakravarty, Harnessing the multicores: Nested data parallelism in Haskell, in: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2008.
[8] P.S. Kochhar, D. Wijedasa, D. Lo, A large scale study of multiple programming languages and code quality, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, Vol. 1, SANER, IEEE, 2016, pp. 563–573.
[9] P. Bhattacharya, I. Neamtiu, Assessing programming language impact on development and maintenance: A study on C and C++, in: Proceedings of the 33rd Int. Conference on Software Engineering, ACM, 2011, pp. 171–180.
[10] N.M. Holtz, W.J. Rasdorf, An evaluation of programming languages and language features for engineering software development, Eng. Comput. 3 (4) (1988) 183–199.
[11] D. Badampudi, K. Wnuk, C. Wohlin, U. Franke, D. Smite, A. Cicchetti, A decision-making process-line for selection of software asset origins and components, J. Syst. Softw. 135 (2018) 88–104.
[12] E. Triantaphyllou, B. Shu, S.N. Sanchez, T. Ray, Multi-criteria decision making: an operations research approach, Encycl. Electr. Electron. Eng. 15 (1998) (1998) 175–186.
[13] S. Farshidi, S. Jansen, R. De Jong, S. Brinkkemper, A decision support system for cloud service provider selection problems in software producing organizations, in: 2018 IEEE 20th Conference on Business Informatics, Vol. 1, CBI, IEEE, 2018, pp. 139–148.
[14] S. Farshidi, S. Jansen, R. De Jong, S. Brinkkemper, Multiple criteria decision support in requirements negotiation, in: The 23rd International Conference on Requirements Engineering: Foundation for Software Quality, REFSQ 2018, Vol. 2075, 2018, pp. 100–107.
[15] S. Farshidi, S. Jansen, A decision support system for pattern-driven software architecture, in: Proceedings of the 14th European Conference on Software Architecture, Vol. 1, ECSA 2020, ACM, 2020, pp. 1–12.
[16] S. Farshidi, S. Jansen, S. Fortuin, Model-driven development platform selection: four industry case studies, Softw. Syst. Model. (2021) 1–27.
[17] S. Farshidi, S. Jansen, S. España, J. Verkleij, Decision support for blockchain platform selection: Three industry case studies, IEEE Trans. Eng. Manage. (2020).

[18] S. Farshidi, S. Jansen, R. de Jong, S. Brinkkemper, A decision support system for software technology selection, J. Decis. Syst. (2018).
[19] S. Farshidi, S. Jansen, J.M. van der Werf, Capturing software architecture knowledge for pattern-driven design, J. Syst. Softw. (2020).
[20] J.R. Meredith, A. Raturi, K. Amoako-Gyampah, B. Kaplan, Alternative research paradigms in operations, J. Oper. Manage. 8 (4) (1989) 297–326.
[21] R.B. Johnson, A.J. Onwuegbuzie, Mixed methods research: A research paradigm whose time has come, Educ. Res. 33 (7) (2004) 14–26.
[22] T.R. Gruber, Automated knowledge acquisition for strategic knowledge, in: Knowledge Acquisition: Selected Research and Commentary, Springer, 1989, pp. 47–90.
[23] H.A. Simon, The Sciences of the Artificial (3rd Ed.), MIT Press, Cambridge, MA, USA, 1996.
[24] A.R. Hevner, S.T. March, J. Park, S. Ram, Design science in information systems research, MIS Q. (2004) 75–105.
[25] D. Fortus, J. Krajcik, R.C. Dershimer, R.W. Marx, R. Mamlok-Naaman, Design-based science and real-world problem-solving, Int. J. Sci. Educ. 27 (7) (2005) 855–879.
[26] J.G. Walls, G.R. Widmeyer, O.A. El Sawy, Building an information system design theory for vigilant EIS, Inf. Syst. Res. 3 (1) (1992) 36–59.
[27] M. Majumder, Multi criteria decision making, in: Impact of Urbanization on Water Shortage in Face of Climatic Aberrations, Springer, 2015, pp. 35–47.
[28] M.D. Myers, M. Newman, The qualitative interview in IS research: Examining the craft, Inf. Organ. 17 (1) (2007) 2–26.
[29] J. Corbin, A. Strauss, Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory, Sage Publications, 2014.
[30] J. Saldaña, The Coding Manual for Qualitative Researchers, Sage, 2015.
[31] S. Jansen, Applied multi-case research in a mixed-method research project: Customer configuration updating improvement, in: Information Systems Research Methods, Epistemology, and Applications, IGI Global, 2009, pp. 120–139.
[32] R.K. Yin, Case Study Research and Applications: Design and Methods, Sage Publications, 2017.
[33] R.K. Yin, The case study as a serious research strategy, Knowledge 3 (1) (1981) 97–114.
[34] DSDM Consortium, et al., The DSDM Agile Project Framework Handbook, DSDM Consortium, Ashford, Kent, UK, 2014.
[35] R. Garg, R. Kumar, S. Garg, MADM-based parametric selection and ranking of E-learning websites using fuzzy COPRAS, IEEE Trans. Educ. 62 (1) (2018) 11–18.
[36] L. Xu, S. Brinkkemper, Concepts of product software, Eur. J. Inf. Syst. 16 (5) (2007) 531–541.
[37] B. Fitzgerald, K.-J. Stol, Continuous software engineering and beyond: trends and challenges, in: Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering, 2014, pp. 1–9.
[38] I. Rus, M. Halling, S. Biffl, Supporting decision-making in software engineering with process simulation and empirical studies, Int. J. Softw. Eng. Knowl. Eng. 13 (05) (2003) 531–545.
[39] D.R. Fitzgerald, S. Mohammed, G.O. Kremer, Differences in the way we decide: The effect of decision style diversity on process conflict in design teams, Pers. Individ. Differ. 104 (2017) 339–344.
[40] L. Kaufmann, S. Kreft, M. Ehrgott, F. Reimann, Rationality in supplier selection decisions: The effect of the buyer's national task environment, J. Purch. Supply Manag. 18 (2) (2012) 76–91.
[41] R. Garg, MCDM-based parametric selection of cloud deployment models for an academic organization, IEEE Trans. Cloud Comput. (2020).
[42] R. Garg, R. Sharma, K. Sharma, MCDM based evaluation and ranking of commercial off-the-shelf using fuzzy based matrix method, Decis. Sci. Lett. 6 (2) (2017) 117–136.
[43] Sandhya, R. Garg, R. Kumar, Computational MADM evaluation and ranking of cloud service providers using distance-based approach, Int. J. Inf. Decis. Sci. 10 (3) (2018) 222–234.
[44] R. Garg, Parametric selection of software reliability growth models using multi-criteria decision-making approach, Int. J. Reliab. Saf. 13 (4) (2019) 291–309.
[45] M. Doumpos, E. Grigoroudis, Multicriteria Decision Aid and Artificial Intelligence, Whiley (UK), 2013.
[46] J.S. Dodgson, M. Spackman, A. Pearman, L.D. Phillips, Multi-criteria Analysis: A Manual, Department for Communities and Local Government, London, 2009.
[47] C.A. Floudas, P.M. Pardalos, Encyclopedia of Optimization, Springer Science & Business Media, 2008.
[48] O. Dvořák, R. Pergl, P. Kroha, Affordance-driven software assembling, in: Enterprise Engineering Working Conference, Springer, 2018, pp. 39–54.
[49] S. Farshidi, S. Jansen, M. Deldar, A decision model for programming language ecosystem selection: Seven industry case studies, Mendeley Data, Utrecht University, 2020. http://dx.doi.org/10.17632/5tc6v6zkzf.1. (Accessed 15 May 2020).
[50] ISO, IEC25010: 2011 systems and software engineering–Systems and software Quality Requirements and Evaluation (SQuaRE)–System and software quality models, Int. Organ. Stand. 34 (2011) 2910.
[51] J.P. Carvallo, X. Franch, Extending the ISO/IEC 9126-1 quality model with non-technical factors for COTS components selection, in: Proceedings of the 2006 International Workshop on Software Quality, ACM, 2006, pp. 9–14.
[52] G. Montibeller, D. Winterfeldt, Cognitive and motivational biases in decision and risk analysis, Risk Anal. 35 (7) (2015) 1230–1251.
[53] S.R. Jones, Was there a Hawthorne effect? Am. J. Sociol. 98 (3) (1992) 451–468.
[54] R. Nadeau, E. Cloutier, J.-H. Guay, New evidence about the existence of a bandwagon effect in the opinion formation process, Int. Pol. Sci. Rev. 14 (2) (1993) 203–213.
[55] R. Khadka, B.V. Batlajery, A.M. Saeidi, S. Jansen, J. Hage, How do professionals perceive legacy systems and software modernization? in: Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 36–47.
[56] P. Donzelli, Decision support system for software project management, IEEE Softw. 23 (4) (2006) 67–75.
[57] A.K. Sujeeth, H. Lee, K.J. Brown, T. Rompf, H. Chafi, M. Wu, A.R. Atreya, M. Odersky, K. Olukotun, OptiML: an implicitly parallel domain-specific language for machine learning, in: ICML, 2011.
[58] D. Dougherty, A. Robbins, Sed & Awk: UNIX Power Tools, O'Reilly Media, Inc., 1997.
[59] A.R. Mishra, A. Chandel, D. Motwani, Extended MABAC method based on divergence measures for multi-criteria assessment of programming language with interval-valued intuitionistic fuzzy sets, Granul. Comput. 5 (1) (2020) 97–117.
[60] P. Costanza, C. Herzeel, W. Verachtert, A comparison of three programming languages for a full-fledged next-generation sequencing tool, BMC Bioinformatics 20 (1) (2019) 301.
[61] A. Yıldızbaşı, B. Daneshvar, Multi-criteria decision making approach for evaluation of the performance of computer programming languages in higher education, Comput. Appl. Eng. Educ. 26 (6) (2018) 1992–2001.
[62] M. Feraud, S. Galland, First comparison of SARL to other agent-programming languages and frameworks, Procedia Comput. Sci. 109 (2017) 1080–1085.
[63] I. Yoon, J. Kim, W. Lee, The analysis and application of an educational programming language (RUR-PLE) for a pre-introductory computer science course, Cluster Comput. 19 (1) (2016) 529–546.
[64] B. Ray, D. Posnett, V. Filkov, P. Devanbu, A large scale study of programming languages and code quality in github, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2014, pp. 155–165.
[65] S.H. Lesani, B. Rouyendegh, B. Erdebilli, Object-oriented programming language selection using fuzzy AHP method, in: Annual Meeting of the ISAHP, Vol. 29, 2014, pp. 1–17.
[66] T.F. Bissyandé, F. Thung, D. Lo, L. Jiang, L. Réveillère, Popularity, interoperability, and impact of programming languages in 100,000 open source projects, in: 2013 IEEE 37th Annual Computer Software and Applications Conference, IEEE, 2013, pp. 303–312.
[67] L. Mannila, M. de Raadt, An objective comparison of languages for teaching introductory programming, in: Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006, 2006, pp. 32–37.
[68] K.R. Parker, J.T. Chao, T.A. Ottaway, J. Chang, A formal language selection process for introductory programming courses, J. Inf. Technol. Educ. Res. 5 (1) (2006) 133–151.
[69] J.K. Cochran, H.-N. Chen, Fuzzy multi-criteria selection of object-oriented simulation software for production system analysis, Comput. Oper. Res. 32 (1) (2005) 153–168.
[70] J. Gil-Aluja, Handbook of Management under Uncertainty, Vol. 55, Springer Science & Business Media, 2013.
[71] T.L. Saaty, How to make a decision: the analytic hierarchy process, European J. Oper. Res. 48 (1) (1990) 9–26.
[72] R.A. Ribeiro, A.M. Moreira, P. Van den Broek, A. Pimentel, Hybrid assessment method for software engineering decisions, Decis. Support Syst. 51 (1) (2011) 208–219.
[73] M. Khari, N. Kumar, Comparison of six prioritization techniques for software requirements, J. Glob. Res. Comput. Sci. 4 (1) (2013) 38–43.
[74] I. Ibriwesh, S.-B. Ho, I. Chai, Overcoming scalability issues in analytic hierarchy process with ReDCCahp: An empirical investigation, Arab. J. Sci. Eng. 43 (12) (2018) 7995–8011.

Online references

[75] Small place to discover languages in GitHub, 2014, https://githut.info/.
[76] Visualizing language migration over time, 2017, https://www.i-programmer.info/news/98-languages/10943-visualizing-language-migration-over-time.html.
[77] Dear developers: Coding languages that will set you apart, 2019, https://hired.com/blog/candidates/data-reveals-hottest-coding-languages/.
[78] Developer survey results, 2019, https://insights.stackoverflow.com/survey/2019.
[79] Do you speak code? 2019, https://www.codingame.com/work/resources/codingame-2019-developer-survey/programming-languages/.
[80] Eight top programming languages and frameworks of, 2019, https://hackernoon.com/8-top-programming-languages-frameworks-of-2019-2f08d2d21a1.
[81] Look at 5 of the most popular programming languages, 2019, https://stackify.com/popular-programming-languages-2018/.
[82] Most used programming languages among developers worldwide, 2019, https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/.

[83] Programming languages InfoQ trends report, 2019, https://www.infoq.com/articles/programming-language-trends-2019/.
[84] PYPL PopularitY of Programming Language, 2019, http://pypl.github.io/PYPL.html.
[85] The RedMonk programming language rankings, 2019, https://redmonk.com/sogrady/2019/07/18/language-rankings-6-19/.
[86] The state of developer ecosystem, 2019, https://www.jetbrains.com/lp/devecosystem-2019/.
[87] The top programming languages, 2019, https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019.
[88] Top programming languages rankings, 2019, https://dzone.com/articles/top-programming-languages-rankings.
[89] What stats & surveys are saying about top programming languages, 2019, https://codinginfinite.com/top-programming-languages-2020-stats-surveys/.
[90] TIOBE index, 2020, https://www.tiobe.com/tiobe-index/.
[91] The DSS usage data, 2021, https://github.com/SiamakFarshidi/dss-usage-data-log-analysis.git.