FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Adaptable Costs Evaluator
Student:	Bc. Tomáš Patro
Supervisor:	Ing. Marek Suchánek
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of winter semester 2022/2023

Instructions

Storage Costs Evaluator [1,2] is a web application used for estimating the costs of data storages [3]. There are many other areas where similar estimation applications would be highly beneficial; however, the calculations are hard-coded in the Evaluator. The goal of this thesis is to develop a web application with configurable inputs and calculations.

- Analyze the Storage Costs Evaluator and set the requirements for the new configurable application.
- Research existing solutions, methods, and technologies relevant to the topic (configurable computations, API design, spreadsheets imports and export, etc.).
- Design the application with a focus on its versatility and maintainability.
- Implement the application with the use of functional programming languages.
- Discuss the benefits of the application and outline future development.

[1]: https://github.com/ds-wizard/storage-costs-evaluator
[2]: https://storage-costs-evaluator.ds-wizard.org
[3]: https://doi.org/10.5281/zenodo.4033086

Electronically approved by Ing. Michal Valenta, Ph.D. on 11 March 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Adaptable Costs Evaluator

Bc. Tomáš Patro

Department of Software Engineering

Supervisor: Ing. Marek Suchánek

January 4, 2022

Acknowledgements

I would like to express my special thanks of gratitude to my supervisor Ing. Marek Suchánek, who has always been helpful during the work on this thesis. Also, I would like to thank my family and closest friends for their love and support that helped me finish this project within the limited time.

Declaration

In Prague on January 4, 2022 . . . .

Citation of this thesis

Patro, Tomáš. *Adaptable Costs Evaluator*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Funkcionálne programovanie je paradigma, ktoré za posledných pár rokov narástlo na svojej popularite. Veľa aspektov tohto programovacieho štýlu z neho robí vhodnú voľbu pre softvérové projekty a webový vývoj. Diplomová práca aplikuje funkcionálne programovanie pri vývoji novej webovej aplikácie, ktorá umožňuje užívateľom adaptabilným spôsobom tvoriť vlastné výpočty, ukladať výsledky a zdieľať ich s kolegami v organizáciách. Práca analyzuje existujúce riešenia a metódy danej problematiky, a získava, a vyhodnocuje spätnú väzbu od užívateľov aplikácie *Storage Costs Evaluator*. Taktiež poukazuje na problémy, ktoré má implementácia tejto aplikácie a navrhuje nový dizajn, ktorý by bol viac flexibilný, udržateľný a ktorý by riešil dané problémy. Implementácia novej aplikácie je do detailu opísaná vrátane použitých technológií, metód a využitia funkcionálnych jazykov. Práca nakoniec diskutuje výhody nového dizajnu a implementácie a podáva návrh na budúci vývoj aplikácie.

Kľúčové slová funkcionálne, programovanie, výpočty, adaptabilné, softvér, dizajn, web, vývoj

Abstract

Functional programming is a paradigm that has gained popularity in the past few years. Many aspects of this programming style make it suitable for software projects and web development. This thesis applies functional programming to implement a new web application that enables the users to create custom computations in an adaptable way, store their results, and share them with colleagues in the organizations. It analyzes current solutions and methods in the given field and gathers and evaluates feedback from the users of the *Storage Costs Evaluator* application. The thesis points out the problems with the implementation of the given application and proposes a new design that would be more versatile, maintainable, and address the discussed problems. The implementation of a new application is described in detail, including used technologies, methods, and usage of functional programming languages. Lastly, the benefits of the new design and implementation are discussed, and the proposal for future development is presented.

Keywords functional, programming, computations, adaptable, software, design, web, development

Contents

List of Figures

List of Tables

List of Code Snippets

Introduction

Functional programming is a paradigm that is often compared to imperative and object-oriented programming. It comes with a specific programming style where the usage of the pure function is the primary tool to structure the code. This approach tries to solve some of the problems with the constructs commonly occurring in the imperative programming like a variable mutability, side effects caused by the global state, etc. [1]

This thesis uses functional programming to implement a configurable web application for adaptable computations. This new implementation enhances *Storage Costs Evaluator (SCE)* application [2] that is a current solution for the computation of the storage costs. The problem with the current solution is that the computations are hard-coded in the implementation. This fact makes changes in the computation logic bounded to the re-implementation of the application that can be a slow and inflexible process.

We propose an application design that addresses the mentioned problems. It points to the inflexible parts of the existing application and proposes solutions that redesign these parts in a more flexible and versatile manner. The new *Adaptable Costs Evaluator (ACE)* web application [3] implements the created design allowing clients to manipulate with the adaptable computations through the well-defined Representational State Transfer (REST) [4] Application Programming Interface (API). We achieve this by using functional programming technologies and following the best web development practices.

The first part of the thesis analyzes the current Storage Costs Evaluator (SCE) application. It looks at its features, architecture but also gathers and evaluates feedback from the existing users of the application. In the analysis, we also look at the existing solutions, methods, and technologies in the field. Based on the conducted analysis, we set functional requirements on the new application and divide them into several groups based on their nature.

In the second part, we propose a new application design. We describe a high-level architecture design that discusses the application's front-end, back-end, and database parts and looks at the design in the context of functional programming.

The last part describes the conducted implementation. We look at the used technologies for the development and dig into the implementation details. We also discuss how we approached additional aspects of the implementation like code testing, documentation, and Continuous Integration (CI) [5]. Lastly, we discuss the implemented functional requirements, outline the future development, and analyze the benefits of the new implementation.

Goals

One of the goals of Chapter 1 is to take a look at the existing SCE application and point out the problems this implementation has. Next, the goal is to gather and analyze the feedback from the users of the given application. Then, it is to take a look and analyze the existing solutions, methods, and technologies. Lastly, based on the previous analysis, the goal is to set Functional Requirements (FRs) for the new Adaptable Costs Evaluator (ACE) application.

The goal of Chapter 2 is to propose a new application design that would address the problems and FRs set in the previous chapter. Lastly, the main goal of Chapter 3 and thesis is to develop a new ACE application based on the conducted analysis and new design. The goal is also to discuss the implementation details, outline future development, and look at the benefits of the new implementation.

Chapter 1

Analysis

This chapter contains the analysis that we later use in the design process of the ACE application. The decisions made while designing the application are mostly derived from the results of the analysis we present in the following chapter. It is broken down into the following parts:

1. **SCE Features Analysis** – features and FRs analysis of the SCE application.
2. **SCE Architecture Analysis** – analysis of the technologies and architecture of the SCE application.
3. **SCE Users Survey** – usage information and feedback from the existing users of the SCE application.
4. **Users Survey Results** – feedback from the users currently using the SCE application.
5. **Analysis of Existing Solutions, Methods, and Technologies** – analysis and research of the existing solutions, methods, and technologies regarding the topic.
6. **ACE Functional Requirements** – FRs on the new ACE application based on the previous analysis.

The order of the different parts of the analysis, as listed above, corresponds to the order in which the analysis was conducted.

1.1 Storage Costs Evaluator (SCE) Features Analysis

This section contains an analysis of the basic features of the SCE application. It is an analysis from the end-user perspective.

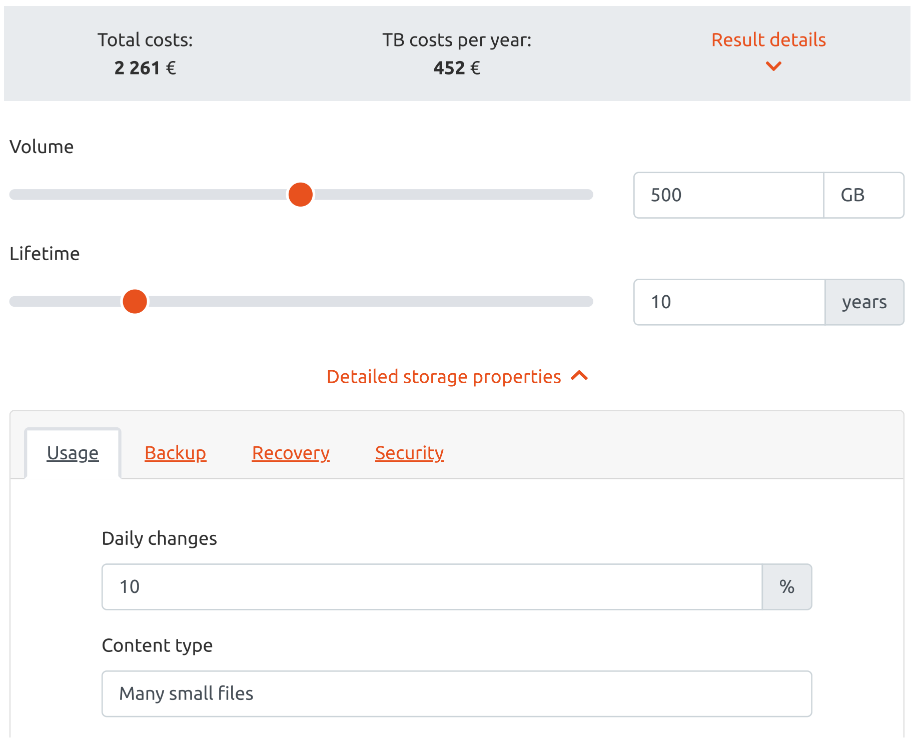
The output of this analysis is a set of FRs which the SCE application currently implements. Since the new proposed application should be an extended and generalized version of the SCE, these FRs are to be taken into account in the design process of the new application. The existing requirements do not need to be incorporated 1:1 in the new application, but a fair portion of them should be included in some way in the design process.

1.1.1 Features Overview

SCE provides functionality to calculate costs of (digital) storage. The calculations are based on the spreadsheet created by Rob Hooft [6]. Let's take a look at the implemented FRs from the perspective of a whole application (fron-tend and back-end).

The application User Interface (UI) is in the form of a Single Page Application (SPA). The application can be divided into two screens. Both screens share the header with the total costs and yearly costs for the terabyte (TB) of storage.

The first screen from Figure 1.1 consists of multiple input fields of different kinds (slider, text input, select box, etc.). These input fields correspond to the input cells defined in Hooft's spreadsheet.


Total costs: **2 261 €**
TB costs per year: **452 €**
Result details

Volume: 500 GB
Lifetime: 10 years

Detailed storage properties

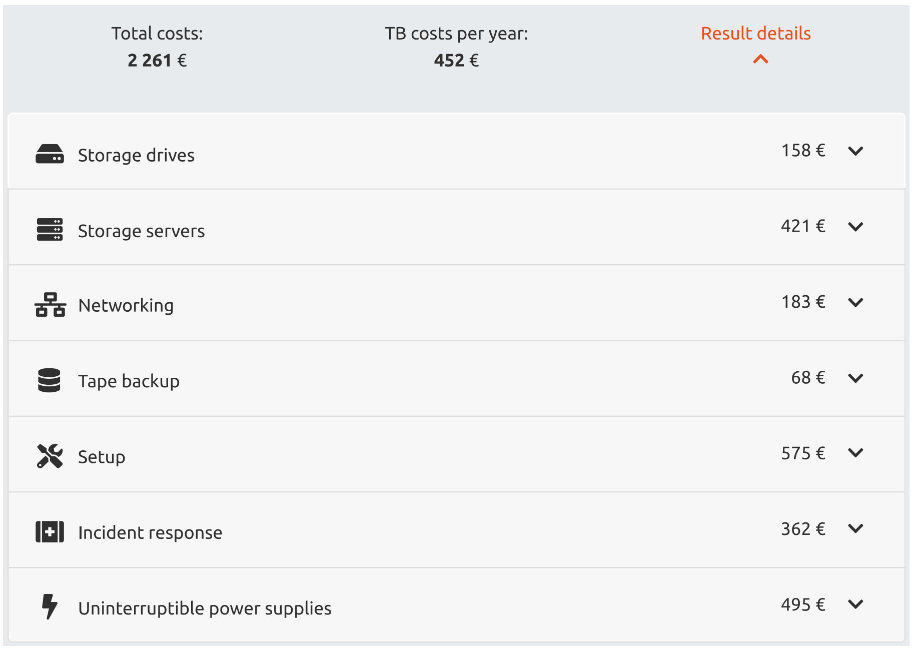
Usage | Backup | Recovery | Security

Daily changes: 10 %
Content type: Many small files

Figure 1.1: SCE first screen [2]

The input fields are logically divided by various categories – backup, recovery, servers, etc. There are also three levels of configuration – basic, detailed, and expert. By default, only the first level is visible to the user on the page load. The application recalculates the costs on every change of any of the input fields.

The second screen from Figure 1.2 provides result details. The overall costs are broken down to several categories like storage drives, networking, incident response, etc. Each category can also be expanded and a user can reveal a detailed breakdown of the needed hardware/software components and services. Partial prices for the particular category are also available in the expanded menu.



Total costs: **2 261 €**

TB costs per year: **452 €**

Result details

Storage drives	158 €
Storage servers	421 €
Networking	183 €
Tape backup	68 €
Setup	575 €
Incident response	362 €
Uninterruptible power supplies	495 €

Figure 1.2: SCE second screen [2]

1.1.2 Functional Requirements

Based on the overview from the previous section, we can summarize the existing features to the following FRs, in Table 1.1, which the application implements. Note that we also add some implementation details to the table. Even though we haven't covered the architecture and code, FRs often include some implementation aspects.

The FRs summarized in the Table 1.1 give us a high-level view of what functionalities the current application implements. We will later use this table as a foundation for the FRs of the new application.

FR No.	FR Description
FR 1	The user can fill in all input fields as they are defined in Hooft's spreadsheet. [6]
FR 2	Total costs and TB costs per year are always visible to the user.
FR 3	All of the fields have default values on the page load.
FR 4	The input fields are logically divided into several categories based on their characteristics (backup, security, etc.).
FR 5	The input fields are divided into three major categories based on the level of configuration – basic, detailed, and expert.
FR 6	The application recalculates the costs on every change of an input field.
FR 7	The application offers detailed results where the price is decomposed to the various categories and items – storage drives, networking, setup, etc.

Table 1.1: SCE FRs

1.2 SCE Architecture Analysis

The following section contains the analysis of the code, technologies, and architecture of the SCE application. [2][7]

In the first section, we will look at the technologies analysis of the application. We will mainly focus on the technical details of the application. In the second section, we will look at the high-level architectural design of the code.

1.2.1 Technologies

We break down the used technologies into two major categories – front-end and back-end. We take a closer look at the used technologies and how they relate to the used programming languages.

1.2.1.1 Front-end Technologies

Front-end is a straightforward SPA that uses Embedded JavaScript (EJS) templating engine [8] to render the content fetched from the webserver. The EJS templates are combined with pure JavaScript to deliver dynamic content and proper rendering of the fetched data from the webserver.

The EJS templates are enriched with Bootstrap framework [9], which is used to deliver a better User Experience (UX), incorporate Saas CSS pre-processor, and offer a more appealing design. The application also uses some other popular libraries like jQuery [10] or Font Awesome [11]. The application uses Webpack [12] to build the front-end and bundle different modules and assets together. To summarize the major used front-end technologies:

- **Embedded JavaScript (EJS)** – HTML templating engine.
- **Bootstrap** – front-end toolkit to enhance design and UX.
- **Webpack** – modules and assets bundler.

1.2.1.2 Back-end Technologies

The back-end of the application is written purely using Haskell programming language. It is a simple web application with two HTTP endpoints. SCE back-end uses Scotty web framework [13] to implement the HTTP endpoints in a declarative way. The back-end application uses Stack tool [14]. It is a cross-platform program for developing and building Haskell projects. SCE uses it to compile and run the code. To summarize the major used back-end technologies:

- **Haskell** – programming language for back-end.
- **Scotty** – web framework for the Haskell language.
- **Stack** – development and build tool for Haskell programs.

1.2.2 Architecture Design

In the following section, we will look at the high-level design of the SCE's architecture. Let's start by looking at Figure 1.3 capturing the architecture design.

The diagram shows us how the different parts of the application communicate with each other and with the client. In the context of SCE, the client is a web browser.

The first thing we want to discuss is the communication between the front-end and back-end. From the Figure 1.3, it would seem that the back-end and front-end are two separate applications. In fact, the application is written as a monolith. The diagram only captures different parts of the monolith. We can relate the design of the application to the Model-View-Controller (MVC) web frameworks like Ruby on Rails [15]. The MVC web frameworks usually separate HTML templates (*View*) to the separate layer within the application. It is also similar in the context of SCE.

The core of the application is back-end written using the Scotty web framework. We could say that this part contains both *Model* and *Controller* parts if we compared it to the traditional MVC applications. This comparison is not so obvious in the context of the Functional Programming (FP) and web frameworks written using FP languages.

1.2.2.1 Back-end Architecture

Let's look at the back-end design of the application. As mentioned before, this part consists of the *Model* and *Controller* parts if we related it to the traditional MVC applications.

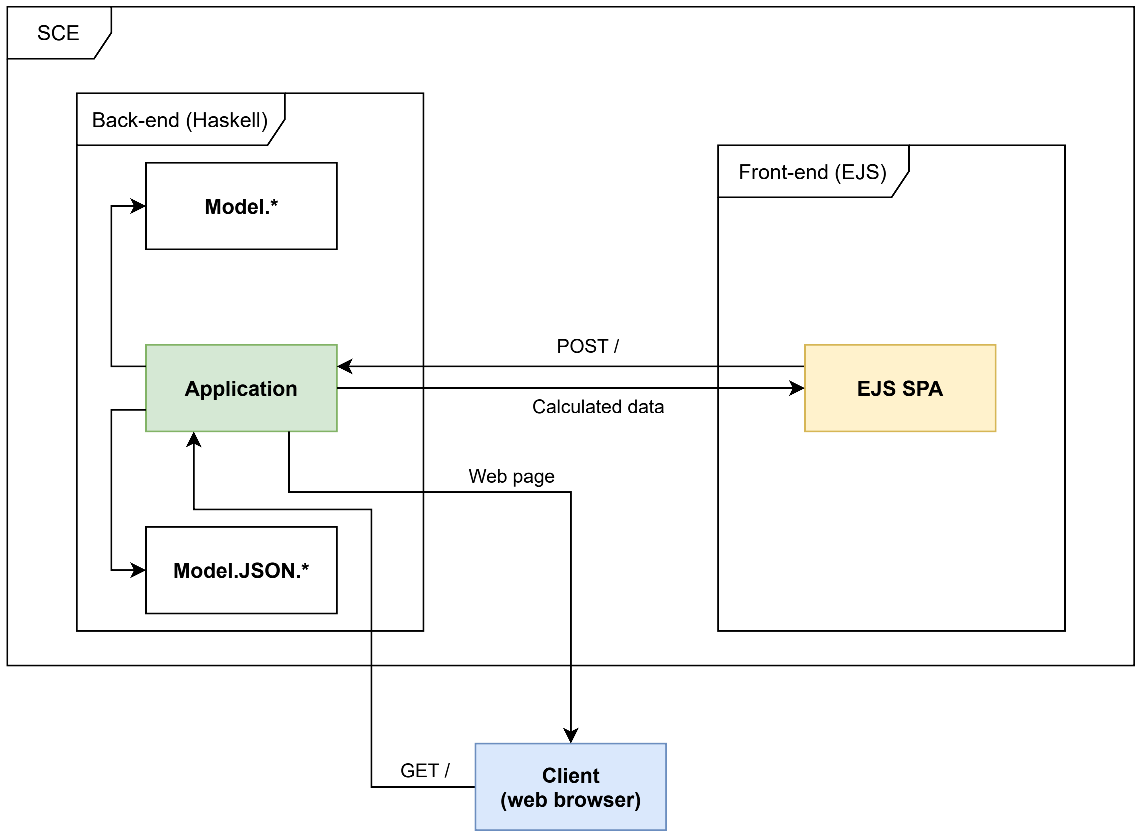


Figure 1.3: SCE architecture design

However, we have to take a different approach and change our point of view when we are talking about the FP design.

We will not talk about the concepts of the FP in this part. Let's emphasize that every programming construct of FP is a (pure) function. We can imagine complex programs as a set of higher-order functions and their compositions. The back-end part of the application is written using the Haskell programming language. One of the core constructs of the language is a module. The module is a set of functions that can be exported and reused in the other part of the program which imports this module. The module is also a construct that helps us to achieve the Separation of Concerns (SoC).

The architecture of the SCE back-end is designed around these modules. We can see the main modules and relations between them in the diagram above. The design is quite simple. The `Application` is the main module that defines the application routing – exposed HTTP API endpoints and their handlers.

Now, we can take a closer look at the other two modules from the diagram. The `Models.*` modules consist of the business logic of the application. They define things like input processing and particular computations. The computations in these modules are an imprint of the equations which Hooft defined in his spreadsheet [6].

The scope and possible extension of the computations are limited by their definition in the code. In order to add a new equation or change an existing one, we need to change the program's code. This is a limitation that has many implications. One of the implications is the need to re-compile and re-deploy the application to the server every time we want to make a change to the computations.

The second set of `Model.JSON.*` modules are serialization and deserialization definitions of the JSON objects. To be precise, they contain functions to map incoming JSON payload to the application data types and mapping of the data types to the response JSON payload.

To summarize, if we tried to compare the design to the MVC design pattern, we could say that the `Application` module plays the role of the *Controller* mixed with the routing middleware. The `Model.*` and `Model.JSON.*` modules play the role of *Models* mixed with some library business logic.

1.2.2.2 Front-end Architecture

As mentioned before, the front-end design of the application is based on the EJS templates. We will not cover the details of different templates design since it is not necessary for the sake of this analysis. Let's just say that different templates represent different parts of the SPA. The templates work as construction components of the whole webpage. The JavaScript files orchestrate the construction itself. The whole design is straightforward, and there is no complexity above average. The EJS templates can be seen as the *View* part of the traditional MVC design pattern.

1.2.2.3 Communication Between Layers

The layers of the applications communicate with each other using the HTTP protocol. There is only one published HTTP endpoint at the back-end side at the root path `/`. We can send HTTP requests using either the `GET` or `POST` HTTP method. The `POST` method is used by the front-end layer to send the current data from the forms and to receive the recalculated results. The `GET` method is intended to be used by the web browser client to fetch the webpage itself. As we can see from the Figure 1.3, there is no further complexity in the communication between the layers.

1.2.3 Summary

We can summarize that the design of the application is quite simple. The current design satisfies the needs for the particular mathematical model for computations. However, the design does not offer any means of configuration and adjusting the current mathematical model. It also does not offer the user to either save the results in the application or export them. The application's business logic only offers the means of receiving the input data and outputting the results after the computation.

The other weak part of the current solution is the public API. The application offers only one HTTP endpoint. It is a satisfactory design for the current solution, but it does not offer much flexibility for the possible extensions of the application and integrations with the foreign systems.

The front-end of the application is a simple SPA that reflects the back-end solution and its capabilities. However, it is not a design that would be suitable for further extensions and new features since it does not offer the means of modern front-end web frameworks.

1.3 SCE Users Feedback

The following section contains the description of the methodology used to gather feedback from the current users of the SCE application. As mentioned before, the SCE application is a simple SPA. Therefore, we decided to gather the feedback using an online survey.

The survey was conducted using the Google Forms platform [16]. The advantage of this platform is that it saves the results directly to the spreadsheet, which can be exported in CSV format and programmatically processed.

1.3.1 Survey Structure

We designed the questions and format of the survey by following Sheinderman's recommendations from the book *Designing User Interface* [17]. The survey is split into two parts. The survey begins by collecting some characteristics about the users. These characteristics include the following items:

1. background demographics (age, gender, education, job),
2. experience with computers (scoped to the web applications),
3. and familiarity with the computations (storages, costs, mathematical models, etc.).

The users can answer the questions, or they can mark that they prefer not to say in each question. We use the demographics data to understand better the users that use the application. The demographics data can also be correlated with the results from the other parts of the survey.

The first part consists of the series of statements where users are asked to respond using the *Likert* scale described in Table 1.2.

The results are used to demonstrate the current applications' strong and weak aspects. We also use them to indicate whether the users find the characteristics of the proposed

Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree

Table 1.2: *Likert* scale for the questions in the first part of the survey

extended application feasible and agree with them (*Likert* scale). The important characteristic of the first part is the option to process the results statistically.

The second part is used as a set of recommendations from the users. These recommendations are later incorporated into the functional requirements, and we can take advantage of them while designing the new application.

We gather the recommendations by letting the users freely answer the prepared questions and express their opinions about a particular aspect. The users are also able to leave an extra comment about anything related to the topic.

1.3.2 Survey Questions

First, we have to emphasize that some questions are accompanied by a brief description, so the user better understands them.

In the following section, we summarize the questions of the survey into the two lists. The first list consists of the questions from the **first part** (*Likert* scale):

1. It is intuitive to work with the UI of the current application.
2. The application offers functionalities that satisfy my expectations.
3. I can effectively perform the tasks using the UI.
4. The input fields and results are placed where I expected to find them in the UI.
5. I would welcome the ability to adjust how the calculations are performed.
6. It would be convenient to create/import my own set of rules and conditions for the calculations.
7. I would find it useful to be able to create an account and save the preferences, calculation results, etc.
8. My colleagues and I would like to use the application together, save, and share the calculations and results within the application.
9. It would be useful for me to have the ability to export the calculation results in the given format (CSV, XML, etc.).
10. Me or my colleagues would use the web API to interact with the application and integrate it into other systems.

The second list is a set of questions from the **second part** (textual):

1. Write down 1–3 (or more) most important features you would welcome in the new application.
2. State 1–3 (or more) aspects of the current application that prevent you from achieving your goals or do not let you work efficiently.
3. What are the most important features of the current application that you or your colleagues find useful?
4. Write down any other ideas, recommendations, observations, or notes regarding the new application.

1.4 Users Survey Results

In this section, we analyze the results from the users survey. We described the methodology used to design the survey in the Section 1.3. We asked approximately 40–50 users of the current application to fill in the survey. However, we gathered responses from overall 6 current users of the SCE application.

1.4.1 Demographics

For the purpose of correlating the results, we first take a look at the demographics of the respondents.

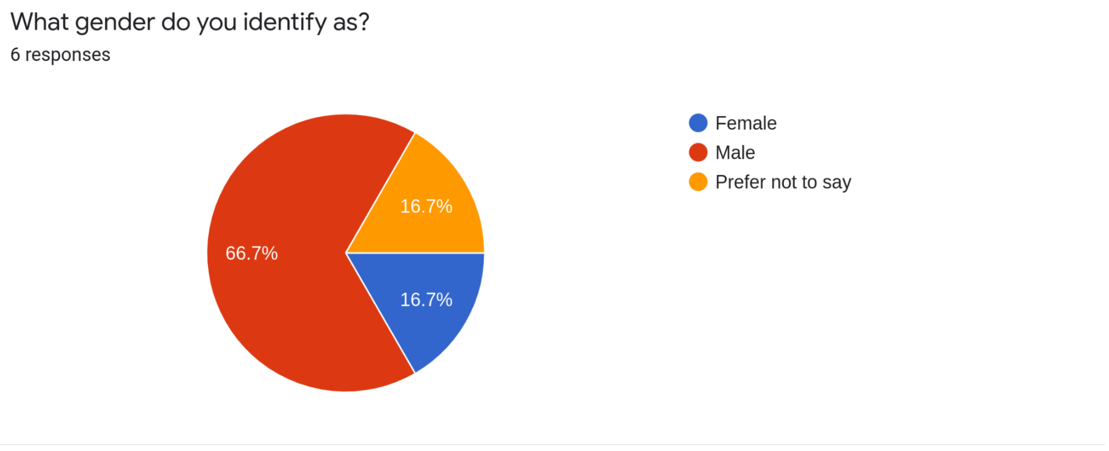


Figure 1.4: Demographics question no. 1

The majority of the users identify themselves as male. Based on the *Engineering UK 2018* report [18], only 12.37% of all engineers are women in the UK. We can relate these statistics to the results we got.

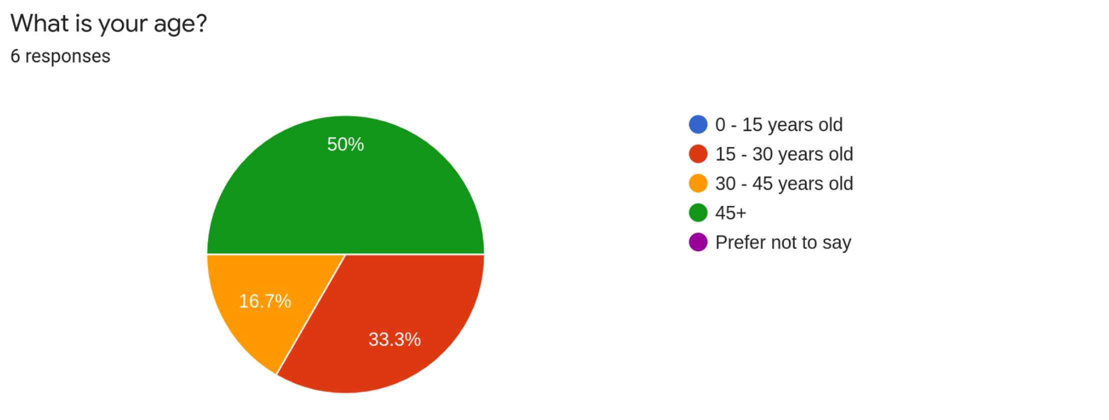


Figure 1.5: Demographics question no. 2

The majority of the respondents are 45+ years old. This can be related to the fact that SCE is a domain-based application that requires some level of expertise to interpret the results it computes.

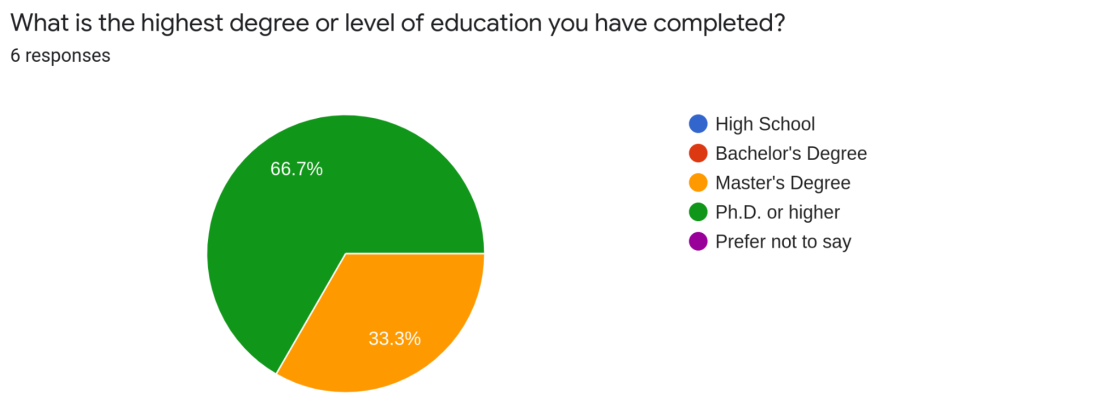


Figure 1.6: Demographics question no. 3

The majority of the respondents completed a Ph.D. or higher education. Also, all respondents completed at least a master's degree. The SCE application is probably mainly used within the academic environment or the people with completed higher education. This fact also relates to the age ratio of the respondents who are 45+ years old, where there is commonly a higher chance that the academics would have a higher level of education as they are getting older.

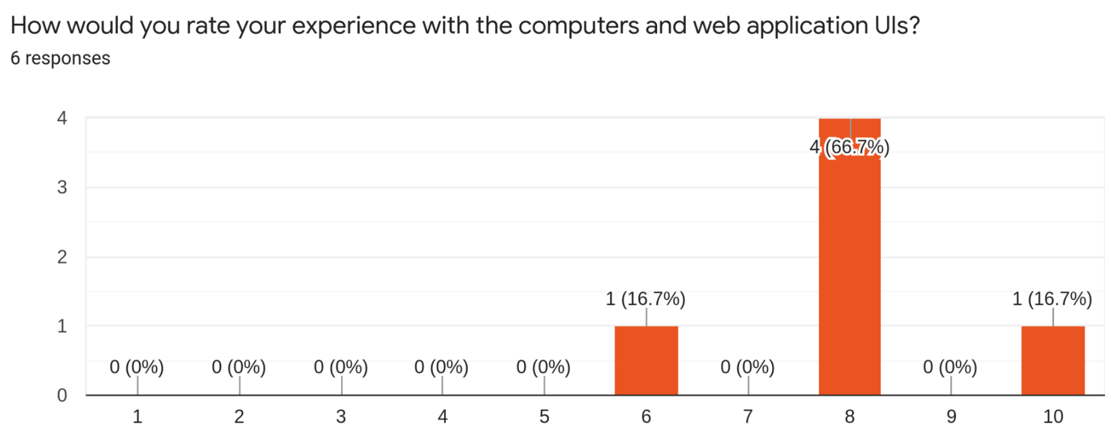


Figure 1.7: Demographics question no. 4

Most of the respondents are experienced users of computers and web applications. We can relate this to the fact that the domain is related to the field of computer science.

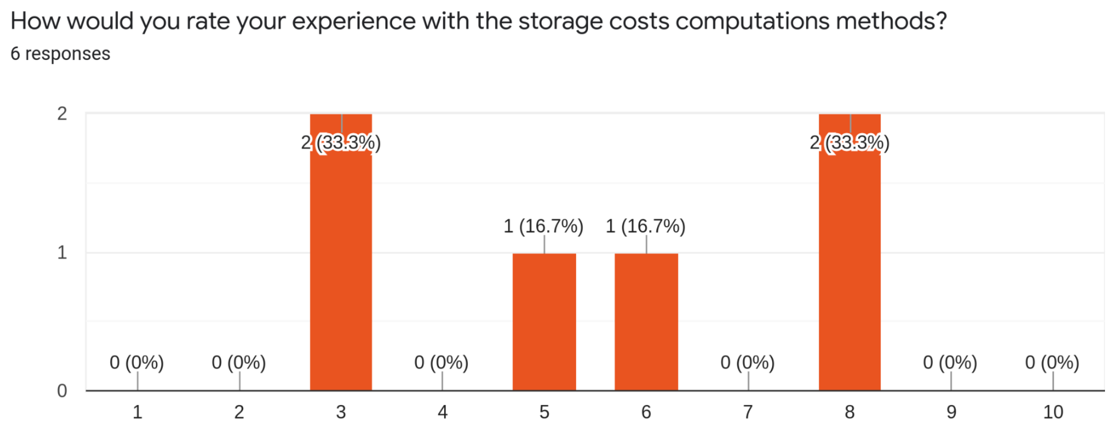


Figure 1.8: Demographics question no. 5

The results of this question are various. We can observe that a portion of respondents does not have extensive knowledge of the computation methods. It would be reasonable to design a new application to be understandable also for the users without previous expertise in the computation methods domain.

1.4.1.1 Summary

We can observe that most of the respondents are male who are 45+ years old with the completed higher education. The most common level of education is Ph.D. or higher. Based on this fact, we can conclude that the SCE application is mainly used by academics or specialists in the given domain. The ratio of male respondents can relate to the fact that the majority of the people working in engineering are male.

The respondents are mostly skilled computer and web applications users. This probably relates to the domain, which is associated with the computer science field. On the other hand, the respondents are not so skillful when it comes to the domain of computation methods. We should consider this fact when designing the application.

1.4.2 Likert Scale Questions

In this part, we take a look at the results from the Likert scale questions. These are the questions where the respondents marked how likely they agree or disagree with the given question (statement) on the given scale.

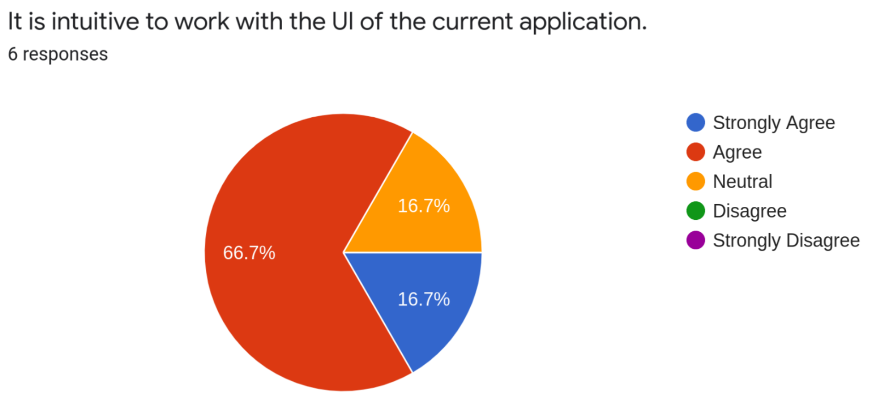


Figure 1.9: Likert scale question no. 1

The users mostly agree that the UI of the current application is intuitive to work with. We should take this fact into count when designing the new application. The UI characteristics of the current application can serve as an inspiration when designing a new application.

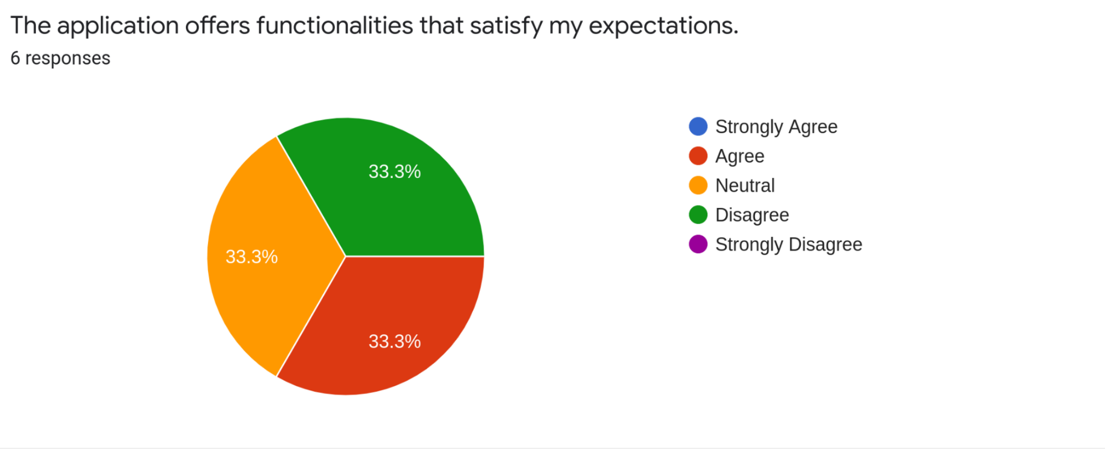


Figure 1.10: Likert scale question no. 2

We can see that the respondents are divided when it comes to this question. Many of them are not satisfied with the implemented functionalities and would welcome new features in the app.

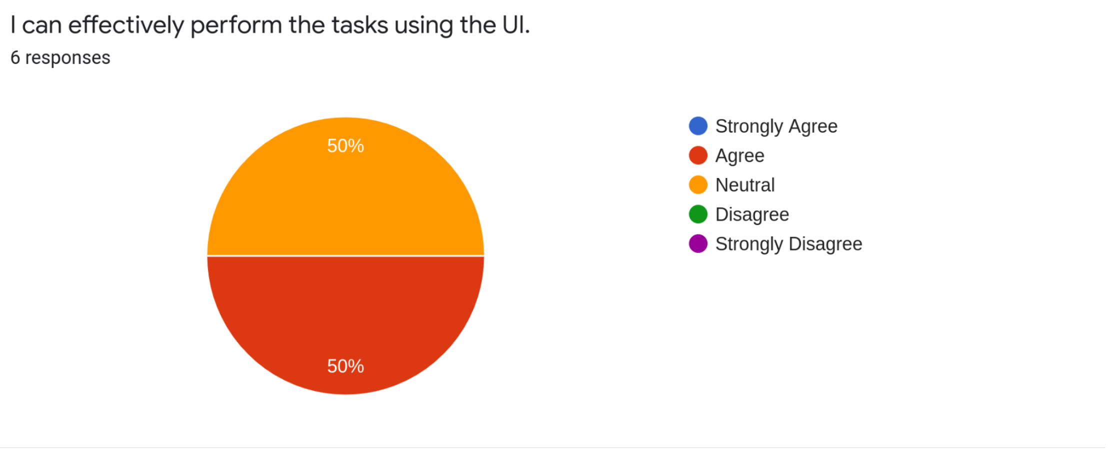


Figure 1.11: Likert scale question no. 3

Half of the respondents have a neutral opinion when it comes to the effectiveness while performing the tasks in the UI. The other half agree that they can be effective while performing these tasks. However, we should think about better usability when designing the new application.

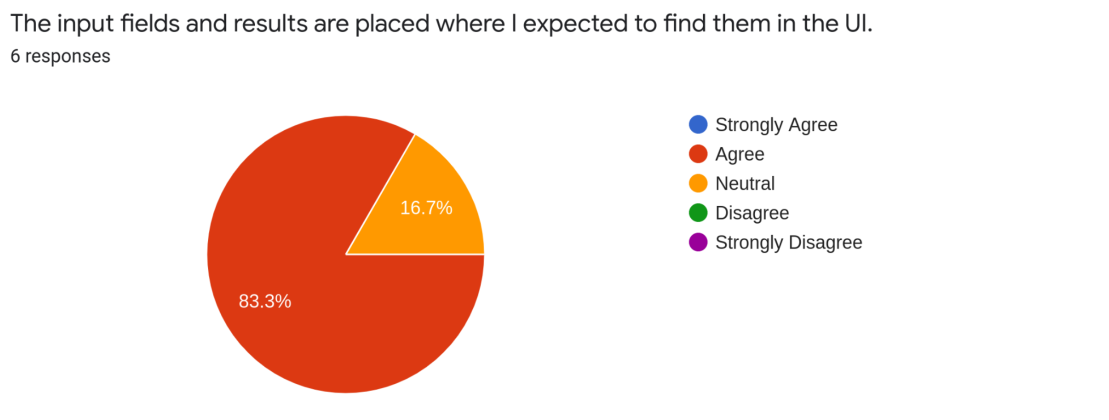


Figure 1.12: Likert scale question no. 4

This is an important question since the location of the input fields and corresponding results of the computations can play an essential role in the overall feeling from using the UI. Most of the respondents agree that these fields are placed where they would expect them to be. We can design the new application and place the input fields in a similar fashion as the current app.

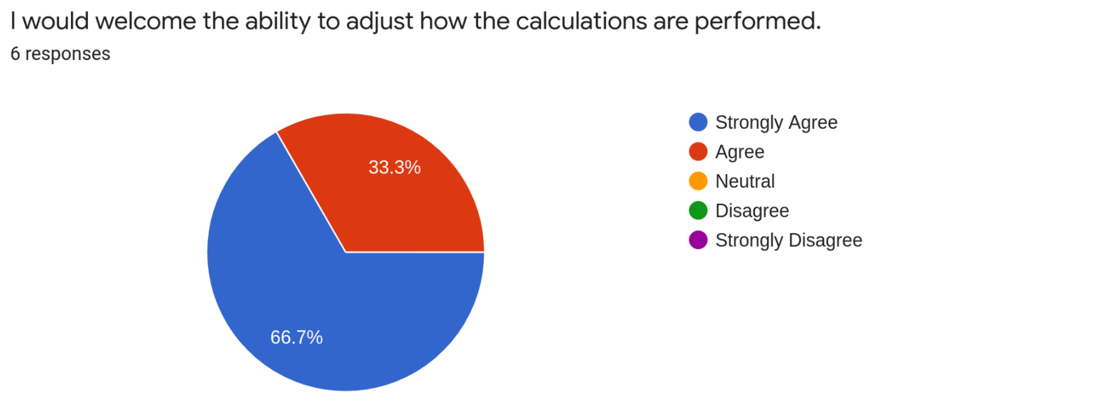


Figure 1.13: Likert scale question no. 5

The respondents agree that they would welcome the functionality where they could adjust the way that the calculations are performed. This is an important fact to consider when setting the FRs for the new application.

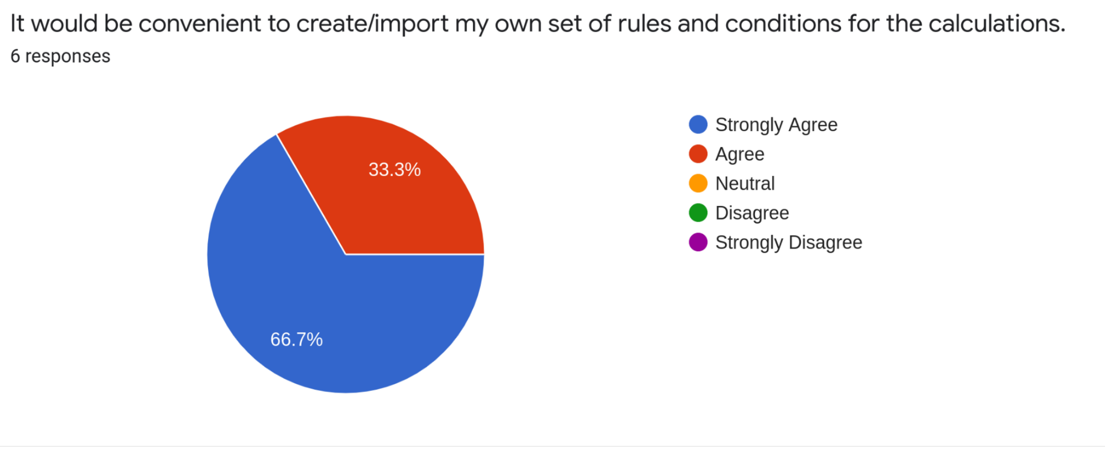


Figure 1.14: Likert scale question no. 6

Another important feature to include in the FRs. The ability to create or import your own set of rules and conditions for the calculations is an important function of the adaptability of the new app.

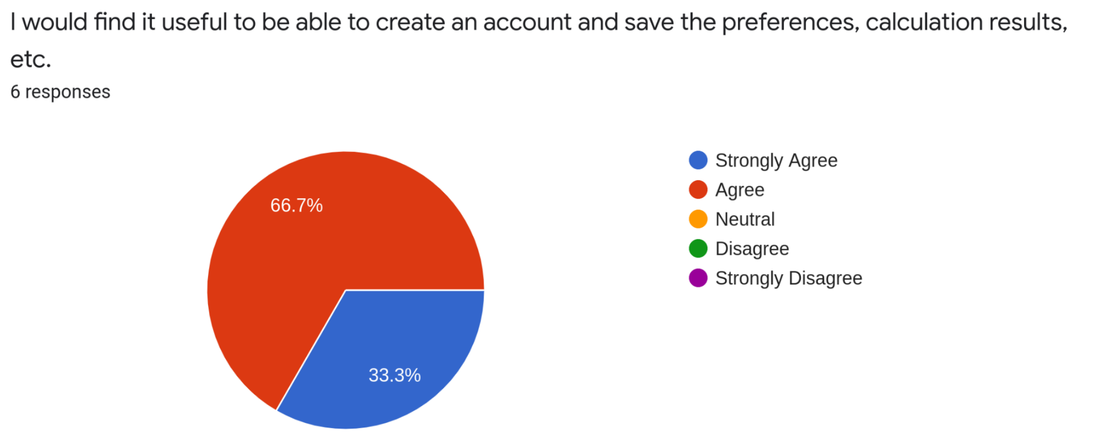


Figure 1.15: Likert scale question no. 7

This functionality would require implementing a system of users with proper authentication and eventually authorization. However, most of the respondents agree that they would welcome this functionality. It would enable the users to save their preferences, results of the calculations, etc.

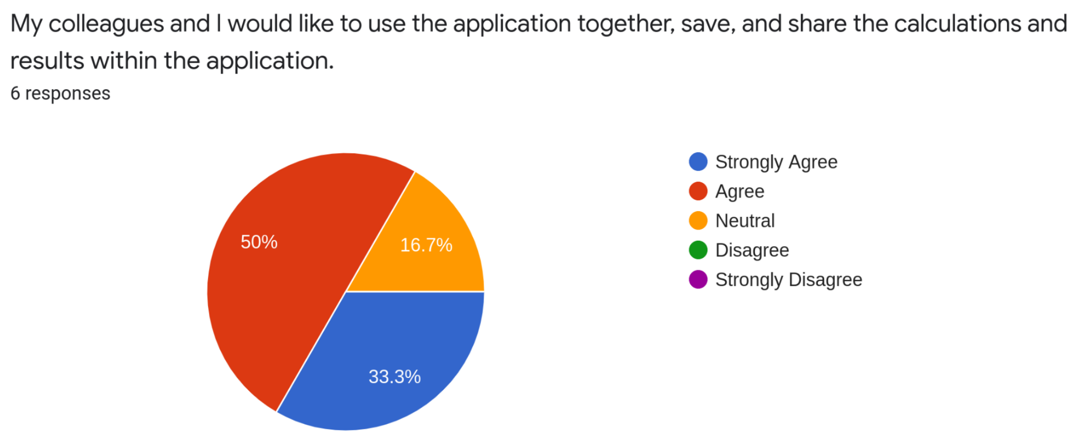


Figure 1.16: Likert scale question no. 8

Most of the respondents would also welcome this feature enabling a form of collaboration. The implementation would be based on the system of users and organizations or similar entities grouping these users together.

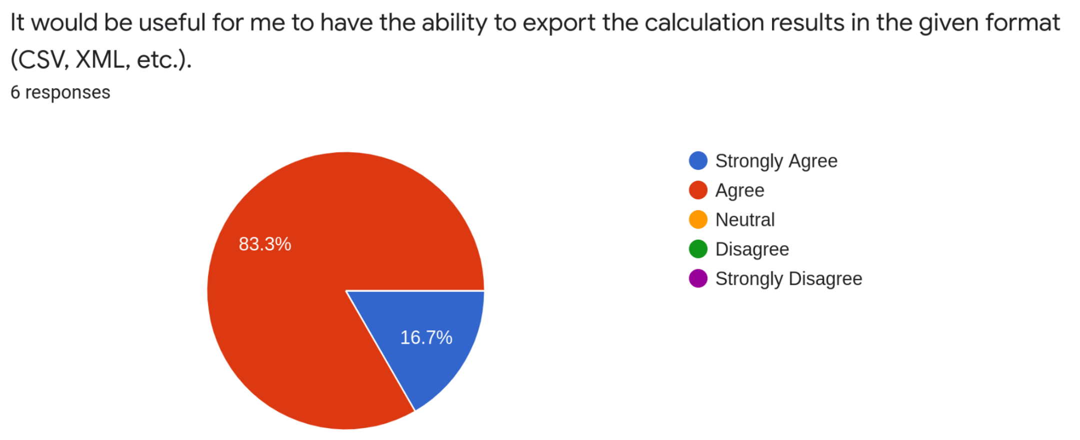


Figure 1.17: Likert scale question no. 9

This functionality would be helpful for the people who would like to take the results of the computations and import them into a different system or application in the given format. Most of the respondents marked that they would find this functionality useful.

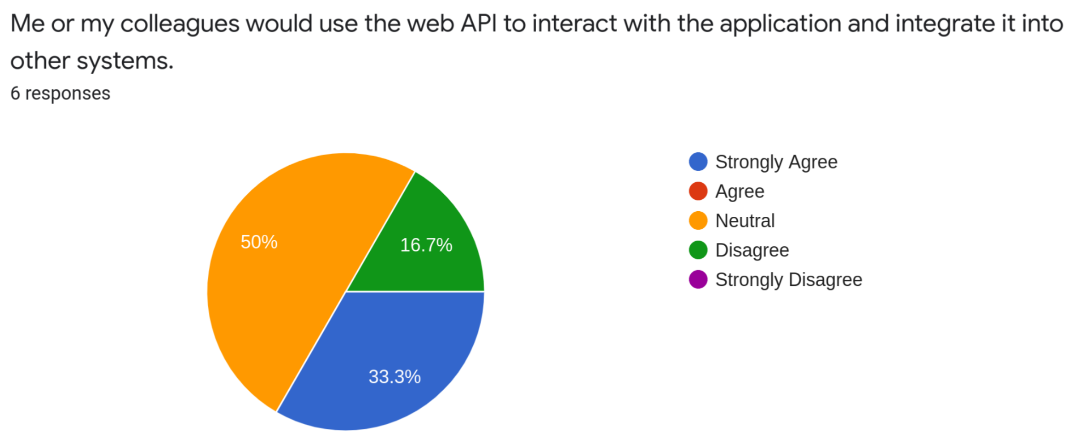


Figure 1.18: Likert scale question no. 10

The decision to implement a public web API for the application would enable the third parties to integrate the application into the different systems and UIs. Half of the respondents marked that they would use this type of API.

1.4.2.1 Summary

Most of the respondents agree that they can effectively work with the current application. They also agree that the UI is well-designed, and the experience of working with it is positive.

The respondents agree that they would find it useful if the application had extended functionality. They positively responded to the proposed features and marked that they would find them useful in the new application. Half of the users agree that they would take advantage of the public web API to interact with the application and integrate it into other systems.

1.4.3 Textual Questions

In the following section, we take a look at the textual answers to the questions. These questions are aimed to gather the opinions and ideas from the respondents. These questions serve as valuable feedback to identify the strengths of the current application and gather the ideas for the features to be included in the newly proposed application.

Note that the text of the answers is genuine and we only corrected the grammatical mistakes and word order.

Write down 1–3 (or more) most important features you would welcome in the new application.

Answers
• Customization of the rules and units. • Account creation for sharing and collaborative work.
• Tuning the calculation with the cost of my infrastructure.
• Customize the calculation based on the local needs. • Ability to export (esp. with an Excel compatible format) the results for further processing. • Save the calculation (in browser).
• Persistent configuration of the calculation.
• Although it might be difficult to incorporate, it would be useful to get more accurate prices depending on the country you store the data in.
• Configure the calculations.

Table 1.3: Textual answers no. 1

We can summarize that the most wanted features are the ability to customize the calculations and have an option to persist the configurations and preferences. The users also suggested some more advanced functionalities like calculating the prices depending on the country. However, if we look at this suggestion from a higher level, we can achieve it by giving the user the freedom to configure their own calculations. However, the configuration options would have to be general enough to achieve it.

State 1–3 (or more) aspects of the current application that prevent you from achieving your goals or do not let you work efficiently.

Answers
• No possibility to export and no memory. • Always need to fill in the values from scratch.
• No option to save the configuration of the variable.

Table 1.4: Textual answers no. 2

The negative aspects of the current application closely relate to the suggestions (for the features of the new application) the respondents gave in the previous question. In summary, the respondents miss the option to export the results and to persist the calculation definitions within the application.

What are the most important features of the current application that you or your colleagues find useful?

Answers
• The design of the UI is intuitive.
• The application is extremely useful to explain the cost of a service to new users.
• The intuitive calculation where the main power is in the formulas.
• Application has a logical structure.
• Price variation of the different types of storage is very useful.

Table 1.5: Textual answers no. 3

The users find the design of the application intuitive and useful. The logical structure and division of the results into the categories make the application useful for explaining how the calculation was conducted. The main message to take from this feedback is to design the new application in a way where the calculations designer would be able to define and show the end-user intermediate results with a possible explanation of how the calculation was done.

Write down any other ideas, recommendations, observations, or notes regarding the new application.

Answers
• It could also be nice to change the branding (mainly colors) of the app for the specific institution when they deploy their own instance.
• Import of the estimated data volume from DMP (Data Management Platform).

Table 1.6: Textual answers no. 4

We can relate the first suggestion to the functionality where the user can create their own account in the application and potentially be a part of the organization. This organization would probably be managed by a user with the role of owner. We can potentially add functionality where the user would be able to change the colors of the UI or change the style of different elements of the application (e.g., organization logo).

The second suggestion can be linked with the functionality to import and export data to and from the application. We should think about different formats of the data we want to support for import/export. If we design this functionality generally enough, it would be no problem to add support for the new formats in the future dynamically.

1.4.3.1 Summary

We can summarize that the suggestions from the respondents are similar to the suggested features we introduced in the Likert scale questions. The users would like to have an application that would enable them to adjust the computations according to their needs and to have the ability to persist the configurations and preferences.

The respondents also stated that the UI of the current application is well-designed, and they appreciate different aspects of it. We should take the current design into consideration while designing a new application.

We also received a few custom recommendations from the respondents on how to enhance the features of the current application.

1.4.4 Overall Summary

The survey gave us important insights from the users on how to properly design the new application and what should be included in the new FRs. We also received positive feedback related to the new suggested features we would like to include in the new application.

The respondents also marked the features and aspects of the current application they find positive and useful but also negative and missing. We should take this feedback and use it in the design process of the new application by keeping the best from the current application and implementing the missing parts.

The data from the survey will be the basis of the decisions we will make while constructing the new FRs. The feedback from the users is very important since it will also be these users who will use the new application.

1.5 Analysis of Existing Solutions, Methods, and Technologies

In this section, we will look at some of the existing methods and technologies regarding configurable computations and the creation of custom calculations. We will go through the particular solutions and analyze them.

1.5.1 Spreadsheets

One of the popular formats to define custom calculations is the usage of spreadsheets. Nowadays, we have multiple providers of spreadsheet technologies. The most well-known include Microsoft Excel [19], LibreOffice Calc [20], Google Sheets [21], and many more.

We will base our analysis on the LibreOffice Calc Guide [22], which offers basic and advanced explanations of the LibreOffice spreadsheet technology. So, first of all, what is Calc? The guide gives us this explanation:

"Calc is the spreadsheet component of LibreOffice. You can enter data (usually numerical) in a spreadsheet and then manipulate this data to produce certain results.

Alternatively, you can enter data and then use Calc in a 'What if...' manner by changing some of the data and observing the results without having to retype the entire spreadsheet or sheet."

The spreadsheet has many various components. The main component is the main window in which the spreadsheet resides as we can see in Figure 1.19.

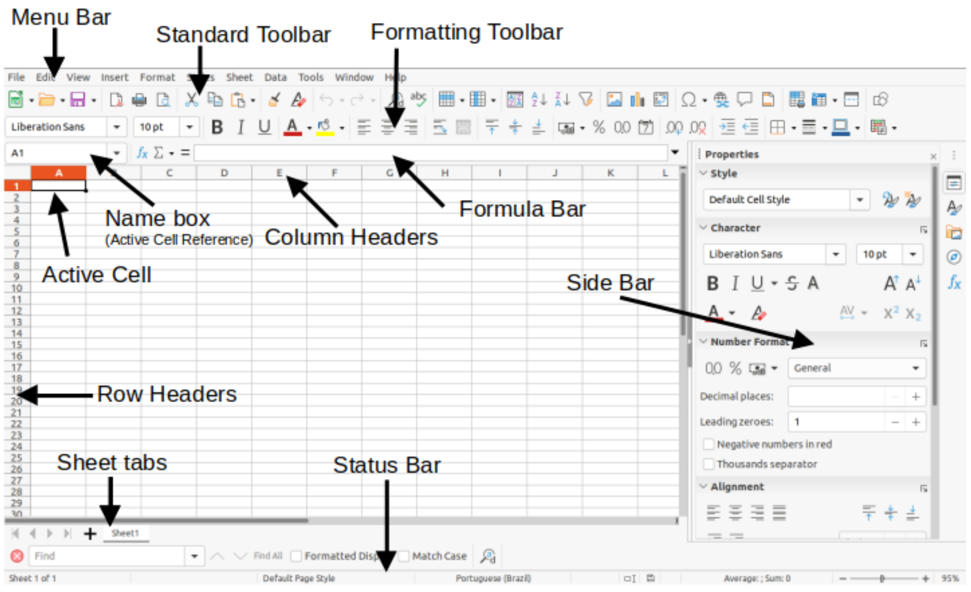


Figure 1.19: LibreOffice Calc main window [22]

We will assume that you have a basic knowledge of spreadsheet technologies, so we won't define each particular component of the spreadsheet. Instead, we will analyze only components of the technology relevant to the ACE application's design process.

The spreadsheet is composed of rows and columns. Each combination of a row and a column represents a single cell. If we simplified things, we could say that a cell can either hold a value or/and represent a result.

The value can have multiple different formats like text, date, a boolean value, etc. The cell can also be seen as an input field that expects a value from the user.

A formula represents the result. The formula is a composition of operators, values, references to the cells, and other complex expressions. Table 1.7 shows us some examples of the Calc formulas.

The result of the formula is stored in a cell where we define the formula. The formula language is a type of Domain Specific Language (DSL). The spreadsheet is only one of the possible ways to create and store the formulas. The Calc DSL could also be used using different front-ends, e.g., web browser, and input/output HTML fields. We also emphasize

Formula	Description
`=A1+10`	Displays the contents of cell `A1` plus `10`.
`=A1*16%`	Displays `16%` of the contents of `A1`.
`=ROUND(A1,1)`	Displays the contents of cell `A1` rounded to one decimal place.
`=SUM(B8,SUM(B10:B14))`	Calculates the sum of cells `B10` to `B14` and adds the value to `B8`.

Table 1.7: Examples of the Calc formulas

that spreadsheet formula DSLs are quite similar across different spreadsheet technologies (Microsoft Excel [19], Google Sheets [21], etc.) due to the simplicity of the syntax and ability to define your own functions using macros.

1.5.1.1 Web APIs

Cloud spreadsheet technologies like Google Sheets [21] are implemented as Software as a service (SaaS) [23]. These technologies often come with web APIs, so we can integrate them with the other systems. Google offers Sheets API [24] to read and write data from/to spreadsheets programmatically. We can, for example, write data to the cell:

```javascript
var values = [
    [
        // Cell values ...
    ],
    // Additional rows ...
];
var body = {
    values: values
};
gapi.client.sheets.spreadsheets.values.update({
    spreadsheetId: spreadsheetId,
    range: range,
    valueInputOption: valueInputOption,
    resource: body
}).then((response) => {
    var result = response.result;
    console.log(`${result.updatedCells} cells updated.`);
});
```

Code Snippet 1.1: Example of writing data to the cell in Sheets API [24] using JavaScript client library

1.5.2 Website Calculator Builders

Another example of adaptable/customizable computations technology is website calculator builders. The builders usually offer the functionality to take advantage of an interactive canvas and design your own calculator using custom input and output fields. We can see an interactive canvas example using the uCalc [25] app in Figure 1.20.

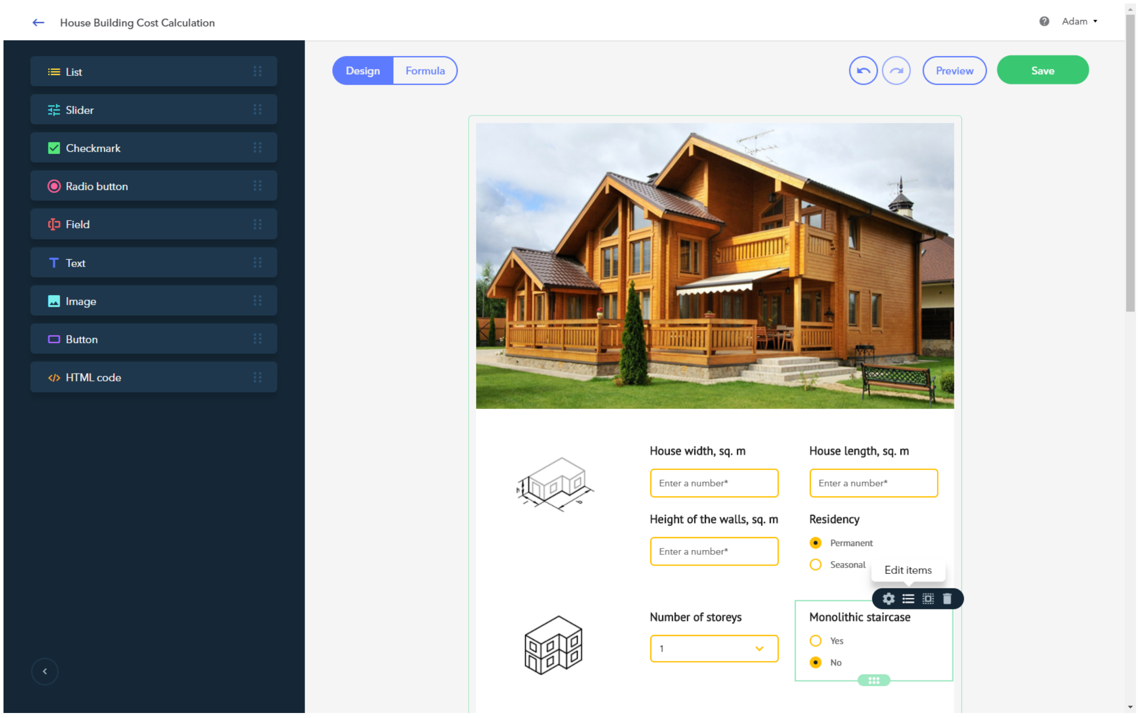


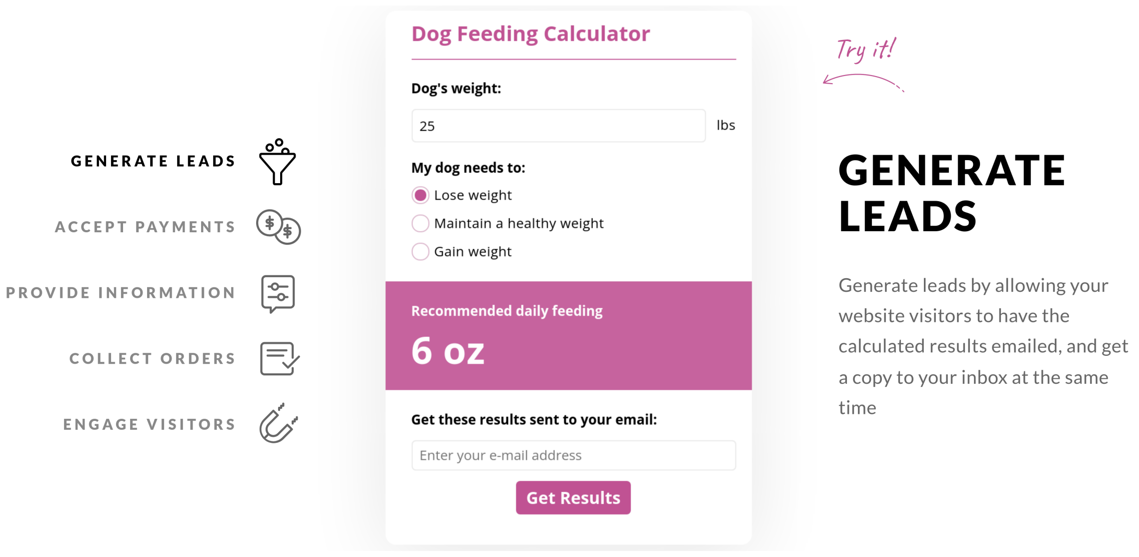
Figure 1.20: uCalc UI example [25]

Howard Steele summarizes some of the niches, in his article regarding website calculator builders [26], where we can frequently find usage of this technology:

- logistics,
- insurance,
- loans,
- real estate,
- and construction industry.

Since these niche companies often come with their own websites, it is important to have the ability to embed the calculator into the external system. It is mostly done by including the JavaScript (JS) code generated using the website calculator builder app. We can see an example in Figure 1.21.

The builders work similarly to the spreadsheets. You draw and design your input/output fields, give them proper formatting, and bind the input fields to the output fields by creating a formula for the computation or interactively describing it.



Dog Feeding Calculator

Dog's weight:

25 lbs

My dog needs to:

- [✓ Lose weight]
- Maintain a healthy weight
- Gain weight

Recommended daily feeding

6 oz

Get these results sent to your email:

Enter your e-mail address

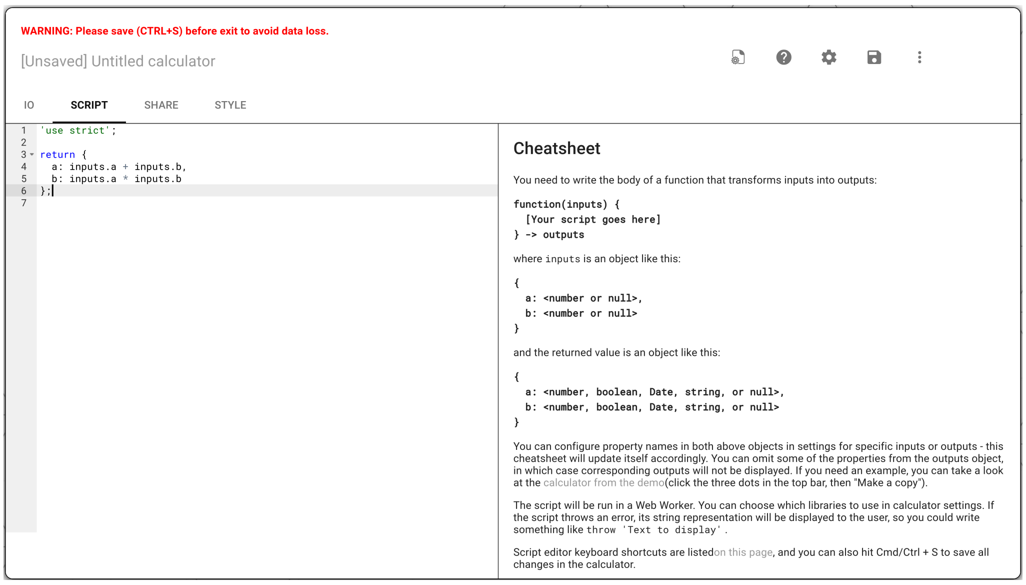
GENERATE LEADS

Generate leads by allowing your website visitors to have the calculated results emailed, and get a copy to your inbox at the same time

Figure 1.21: Embedded website calculator using Calconic App [27]

There are a lot of other apps similar to uCalc. Some of the most well-known include Calconic [27], Calculoid [28], or ConvertCalculator [29].

Another similar approach takes the JSCalc app [30]. This builder takes advantage of the JavaScript programming language to define the mapping between input and output fields as we can see in Figure 1.22. These fields are, on the other hand, interactively defined using another part of JSCalc UI.



WARNING: Please save (CTRL+S) before exit to avoid data loss.

[Unsaved] Untitled calculator

```
'use strict';

return {
  a: inputs.a + inputs.b,
  b: inputs.a * inputs.b
};
```

Cheatsheet

You need to write the body of a function that transforms inputs into outputs:

```
function(inputs) {
  [Your script goes here]
} -> outputs
```

where `inputs` is an object like this:

```
{
  a: <number or null>,
  b: <number or null>
}
```

and the returned value is an object like this:

```
{
  a: <number, boolean, Date, string, or null>,
  b: <number, boolean, Date, string, or null>
}
```

You can configure property names in both above objects in settings for specific inputs or outputs - this cheatsheet will update itself accordingly. You can omit some of the properties from the outputs object, in which case corresponding outputs will not be displayed. If you need an example, you can take a look at the calculator from the demo(click the three dots in the top bar, then "Make a copy").

The script will be run in a Web Worker. You can choose which libraries to use in calculator settings. If the script throws an error, its string representation will be displayed to the user, so you could write something like `throw 'Text to display'`.

Script editor keyboard shortcuts are listedon this page, and you can also hit Cmd/Ctrl + S to save all changes in the calculator.

Figure 1.22: JSCalc JavaScript canvas [30]

1.5.3 DSLs and Scripting Languages

DSL is a high-level programming language optimized for a specific class of problems. A DSL uses the concepts and rules from the given field or domain. [31]

The canonical example of the DSL suitable for the computations is the formula language used in the Calc we described in Section 1.5.1. DSL is a very general yet powerful approach for expressing complex definitions, relations, and calculations conveniently that can be programmatically processed.

Another powerful and popular technology is scripting languages. There are many different definitions of the scripting language. If we wanted to generalize them, we could say that scripting language is a high-level and lightweight programming language that is interpreted rather than compiled ahead of time. It is often a general-purpose programming language, or it may be limited in a specific way. [32]

There are many different scripting languages out there. One of the most commonly known is JS which is widely used in modern browsers. As mentioned above, JSCalc, for example, uses JS to define mappings between input and output fields.

One of the very specific usages of the scripting languages is their embedment into other general-purpose programming languages. For example, there is an implementation of JavaScript in the Java programming language called Rhino. Rhino contains JS compiler, shell, and debugger. This feature enables developers to store JS scripts in the Java variables or files and execute them directly from the Java code. [33]

Another interesting lightweight scripting language is Lua [34]. There are also implementations into other programming languages like Lupa to CPython [35] or HsLua [36] to Haskell programming language.

The main benefit of these embedment libraries is using an existing syntax in the given computation use-case or domain. For example, we don't need to define a whole new DSL for computation formulas, but we can take advantage of the existing scripting language. Users can afterward interactively define these formulas by using web UI, for example. The JSCalc app uses a similar approach using JavaScript language. We can transport the script using the HTTP protocol, evaluate the script using a given back-end (Java, Haskell, etc.), and send back the results.

1.5.4 Analysis Results

The goal of the analysis was to research existing technologies. We analyzed how different solutions use different types of methods to take input values of the calculations and transform them into the output results based on the predefined rules. Namely, we discussed three types of calculation technologies:

1. spreadsheets,
2. website calculator builders,
3. DSLs, and scripting languages.

Each type of technology uses a slightly different approach to capture input values and present them to the user. However, we can see a common pattern in the calculation flow represented in Figure 1.23.

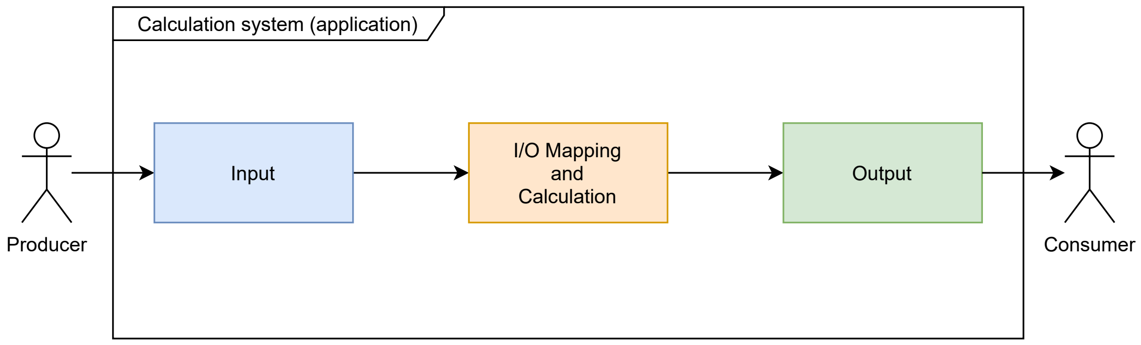


Figure 1.23: Common calculations flow

The given diagram is a simplified version of the common calculation flow. The important thing for us is the realization that all of these technologies use the same principle or very similar high-level approach on how to realize calculations. We can summarize it in the following steps:

1. The producer provides the system with the input values.
   a) It can be realized manually through UI or programmatically by another application.
2. The application takes the input and represents it in the computer-readable format.
   a) It can be, for example, parsing of text from the given format (JSON, XML, etc.) to the applications' type system.
3. The application takes predefined rules and calculates the result based on them. It maps the input values to the output values in the process. The producer can provide the definition of the rules.
4. The application transforms the results into a format readable by the consumer.
   a) Results can be rendered to the UI or sent to another application.
   b) The consumer and producer can be the same entity.

5. The consumer reads the results.

This summary (process) can be used as a high-level guideline in the design process of the calculation system similar to the ones we analyzed in this section. The particular implementation decisions in each step should be carefully considered and designed to fit best to the given domain.

1.6 Adaptable Costs Evaluator (ACE) Functional Requirements

In this section, we set the FRs on the new ACE app. The requirements are based on the analysis described in the previous sections.

1.6.1 Methodology

Each FR we include in this section has its own ID, definition, and *basis*. The basis is a textual explanation describing the reason why we include this functional requirement in the list. Most of the FRs are based on the previous analysis, as we described above.

Furthermore, FRs are divided into several groups based on their common characteristics. We add a textual description of each of the groups, so the reader can better grasp why we grouped the particular FRs together.

The FRs we present here are based on our preliminary analysis as described above. It is worth mentioning that the quality of the application design is highly based on the feedback from the users. Thus, these FRs may be updated during the next development phases based on the feedback from the users.

1.6.2 Group No. 1 – Computation Functionalities

The FRs in the first group, in the Table 1.8, describe the computation capabilities and functionalities of the application. They are related to the aspects like how to conduct the computations, what are the results of the computations, or how we describe the computations.

Group No. 1		
FR No.	**FR Description**	**FR Basis**
FR 1.1	The user is able to adjust how the calculations are performed. This would mean implementing a functionality where users could create their own input fields and corresponding output fields and mapping between them (calculation definition).	Based on the survey, we can see that the users (strongly) agree that they would welcome the ability to adjust how the calculations are performed. The respondents also identified this need in the textual answers. We can also take some inspiration from the existing solutions we described in the previous analysis (e.g., website calculator builders).
FR 1.2	The user is able to create/import their own set of rules and conditions for the calculations. This FR is closely related to FR 1.1. This FR describes the need to have a formal language to describe the formulas, conditions, and Input/Output (I/O) mappings to produce the computation results.	Again, based on the survey, the users (strongly) agree that they would welcome this functionality. Again, some respondents also identified this need in the textual answers. Also, based on the demographics data, this would probably be an acceptable functionality in the means of learning to use it. We base this assumption on the fact that most of the respondents have completed a Ph.D. or higher education and are skilled with computers. Most of the existing solutions also implement some form of formal language to describe the computations (either textual or graphical). This is also an important FR that should be considered when designing the architecture of the new application.

Table 1.8: FRs group no. 1

1.6.3 Group No. 2 – Supportive Functionalities

The supportive functionalities serve to enhance the experience and workflow when using the app. These are functionalities like creating your own account or exporting the results in the given format.

Group No. 2		
FR No.	**FR Description**	**FR Basis**
FR 2.1	User is able to create their own account and save the preferences, calculation results, etc.	We base this FR on the feedback from the survey. The respondents (strongly) agree that they would welcome this functionality. The respondents also identified this need in the textual answers.
FR 2.2	The user is able to create an organization within the application and invite other users to it.	Again, this FR is based on the feedback from the respondents. They would welcome the ability to collaborate with the other users within the organization – share the calculation definitions, results, etc. This would require implementing a logical unit within the application that would group together multiple users – it would be a form of multi-tenancy [37].
FR 2.3	There is a role system with the admin, owner, and regular roles (example names).	This FR is derived from 2.1 and 2.2. The admin would be someone who is responsible for maintaining the whole instance of the application. The owner would be a person responsible for the given organization. They would be able to invite other users to the organization. The regular would be a standard user who uses the computation functionalities of the app. There would also be an option to set the public visibility of the particular computation. The users can encounter a case where they would like to share a particular computation with someone who doesn't use the application on a regular basis. This external user would have limited capabilities (no option to save the results, edit the computations, etc.).

FR 2.4	The user is able to export the calculation results in the given format (CSV, XML, etc.).	We base this FR on the feedback from the respondents. They marked that it would be useful for them to have the ability to export the results for further processing (e.g., MS Excel [19]).
FR 2.5	The user is able to interact programmatically with the application using the web API.	At least half of the respondents would welcome this functionality. The web API option is not entirely a FR. Still, we include it here since the question in the survey was formulated in the context of the API being a functionality of the application.

Table 1.9: FRs group no. 2

1.6.4 Group No. 3 – UX & UI Structure

The third group contains FRs that are related to how we should design and structure the UI so the users have a good experience when using the app. It also relates to the fact that proper UI design can enhance the effectiveness of the users while using the app.

Group No. 3		
FR No.	**FR Description**	**FR Basis**
FR 3.1	User can set default values of the custom fields defined by them.	The current application fills in the default values of the prepared input fields on the page load. We should let the users define the default values of the custom-prepared computation fields. Some input fields may never change their value, so it would be faster to prepare a particular computation using the default values. Also, one of the respondents defined that the application is extremely useful to explain the cost of service to the new users. This functionality can support this aspect.

FR 3.2	Let the user group the I/O fields by the different categories.	Also, the current implementation logically divides the I/O fields into several categories based on their characteristics. The respondents of the survey defined that this is one of the important features of the current application.
FR 3.3	The application recalculates the costs on every change of an input field.	This is one of the features of the current application. This FR closely relates to the application architecture (e.g., SPA). It would be convenient to implement this FR since it allows the users to dynamically see the impact of changing one or more input fields. The users also agree that it is intuitive to work with the UI of the current application, so we can assume that it won't be a problem to incorporate this functionality to the new application.

Table 1.10: FRs group no. 3

1.6.5 Summary

The functional requirements we set here will be the base for the architecture design and implementation of the new application. We covered UI & UX, computation, and supportive FRs. Together, they cover the functionalities from the back-end to the front-end of the new proposed ACE application. After the successful implementation of the new application, we will summarize the implemented FRs from this list.

Chapter 2

Application Design

In the following chapter, we will take a look at the design of the new ACE application. We will mainly focus on the proposed architecture and describe how the back-end functionality works. We will also describe database design and the tables that the back-end of the application uses.

2.1 High-level Architecture Design

This section describes a high-level architectural design of the proposed application. The design is mainly based on the conducted analysis from the Chapter 1. We also tried to follow the best practices in the given field.

Based on the results of the analysis, we decided that the main purpose of the newly proposed application is the ability to create your own set of computations. This feature is also enhanced by the ability to create these computations in a flexible or adaptable manner where the user has full control over what are

1. the input fields to the computations,
2. defined computation rules,
3. and where and how the results should be stored.

Yet again, we can use the Figure 1.23 from the Chapter 1 to describe these 3 steps. We mention this diagram because, from a high-level point of view, our new application won't be too different from the existing solutions and methods we described in the Chapter 1. Nevertheless, it is important to design implementation details of each step properly so the application works correctly as a whole. The implementation details will be further discussed in the Chapter 3 describing the implementation decisions.

We will now look at the high-level architecture and break it down to the back-end and front-end parts. We will look at the design from the technical point of view using language agnostic explanations.

2.1.1 Back-end Design

We decided to implement the solution as a web application. The back-end is accessible using standard HTTP/HTTPS communication and methods. The decision to implement the application using web technologies comes from the description of the assignment on the one hand. However, we also believe that web technologies are a suitable decision because of the ease of integrating the application to other systems and using it without the necessity to install the application locally. Also, based on the results from the users survey described in Section 1.4, half of the respondents agree that they would welcome the option to integrate the application into their own internal systems. The HTTP protocol is also widely used and standardized.

The interface to the back-end (API) is implemented by following the REST architecture and its principles. The specification of the API will be specified in the Chapter 3. For the transportation format of the data, we decided to use the JSON format. The REST architecture has many advantages, including statelessness, caching, uniform interface, or independence from the particular data format, which can benefit us in implementation and further scalability. [4]

The REST API contains a set of standard endpoints for handling users, organizations, and similar common resources. However, the core functionality of the API is enabled by the resources representing the **computations** and other associated resources. We decided that the computations will be represented through the **formal language(s)** that serve as computation formulas mapping input to output. The back-end is able to read this formal language(s), evaluate the formulas, store the definitions and results and return the results to the client.

2.1.1.1 Web Service Aspects

The back-end of the application can be defined as a web service. It has a standard interface that can be used independently of the hardware or software on which it is used. Another important aspect of the back-end design is that it is completely independent of any front-end(s). Many monolithic web application designs combine both back-end and front-end in one codebase. [38]

We wanted to avoid the design combining all layers in one application and use a separate back-end because of the cloud-native principles we want to follow. It enables us, for example, to create multiple possible front-ends for the application and integrate it into the different systems relatively quickly. [39]

2.1.2 Front-end Design

The front-end of the application enables users to interact with the functionality of the back-end by using its web REST API. We decided to mainly focus on the implementation of the back-end functionality, so we will not present a concrete front-end design here.

However, since the back-end is designed as a web application, it would be reasonable to take advantage of the web technologies and implement the front-end using the existing web development tools. Under the tools we mean the standard package of web development technologies such as HTML, CSS, and JavaScript. Since the elements of such application would most probably be interactive, it would also make sense to take advantage of the modern front-end JavaScript frameworks such as React [40], Vue.js [41], Elm [42], etc.

2.1.3 Conclusion

We described a high-level design of the newly proposed application. We concluded that we would follow cloud-native principles while designing and implementing the application. The implementation model can also be described by a popular term SaaS [23]. In Figure 2.1, we illustrate the design we talked about.

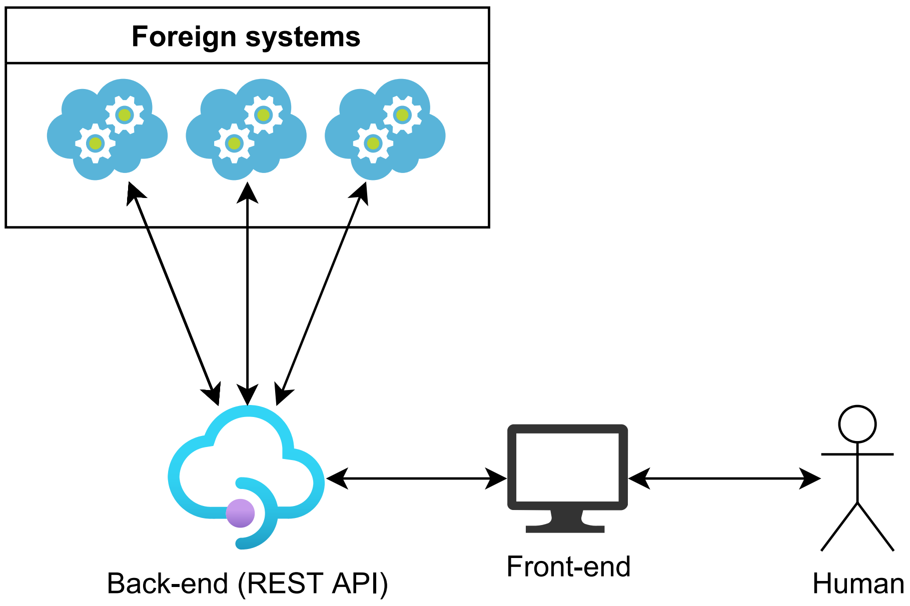


Figure 2.1: ACE high-level architecture design

We can see that the resulting back-end is possible to use to implement a front-end to be used by people. On the other hand, different foreign systems are also able to take advantage of the API for integration purposes.

2.2 Back-end Functionalities & Database Design

In the following section, we will conceptually describe the functionalities that we implement in the back-end of the application. The concrete implementation details such as chosen technologies and methods will be described in Chapter 3.

The functionalities are based on the back-end functional requirements we set in the Section 1.6. The main challenge here was to design a back-end that would mainly allow us to have:

1. a system of users and organizations grouping the users (FR 2.1 & FR 2.2),
2. adaptable computations that the users can share with each other (FR 1.1 & FR 1.2),
3. and a role system that would define permissions for the resources of the users and resources shared within the organizations (FR 2.3).

Beside the mentioned FRs, the architecture of the back-end also conforms to the FR 2.5 by enabling the clients to take advantage of the REST API as described in Section 2.1.

We decided to describe both the business logic (back-end functionalities) and database design together. This decision is based on the nature of the application where it would be redundant to, for example, first describe the business logic using a class diagram and then use a similar diagram to describe the database design. We will use diagrams to describe how the database is structured and how the associations between the different resources (tables) conform to the application's overall design.

Since the diagram with all resources (tables) would be too big to fit into the page and described, we decided to split it into two parts:

1. users and organizations,
2. and computations resources (tables).

Note that all tables have two additional columns that are not shown in the diagrams – `inserted_at` and `updated_at`. These columns are timestamps we use to track down when the particular record was created and updated at. They are omitted from the diagrams to save space. For clarification, we include the list of used symbols and abbreviations in diagrams with the explanations:

- PK – primary key
- FK – foreign key

- UN – unique index
- `*` – mandatory attribute
- `o` – optional attribute

2.2.1 Users & Organizations

We start with Figure 2.2 that describes users, organizations, their relationships, and role system in our application.

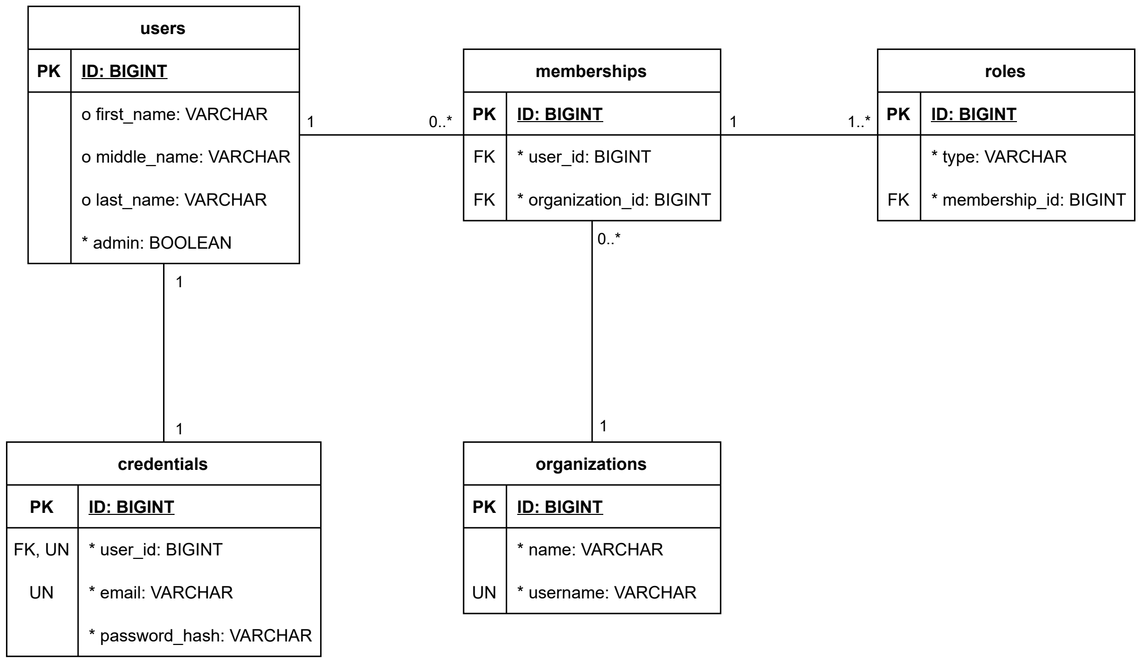


Figure 2.2: Users & organizations database diagram

The **user** is a core resource in the application. It represents an actual user using the application. The user comes with a set of standard attributes like first and last name. A user can be an admin – a special role within the system with full permissions over all other resources. Each user has **credentials** which are used to authenticate the user within the application. The authentication system is enabled by the combination of `users` and `credentials` tables. We will talk more about this in Chapter 3.

Another important resource is the **organization**. The organization groups multiple users together. This is an important feature of the organization because it enables the database multi-tenancy model [37] and sharing other types of resources between the users within the organization.

A user can be part of multiple organizations, and an organization can have multiple users. This property is enabled through the **memberships**. Membership is a simple resource that defines the belonging of the particular user to the particular organization. Another important connected resource is the **role**. Each role record is connected to the particular membership. It defines the role of the user within the organization, and thus, its permissions over the resources within the organization. A user can have multiple roles through a single membership.

Different types of roles are defined by the `type` attribute. There should always be a predefined enum of the role types that the application supports. However, this enum can be gradually updated, and the role system can grow on its complexity. Yet again, we will talk more about the roles in Chapter 3.

All of these described resources together form a design that can be used for the authentication, authorization, role system, and enabling the multi-tenancy within the application.

2.2.2 Computations

The computations and other connected resources and their associations are described in Figure 2.3.

The main resource here is the **computation**. The main purpose of the computation is to group other related resources together. It is always owned by the particular user, which is defined by the `creator_id` attribute. The computation can also be shared within an organization. It can be related to the spreadsheet in a classic office program.

The computation consists of the inputs, outputs, and formulas. The **input** is a resource that describes an input value from the user that can be used in the computation process. The format and value of the input are defined by the linked **field schema**. A field schema record holds a valid *JSON Schema* [43] definition that can be used to validate the value of the input. That is why the data type of the `last_value` attribute is `JSONB`. This way, the values can have simple data types like string or integer but can also hold more complex data types like arrays or user-defined objects.

The second important resource is the **output**. The value of the output is also defined by the connected field schema. It describes the output of the evaluation of a particular **formula**. The formula has a `definition` that can reference multiple inputs through the unique labels of those inputs. The definition of the formula can be evaluated by using the connected **evaluator**. The result is then written to all outputs that are linked to the given formula.

The evaluator is an important resource that describes a particular formula evaluation implementation. The application can implement multiple different evaluators. This property enables us to use evaluation engines either that can be implemented directly in the back-end, but that can also be in the form of foreign systems accessed through the foreign APIs. In this way, the application and its evaluator engines can be easily extended. The

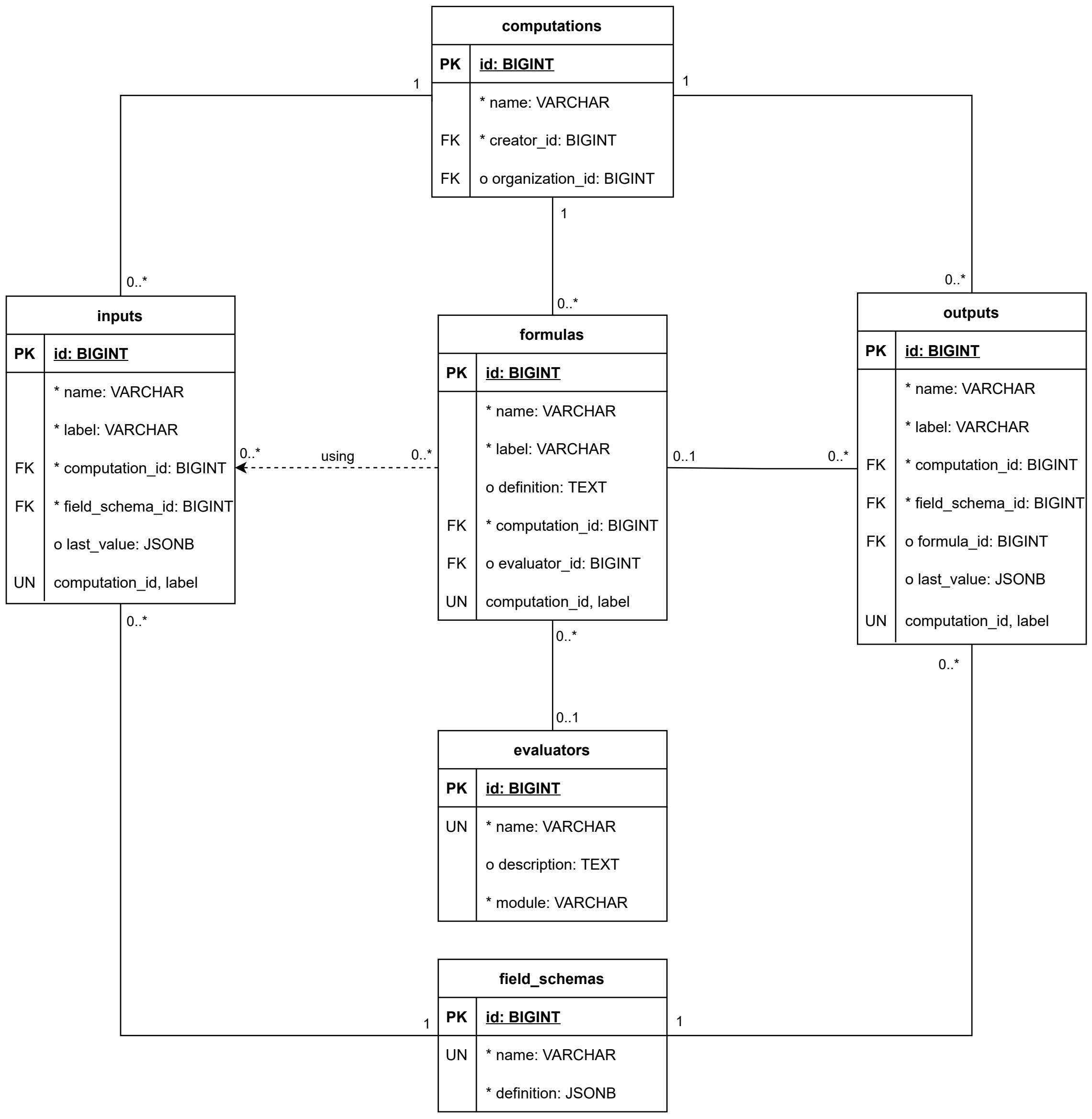


Figure 2.3: Computations database diagram

definition of the formula has to have a compatible syntax with the linked evaluator.

The resources we described form an adaptable design that can be used to implement a system of adaptable computations. To conclude, this aspect is mainly enabled through:

- I/O data types that can be dynamically added and removed to and from the application using the JSON Schema technology,
- formulas that can hold computation definitions referencing the inputs through the

unique labels,

- and most importantly through the usage of different implementations of the evaluators that can be used to evaluate formulas and to bring support for the usage of different evaluation techniques (using complex data types, etc.).

2.3 Use of Functional Programming

One of the thesis requirements is to use functional programming languages for the implementation. Functional programming is a programming paradigm where developers use functions as the primary way to organize the code. This programming style has certain aspects like immutability, pure functions, higher-order functions, or treating functions as first-class citizens. [1]

We will not dig too deep into the nuances of FP, but it is worth mentioning some of the main benefits of the FP in the development:

1. **Pure functions** – these types of functions are one of the main concepts of FP. A pure function is similar to a mathematical function where the same input always results in the same output. It means that the pure function is not dependent on a global state and doesn't alter it. This fact makes the writing of tests much easier because the tests are always deterministic.
2. **Easy debugging and fewer bugs** – The functions are a simple mapping of an input to an output. This means that searching for a problem in the code means looking at the stack trace and analyzing each level of the function call.
3. **Function signatures** – Pure functions depend on the input you provide them with. The function signatures are thus more transparent and also serve as a form of documentation for developers.
4. **Safe concurrency** – The pure functions do not depend on the shared state. The isolation thus provides a safe way to run multi-threaded code.
5. **Reasoning about the code** – The main programming technique in FP is composing functions and creating higher-order functions. Reasoning about the code that is a composition of functions is straightforward, and imagining the execution tree of the code is much easier. [44]

These were just a few aspects and pros of FP. There are many more, and, of course, there are also many functional programming languages. In fact, functional programming is not restricted to functional languages. It is a coding style. It means that you can also implement this style in some of the object-oriented languages like Java or Python. However, some programming languages require you to use this style of programming.

Some of them are very strict in forcing this style, like Haskell, for example. Some of them give you more freedom, like Elixir [45], for example. The decision of which language to choose for your project always depends on your needs, the time you have for the project, and the level of expertise you have with functional programming. In Section 3.1 we will describe which functional programming language we chose and explain why we decided so.

2.4 Conclusion

In this chapter, we looked at the high-level architectural design of the application. We focused on the back-end part of the application and described why we chose to implement the application as a web app with the implemented REST API.

In the second part, we talked about the back-end functionalities and database design. Through the database diagrams, we described, from a high-level point of view, the business logic the application implements. The goal was to show how we approached and solved the problem of the adaptable computations and scalable design of the back-end in this domain.

Chapter 3

Implementation

In this chapter, we will look at the back-end implementation of the ACE application. We will focus on used technologies, dig into the important implementation details, go through testing and documentation and look at the FRs from the Section 1.6 in the context of the current implementation. The back-end code is part of the Appendix B but is also accessible via GitHub [3].

3.1 Used Technologies

One of the parts of the thesis assignment was to develop the application with the use of functional programming languages. The goal here was to choose the programming language that would be suitable for the task as we designed it in Chapter 2. The number of possible functional languages to choose for your project is quite big nowadays. We have languages like Haskell, `F#`, Scala, Clojure, and many more. We were looking for a functional language that would pass these requirements:

- **Stable community** – it is very important to have a stable community that pushes the language forward and develops and maintains a wide spectrum of libraries.
- **Mature tooling** – since this attribute can make the development much more efficient. We are talking about stuff like language servers, build tools, documentation, IDE support, etc.
- **Quick prototyping** – this requirement was important for us because of the nature of the application. Quick prototyping comes very handy in web development, where developers make a lot of small changes in a short period of time.
- **Mature web framework** – we were looking for a language with an existing mature web framework. This was very important because a mature framework with well-defined best practices can eliminate a lot of implementation that doesn't directly relate to the original task.

After some analysis, we decided to choose the **Elixir** programming language. Elixir is a dynamic and functional language that is based on the Erlang programming language and runs on the Erlang VM. We chose this language because it passes all requirements we set before. It is a language with a wide community used by commercial companies. It comes with a set of development tools for building, static code analysis, and many more. It is also supported by IDEs we used for the development. Since the language features dynamic typing, it is suitable for quick prototyping. [45]

Besides the language itself, there is the **Phoenix Framework** we used for the project. Phoenix is a mature production-ready web framework that is used by a number of companies. It follows the principles of MVC but uses functional programming to keep the code clean. It also comes with a set of scripts, enabled by the Elixirs' **mix** build tool, that can generate a lot of repetitive code for you, like defining the mapping between the database tables and code data structures. Phoenix also defines a set of conventions for structuring your code that help you to keep it maintainable and avoid unnecessary repetitions. The framework puts its focus on your business domain. Another very important aspect is that Phoenix comes with a number of guides that can help you with the initial development but also with solving a lot of common web development problems like authentication, authorization, etc. [46]

There are also companies like DockYard that offer expert consulting for the Elixir and Phoenix projects. [47]

3.1.1 Database Engine

Based on the design from Chapter 2, we decided it would be most suitable to use a relational database for our application. We picked PostgreSQL as Relational Database Management System (RDBMS) we will use. It is an open source database system with over 30 years of active development. We picked it mainly for its reliability, robustness, and performance. We also take advantage of its complex data types like `JSONB` that we used in the back-end design of the application as it can be seen in Figure 2.3. [48]

3.1.2 Docker

As we described in Section 2.1.1.1, the back-end of the application can be seen as a web service. To fully take advantage of the cloud-native principles, we decided to virtualize the application using Docker technology [49]. Since the application also uses PostgreSQL database, we use Docker Compose [50] for building and running the application container and connected services like the mentioned database engine.

Docker can be used for the development, testing, and also for production deployment. The application image stays the same. The only difference is in the configuration, which is manipulated using the environment variables.

3.2 Implementation Details

The implementation of the back-end is based on the design we proposed in Section 2.2. Most of the aspects of the implementation were already described in that section. However, there are still some details/specifics that are worth mentioning and describing in greater detail.

In this section, we will take a look at the implementation details like authentication, role system, or a concrete implementation of the formula evaluator.

3.2.1 Authentication

Figure 2.2 shows us that the application implements authentication using email and password. This is a system that is used in the back-end to authenticate the particular user. However, it would be inconvenient and potentially insecure to include email and password in each HTTP request the back-end would receive. We decided to address this by using JSON Web Token (JWT) to authenticate the API calls [51]. The token is then transported using HTTP *Authorization* header as the *Bearer* token as described in RFC 6750 [52].

There are two ways to obtain the token for the particular user. Firstly, the token is returned in the HTTP response for the endpoint that creates a new user. This is a convenience, so a new user can start using authentication for other endpoints right after it is created. The second way is to use a special endpoint for signing in the user. The sign-in request requires the client to send the users' email and password in the body of the request. The server responds with the newly generated token that can be used to authenticate the user. The application should only be accessible through secured HTTPS communication in the production to avoid leakage of emails and passwords of the users.

3.2.2 Authorization & Role System

Since we work with the users and organizations grouping the users in the application, we needed to implement some form of authorization in the application. The design described in Section 2.2 allowed us to implement a role-based system for the authorization.

This system consists of two important parts – authorizing resources owned by a user and resources that are shared within the organization.

The first part is quite straightforward. The user is able to fully manipulate the resource describing itself – user resource. It is also able to fully manipulate the computations which the user created.

The second part involves the organizations. As we can see in Figure 2.2, the user can have multiple roles connected to the particular membership in the particular organization.

This system is quite flexible, and new roles can be introduced through the implementation in the application. Currently, the application supports these three roles:

1. **Regular** – a user who is able to see other users in the organization and read and manipulate the computations shared within the organization.
2. **Maintainer** – a user that can also invite other users to the organization and manage users with the regular role.
3. **Owner** – owner of the organization that can manipulate the organization resource itself and fully manage all other resources within the organization.

There is also a special role of a user, and that is the admin role. If a user is an admin, they are able to fully manipulate all resources within the application. This role is intended for the administrators of the application.

The system is not extensively complex but, by design, is open to being extended by new roles and new authorization rules that can address more complex authorization requirements in the future. More information about the authorization and role system can be found in the *Adaptable Costs Evaluator Wiki* [53].

3.2.3 Simple Evaluator Implementation

As we discussed in Section 2.2, the application can be extended by the implementations of the various types of formula evaluators. Since the application would be hardly usable without at least one implementation, we decided to include the implementation of the *Simple Evaluator* supporting multiple data types. This evaluator is based on the *Abacus script language* [54].

Documentation and capabilities of this script language can be found in its documentation at GitHub [54]. To mention a few of the capabilities, Abacus supports

- basic mathematical operators (`+`, `-`, `*`, `/`,`^`),
- boolean operators (`&&`, `||`, `not`),
- factorial, and some of the other mathematical functions.

To enhance the implementation, the *Simple Evaluator* is also able to parse inputs from the formula definition that are defined by the unique `label` attribute – variable name in a sense. For example, we can see a valid formula definition that can be used and evaluated in Code Snippet 3.1 where `input1`, `input3`, and `input4` are labels of the actual input records defined by the user.

```
((input1 + input3) ^ 42) / input4
```

Code Snippet 3.1: Example of the *Simple Evaluator* expression

3.3 Code Testing

Code testing is important during the development and maintenance of an application. It ensures that the bugs in the new code are found during the development and not released to the production. It also ensures that new changes do not break the old code covered with the tests.

We used unit testing to cover the code with the tests. Since the application is self-contained in its current state, we didn't find it necessary to create complex integration tests. This might change in the future if, for example, a new formula evaluator calling foreign services would be added to the application.

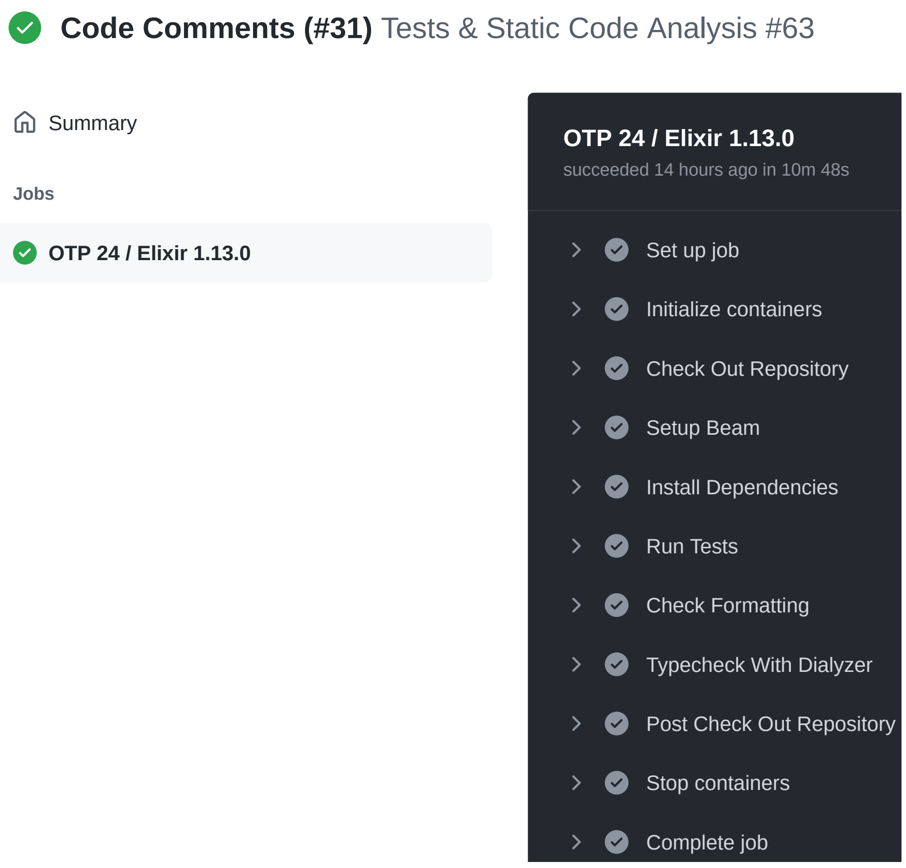
We mostly followed Test Driven Development (TDD) [55] principles during the development, which led us to high test coverage of 87.33%. This number was obtained by calling `mix test --cover` command which runs the Elixir tests and also calculates how much percent of the code is covered by tests.

To add to the advantages of the testing, we also believe that good tests can serve as a form of documentation and help new contributors/developers to understand the codebase.

3.3.1 CI & Automatic Testing

"CI is the practice of automating the integration of code changes from multiple contributors into a single software project. It's a primary DevOps best practice, allowing developers to frequently merge code changes into a central repository where builds and tests then run. Automated tools are used to assert the new code's correctness before integration." [5]

CI is enabled for our project through the *GitHub Actions* [56] which is an automation environment you can use in your GitHub repository. We use Actions to automatically build & test the code when a new *Pull Request* is being made. Merging code to the *master* branch is protected and allowed only once all checks pass. We use it not only to run the tests but also to check the code formatting and run static code analysis that might find possible vulnerabilities and bugs in the code. This way, we make sure that the code in the master branch is always properly formatted and tested. An example of successfully passed CI pipeline can be seen in Figure 3.1.


Code Comments (#31) Tests & Static Code Analysis #63

Summary

Jobs

OTP 24 / Elixir 1.13.0

OTP 24 / Elixir 1.13.0
succeeded 14 hours ago in 10m 48s

- Set up job
- Initialize containers
- Check Out Repository
- Setup Beam
- Install Dependencies
- Run Tests
- Check Formatting
- Typecheck With Dialyzer
- Post Check Out Repository
- Stop containers
- Complete job

Figure 3.1: An example of successfully passed GitHub Actions pipeline in the ACE repository [56]

3.4 Documentation

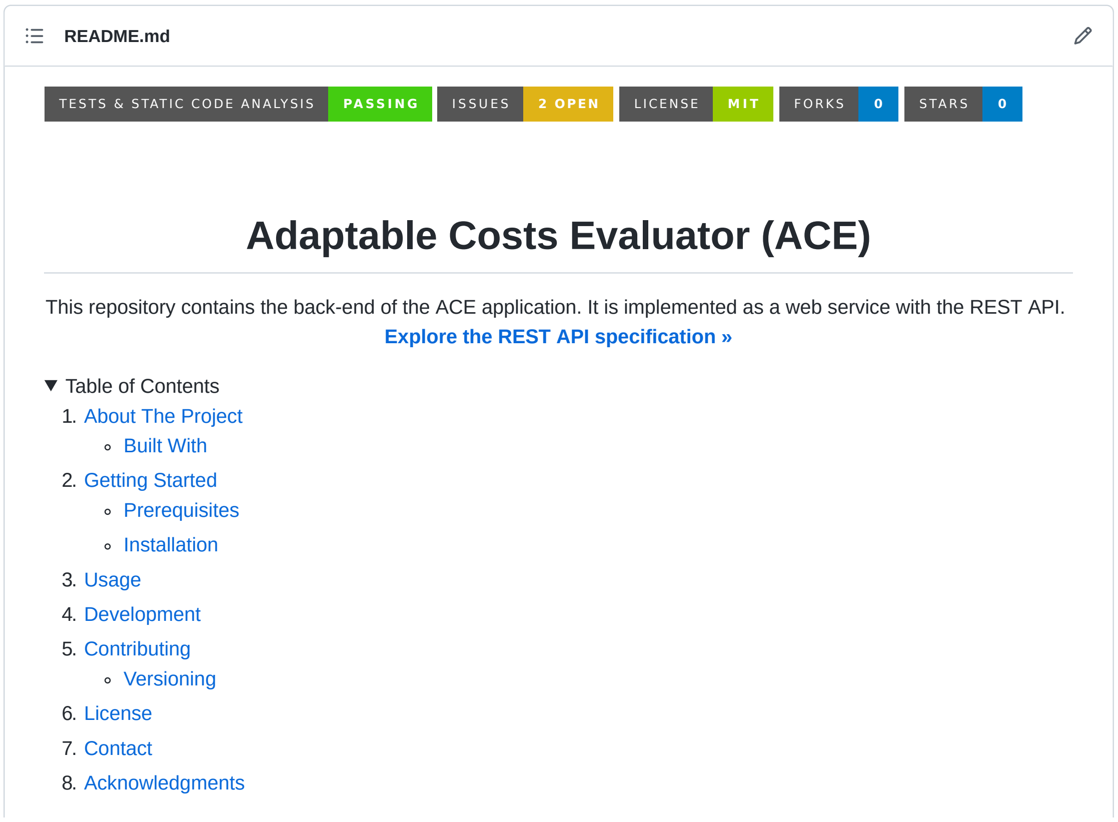
Documentation is an important part of most software projects. It helps to understand how the given application works. In our context, it also gives instructions on installing, developing, and using the application. We included different types of documentation for the ACE application.

3.4.1 Code Comments

Comments in the code are the most basic documentation our application includes. We commented on the most important parts of the business logic. The comments are intended to help new developers better understand the code. They also serve as documentation for the public API parts of our code. The useful thing about the comments in Elixir is that the language treats the comments as first-class citizens. It means that the language offers various functions to access and also generate documentation for the project.

3.4.2 README File

README file is another handy way of writing documentation. We include this file in the root of the GitHub repository [3]. The file is written using Markdown [57] syntax and automatically rendered in the GitHub UI. The header of the file can be seen in Figure 3.2. The file contains a description of the project, how to install and develop it, and other information like licensing, contact, etc. It also contains technical information about the project like the state of the CI pipelines, number of opened issues, etc.



README.md

TESTS & STATIC CODE ANALYSIS PASSING | ISSUES 2 OPEN | LICENSE MIT | FORKS 0 | STARS 0

Adaptable Costs Evaluator (ACE)

This repository contains the back-end of the ACE application. It is implemented as a web service with the REST API.
Explore the REST API specification »

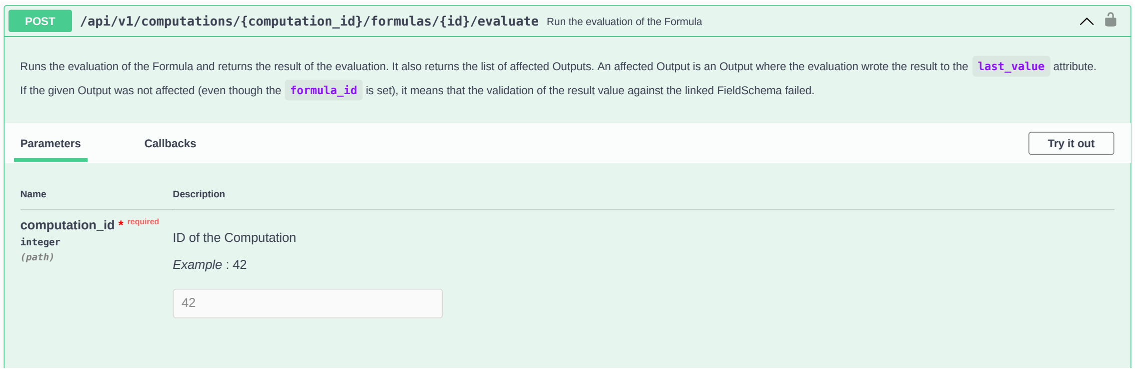
▼ Table of Contents
1. About The Project
   - Built With
2. Getting Started
   - Prerequisites
   - Installation
3. Usage
4. Development
5. Contributing
   - Versioning
6. License
7. Contact
8. Acknowledgments

Figure 3.2: ACE `README.md` file in the root of the GitHub repository [3]

3.4.3 REST API Specification

REST API specification is an important type of documentation for the developers seeking to integrate the back-end into the other systems or applications. It can also help the contributors to better understand how the application works and how different parts (resources) of the application depend on each other. We use *OpenAPI* Specification [58] for our application. It is a machine-readable specification for describing, producing, and visualizing RESTful web services.

We use OpenAPI through the *Open API Spex* [59] Elixir package that enables us to automatically generate a lot of parts of the specification from the code. This package is able to generate a JSON file containing the whole specification. To visualize the specification, we use *Swagger UI* [60] tool. An example of how the visualized specification looks like can be seen in Figure 3.3.


POST `/api/v1/computations/{computation_id}/formulas/{id}/evaluate` Run the evaluation of the Formula

Runs the evaluation of the Formula and returns the result of the evaluation. It also returns the list of affected Outputs. An affected Output is an Output where the evaluation wrote the result to the `last_value` attribute. If the given Output was not affected (even though the `formula_id` is set), it means that the validation of the result value against the linked FieldSchema failed.

Name	Description
computation_id * required `integer` *(path)*	ID of the Computation *Example* : 42 42

Figure 3.3: ACE API specification visualized using Swagger UI [60]

The visualized OpenAPI specification using Swagger UI is also accessible online and hosted using GitHub Pages [61]. It is automatically built every time the specification file, which is located in the repository, is updated and pushed/merged to the GitHub master branch.

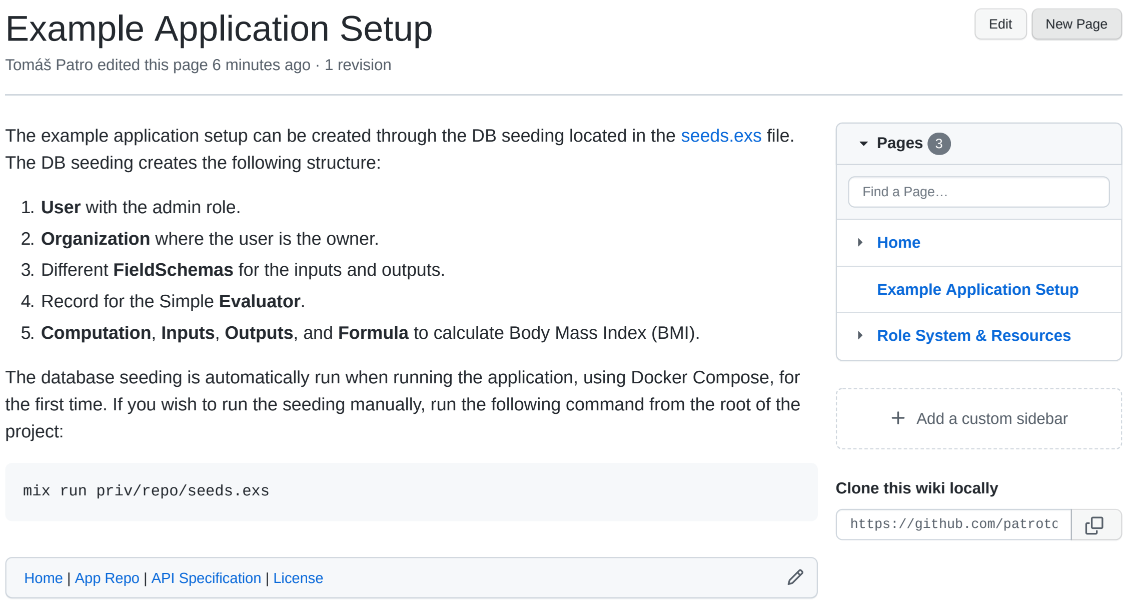
3.4.4 Wiki Pages

Another practical type of documentation is Wiki pages. We use GitHub Wiki [53] to host our Wiki pages. The pages contain information intended for the developers who wish to contribute to the application code or integrate it with foreign systems. We can find pages describing the role system, authorization, or example application setup. A preview of one of the pages can be seen in Figure 3.4.

3.5 Implemented Functional Requirements

To summarize the conducted implementation, we will look at the FRs we implemented in the back-end application. We defined the FRs in Section 1.6. Since we didn't implement a front-end part of the application, we will take a look only at groups no. 1 and 2. We will describe the implemented FRs on a similar table like the one where we defined them. For clarity, we include the same short description next to each FR. On top of that, we add an explanation of how we implemented the given FR and what it enables the client/user to do or why the implementation was omitted.

The first Table 3.1 describes implementation of the FRs from Table 1.8. The second Table 3.2 then describes implementation of the FRs from Table 1.9.



Example Application Setup

Tomáš Patro edited this page 6 minutes ago · 1 revision

The example application setup can be created through the DB seeding located in the seeds.exs file. The DB seeding creates the following structure:

1. **User** with the admin role.
2. **Organization** where the user is the owner.
3. Different **FieldSchemas** for the inputs and outputs.
4. Record for the Simple **Evaluator**.
5. **Computation**, **Inputs**, **Outputs**, and **Formula** to calculate Body Mass Index (BMI).

The database seeding is automatically run when running the application, using Docker Compose, for the first time. If you wish to run the seeding manually, run the following command from the root of the project:

```
mix run priv/repo/seeds.exs
```

Home | App Repo | API Specification | License

Figure 3.4: ACE Wiki Page [53]

Group No. 1		
FR No.	**FR Description**	**Implementation**
FR 1.1	The user is able to adjust how the calculations are performed. This would mean implementing a functionality where users could create their own input fields and corresponding output fields and mapping between them (calculation definition).	The user is able to create their own computation, which is a form of a spreadsheet that can contain user-defined input and output fields. The fields can be of various data types defined by the linked JSON schemas describing them. This way, the client can define simple data types like numbers or strings and more complex ones like arrays or objects. The mapping between the input and output fields is enabled through the formulas that can reference input fields, evaluate the given formula definition, and write the result to the output fields.

FR 1.2	The user is able to create/import their own set of rules and conditions for the calculations. This FR is closely related to FR 1.1. This FR describes the need to have a formal language to describe the formulas, conditions, and I/O mappings to produce the computation results.	The computation process is enabled through the formulas that contain definitions. These definitions are written in a formula language that is readable by the linked evaluator. The application can be extended with the new evaluator implementations that can use different strategies for the evaluation, define more complex data types and even call the external services. We also implemented a simple evaluator that is able to use basic mathematical and logical operators and some of the common mathematical functions.

Table 3.1: Implemented FRs from the group no. 1

Group No. 2		
FR No.	**FR Description**	**Implementation**
FR 2.1	User is able to create their own account and save the preferences, calculation results, etc.	Each user can create and have their own account that can be used to create new computations, evaluate them, and save the results. The user can sign in to their account using the email and password.
FR 2.2	The user is able to create an organization within the application and invite other users to it.	The user can create multiple organizations and can be part of multiple organizations. They can invite other users to the organization and share the computations and other resources with each other.

FR 2.3	There is a role system with the admin, owner, and regular roles (example names).	We implemented a role system that enables authorization within the organization. One specific role is a user that can manipulate their own resources. Then there is a role system within each organization that defines permissions for the users in the organization. The current implementation comes with regular, maintainer, and owner roles. There is also a special admin role that is meant for the application administrators and that enables the user to manipulate all resources.
FR 2.4	The user is able to export the calculation results in the given format (CSV, XML, etc.).	This is the one FR we didn't include in the implementation. It would require further research and feedback from the users to identify in which formats they would like to export the computations (calculations) and how should the exported data look like. This FR is, however, a good candidate for future development.
FR 2.5	The user is able to interact programmatically with the application using the web API.	The whole back-end is implemented as a web service with the REST API. The ability to programmatically interact with it comes from the nature of the implementation.

Table 3.2: Implemented FRs from the group no. 2

To summarize, we were able to implement most of the back-end related FRs except the ability to export the computations in the custom format. This feature can be considered as a candidate for future development. We didn't implement FRs from Table 1.10 since we didn't implement the front-end of the application as described in previous sections.

3.6 Future Development

In this chapter, we described and dag into the existing implementation of the ACE application. Many of the initially set FRs in Section 1.6 were implemented, and the back-end as a web service is ready for usage. However, there are still new features that can be developed and existing features that can be enhanced. We prepared a list of these features with a brief explanation:

1. **Front-end** – An important thing to do in the future. We should either develop a new front-end or integrate the back-end to an existing front-end.
2. **Refactoring of the `inputs` and `outputs` tables** – The current implementation divides input and output representations into two tables. However, these two resources have a lot of similarities, and they could be united under one table (e.g., `fields` table). This would enable us to treat the two resources interchangeably and let us write formulas more flexibly, referencing the result of the evaluation of one formula (some output) in another formula. We would have to solve the problem with the order of evaluations, so a referenced output is already evaluated before starting the evaluation of the formula referencing it (recursive evaluation).
3. **Enhancement of roles system** – The current role system can be enhanced by new roles that would enable more flexible authorization. On the other hand, this system can also be replaced by a classic read/write permissions system. Users would be able to assign permissions for a particular resource in the context of other users or in the context of organizations (e.g., a resource within the organization with the read permission would enable all users in the organization to read it).
4. **Import/export computations from/to the specific format(s)** – The feature which was also set as the FR 2.4 in Section 1.6. It would require designing and implementing special connectors for each format to import/export the computations. We should design the connectors using a unified interface, so implementation of a new connector would, for example, require creating a new module implementing `import` and `export` functions.
5. **API keys** – These keys would be used by the foreign systems integrating our application. They would be used for the authentication instead of the JWTs issued for the users that have a limited lifetime. This would also require implementing a mechanism for controlling the scope of the API keys.
6. **Using OpenAPI specification to validate request bodies** – The API specification contains JSON schemas describing the request bodies for different resources. These schemas can be used to enhance the validation of the request bodies.

3.7 Implementation Benefits Summary

In this section, we would like to take a look at some of the development and implementation decisions that might be beneficial for the future development and usage of the application.

The initial goal that was set was to develop an application that would be able to define computations in a flexible manner. The initiative arose from the fact that the existing

SCE application [2] is constrained with the computations being hard-coded. We designed the new ACE application [3] to solve this problem on one side but on the other side to also be easily extendable and maintainable.

The first important thing enabling this was the decision to use the FP technologies. We discussed some of the benefits of FP in Section 2.3. In the context of ACE, the FP allowed us to write a clean code consisting of the composition of short functions. The benefits of this approach enabled us to cover most of the code with the unit tests since testing the functions proved to be straightforward and effective. Short functions with the proper names and clear function signatures are also self-documenting and enable new contributors to better understand the code. In addition, the Elixir programming language [45] alongside the Phoenix Framework [46] proved to be the right decision for functional web development. We also took advantage of the PostgreSQL database [48] and mainly of its `JSONB` data type that fit our needs for storing JSON schemas [43].

Secondly, we designed the application to be extendable and flexible. The design enables the clients to define their own computations with the input and output fields. However, the main benefit is the ability to define any data types for these fields as long they can be described by a JSON schema. We combined this feature with an option to develop an arbitrary number of evaluators that can be used to evaluate formula definitions. The combination of these two aspects is the key attribute that makes the application versatile and adaptable.

The last thing we want to highlight is the supportive technologies around the application. The most important part here is the documentation. We included different types of documentations – Swagger UI [60] for the API specification, installation manual in the form of README, Wiki pages as a documentation for the developers, and we also documented the main functionalities in the code. In addition, the code is covered with the unit tests that also serve as a form of documentation for the new contributors. To add to this, we set up automatic CI pipelines using GitHub Actions [56] to build & test the code, check the formatting, and conduct a static code analysis to look for the vulnerabilities and potential bugs.

The application back-end is ready to be used and integrated. However, there are still things that should be done as we described in Section 3.6. The future development should be eased by the fact that the process is already defined and supported by the existing documentation, automatic CI pipelines, and the code that is mostly covered by the tests.

Conclusion

In the thesis, we presented different parts of the Software Development Life Cycle (SDLC) [62] we went through when developing the Adaptable Costs Evaluator (ACE) application. The goal was to describe what steps we took in order to design and develop a new web application for the adaptable computations using functional programming technologies.

We started by describing the analysis that we conducted prior to designing and implementing the new ACE application. In this analysis, we presented the current Storage Costs Evaluator (SCE) application that we used as a base for the new and improved application. We went through its architecture to understand how the current solution works and how we could improve it. We gathered and analyzed the feedback from the current users. Afterward, we looked at the other existing solutions, methods, and technologies in this field. Based on this analysis, we created a set of FRs for the new application.

We continued by creating and describing a new application design for the ACE application. We went through the high-level design of the architecture and described the desired back-end functionalities and database design. We also looked at the use of functional programming in the context of the new application.

In the last part, we dug into the implementation of the ACE application. We described the used technologies and some of the implementation details. We also looked at the supporting aspects of the development, like code testing and documentation.

In the end, we summarized the implementation by describing the implemented FRs and by talking about the benefits of the current implementation. We also outlined the future development by looking at the features that would be beneficial to include in the next development phase.

Bibliography

1. TYSON, Matthew. What is functional programming? A practical guide. *InfoWorld* [online]. 2021 [visited on 2021-12-15]. Available from: https://www.infoworld.com/article/3613715/what-is-functional-programming-a-practical-guide.html.
2. SUCHÁNEK, Marek. *Storage Costs Evaluator* [online]. Data Stewardship Wizard, © 2019 [visited on 2021-12-15]. Available from: https://storage-costs-evaluator.ds-wizard.org/.
3. PATRO, Tomáš. *Adaptable Costs Evaluator (ACE)* [online]. GitHub, Inc., © 2021 [visited on 2021-12-15]. Available from: https://github.com/patrotom/adaptable-costs-evaluator.
4. FIELDING, Roy Thomas. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.
5. REHKOPF, Max. What is Continuous Integration? *Atlassian* [online]. © 2021 [visited on 2021-12-15]. Available from: https://www.atlassian.com/continuous-delivery/continuous-integration.
6. HOOFT, Rob W.W. *Calculation model for costs of (digital) storage* [online]. Zenodo, 2020. 2020-09-16 [visited on 2021-12-15]. Available from DOI: 10.5281/zenodo.4033087.
7. SUCHÁNEK, Marek. *Storage Costs Evaluator* [online]. GitHub, Inc., © 2019 [visited on 2021-12-15]. Available from: https://github.com/ds-wizard/storage-costs-evaluator.
8. EERNISSE, Matthew. *Embedded JavaScript* [online]. © 2012 [visited on 2021-12-15]. Available from: https://ejs.co/.
9. *Bootstrap* [online]. Twitter, Inc., © 2011–2021 [visited on 2021-12-15]. Available from: https://getbootstrap.com/.

10. JQUERY CONTRIBUTORS. *jQuery* [online]. OpenJS Foundation, © 2021 [visited on 2021-12-15]. Available from: `https://jquery.com/`.
11. *Font Awesome* [online]. Fonticons, Inc, © 2021 [visited on 2021-12-15]. Available from: `https://fontawesome.com/`.
12. *Webpack* [online]. JS Foundation and other contributors, © 2021 [visited on 2021-12-15]. Available from: `https://fontawesome.com/`.
13. FARMER, Andrew. *Scotty* [online]. GitHub, Inc., © 2012–2017 [visited on 2021-12-15]. Available from: `https://github.com/scotty-web/scotty`.
14. STACK CONTRIBUTORS. *The Haskell Tool Stack* [online]. © 2015–2021 [visited on 2021-12-15]. Available from: `https://docs.haskellstack.org/en/stable/README/`.
15. HANSSON, David Heinemeier. *Ruby on Rails* [online]. © 2005–2021 [visited on 2021-12-15]. Available from: `https://rubyonrails.org/`.
16. *Google Forms* [online]. Google LLC, © 2021 [visited on 2021-12-15]. Available from: `https://www.google.com/forms/about/`.
17. SHNEIDERMAN, Ben; PLAISANT, Catherine; COHEN, Maxine S; JACOBS, Steven; ELMQVIST, Niklas; DIAKOPOULOS, Nicholas. *Designing the user interface: strategies for effective human-computer interaction.* Pearson, 2016.
18. *Engineering UK Useful Statistics* [online]. Women's Engineering Society, © 2021 [visited on 2021-12-15]. Available from: `https://www.wes.org.uk/content/wesstatistics`.
19. *Microsoft Excel* [online]. Microsoft Corporation, © 2021 [visited on 2021-12-15]. Available from: `https://www.microsoft.com/en-us/microsoft-365/excel`.
20. *LibreOffice Calc* [online]. The Document Foundation, © 2021 [visited on 2021-12-15]. Available from: `https://www.libreoffice.org/discover/calc/`.
21. *Google Sheets* [online]. Google LLC, © 2021 [visited on 2021-12-15]. Available from: `https://www.google.com/sheets/about/`.
22. LIBREOFFICE DOCUMENTATION TEAM. *Calc Guide* [online]. © 2021 [visited on 2021-12-15]. Available from: `https://books.libreoffice.org/en/CG71/CG71.html`.
23. CHAI, Wesley; CASEY, Kathleen. Software as a Service (SaaS). *TechTarget* [online]. © 2010–2021 [visited on 2021-12-15]. Available from: `https://searchcloudcomputing.techtarget.com/definition/Software-as-a-Service`.
24. *Sheets API* [online]. Google LLC, © 2021 [visited on 2021-12-15]. Available from: `https://developers.google.com/sheets/api`.
25. *uCalc* [online]. WebTechRazrabotka LLC, © 2021 [visited on 2021-12-15]. Available from: `https://ucalc.pro/en`.

26. STEELE, Howard. Building an Online Calculator. *SuperbWebsiteBuilders.com* [online]. © 2021 [visited on 2021-12-15]. Available from: https://superbwebsitebuilders.com/how-to-create-a-calculator-for-your-website/.
27. *Calconic App* [online]. MB Lumius, © 2017–2021 [visited on 2021-12-15]. Available from: https://www.calconic.com/.
28. *Calculoid* [online]. Easy Software Ltd., © 2021 [visited on 2021-12-15]. Available from: https://www.calculoid.com/.
29. *ConvertCalculator* [online]. ConvertCalculator, © 2021 [visited on 2021-12-15]. Available from: https://www.convertcalculator.co/.
30. *JSCalc* [online]. OLAPHASE SDN. BHD., © 2021 [visited on 2021-12-15]. Available from: https://jscalc.io/.
31. JETBRAINS S.R.O. *Domain-Specific Languages* [online]. © 2000–2021 [visited on 2021-12-15]. Available from: https://www.jetbrains.com/mps/concepts/domain-specific-languages/.
32. PCMAG. *scripting language* [online]. © 1996-2021 [visited on 2021-12-15]. Available from: https://www.pcmag.com/encyclopedia/term/scripting-language.
33. *Rhino: JavaScript in Java* [online]. GitHub, Inc., © 2021 [visited on 2021-12-15]. Available from: https://github.com/mozilla/rhino.
34. *Lua* [online]. Lua.org, PUC-Rio, © 1994–2021 [visited on 2021-12-15]. Available from: http://www.lua.org/home.html.
35. BEHNEL, Stefan. *Lupa* [online]. GitHub, Inc., © 2010–2017 [visited on 2021-12-15]. Available from: https://github.com/scoder/lupa.
36. POLAK, Gracjan; AĞACAN, Ömer Sinan; KREWINKEL, Albert. *HsLua* [online]. GitHub, Inc., © 2007–2021 [visited on 2021-12-15]. Available from: https://github.com/hslua/hslua.github.io.
37. Multi-tenant SaaS database tenancy patterns. *Microsoft Corporation* [online]. © 2021 [visited on 2021-12-15]. Available from: https://docs.microsoft.com/en-us/azure/azure-sql/database/saas-tenancy-app-design-patterns.
38. What is a web service? *IBM Corporation* [online]. © 2014–2021 [visited on 2021-12-15]. Available from: https://www.ibm.com/docs/en/cics-ts/5.2?topic=services-what-is-web-service.
39. What is Cloud Native? *Microsoft Corporation* [online]. © 2021 [visited on 2021-12-15]. Available from: https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/definition.
40. *React* [online]. Facebook Inc., © 2021 [visited on 2021-12-15]. Available from: https://reactjs.org/.
41. YOU, Evan. *Vue.js* [online]. © 2014–2021 [visited on 2021-12-15]. Available from: https://vuejs.org/.

42. CZAPLICKI, Evan. *Elm* [online]. © 2012–2021 [visited on 2021-12-15]. Available from: `https://elm-lang.org/`.
43. JSON SCHEMA TEAM. *JSON Schema* [online]. © 2021 [visited on 2021-12-15]. Available from: `https://json-schema.org/`.
44. WAGNER, Lane. Top 8 Benefits of Functional Programming. *Qvault* [online]. 2021 [visited on 2021-12-15]. Available from: `https://qvault.io/clean-code/benefits-of-functional-programming/`.
45. *Elixir* [online]. The Elixir Team, © 2012–2021 [visited on 2021-12-15]. Available from: `https://elixir-lang.org/`.
46. MCCORD, Chris. *Phoenix Framework* [online]. © 2020 [visited on 2021-12-15]. Available from: `https://www.phoenixframework.org/`.
47. DOCKYARD, INC. *The preeminent Elixir consultancy* [online]. © 2017 [visited on 2021-12-15]. Available from: `https://dockyard.com/capabilities/elixir-consulting`.
48. *PostgreSQL* [online]. The PostgreSQL Global Development Group, © 1996–2021 [visited on 2021-12-15]. Available from: `https://www.postgresql.org/`.
49. *Docker* [online]. Docker, Inc., © 2020 [visited on 2021-12-15]. Available from: `https://www.docker.com/`.
50. *Docker Compose* [online]. Docker, Inc., © 2020 [visited on 2021-12-15]. Available from: `https://docs.docker.com/compose/`.
51. JONES, M.; BRADLEY, J.; SAKIMURA, N. *JSON Web Token (JWT)* [Internet Requests for Comments]. RFC Editor, 2015 [visited on 2021-12-15]. ISSN 2070-1721. Available from: `http://www.rfc-editor.org/rfc/rfc7519.txt`. RFC. RFC Editor.
52. JONES, M.; HARDT, D. *The OAuth 2.0 Authorization Framework: Bearer Token Usage* [Internet Requests for Comments]. RFC Editor, 2012 [visited on 2021-12-15]. ISSN 2070-1721. Available from: `http://www.rfc-editor.org/rfc/rfc6750.txt`. RFC. RFC Editor.
53. PATRO, Tomáš. *Adaptable Costs Evaluator Wiki* [online]. © 2021 [visited on 2021-12-23]. Available from: `https://github.com/patrotom/adaptable-costs-evaluator/wiki`.
54. SCHMALE, Moritz. *Abacus* [online]. GitHub, Inc., © 2016 [visited on 2021-12-15]. Available from: `https://github.com/narrowtux/abacus`.
55. HAMILTON, Thomas. Unit Testing Tutorial: What is, Types, Tools & Test EXAMPLE. *Guru99* [online]. 2021 [visited on 2021-12-15]. Available from: `https://www.guru99.com/unit-testing-guide.html`.
56. *GitHub Actions* [online]. GitHub, Inc., © 2021 [visited on 2021-12-15]. Available from: `https://github.com/features/actions`.

57. GRUBER, John. *Markdown* [online]. The Daring Fireball Company LLC, © 2002–2021 [visited on 2021-12-15]. Available from: `https://daringfireball.net/projects/markdown/`.
58. *OpenAPI Specification* [online]. The Linux Foundation®, © 2021 [visited on 2021-12-15]. Available from: `https://www.openapis.org/`.
59. BUHOT, Michael. *Open API Spex* [online]. GitHub, Inc., © 2017 [visited on 2021-12-15]. Available from: `https://github.com/open-api-spex/open_api_spex`.
60. *Swagger UI* [online]. SmartBear Software, © 2021 [visited on 2021-12-15]. Available from: `https://swagger.io/tools/swagger-ui/`.
61. *Adaptable Costs Evaluator Swagger UI* [online]. GitHub, Inc., © 2021 [visited on 2021-12-15]. Available from: `https://patrotom.github.io/adaptable-costs-evaluator/`.
62. Software Development Life Cycle (SDLC). *Synopsys, Inc.* [online]. © 2021 [visited on 2021-12-15]. Available from: `https://www.synopsys.com/glossary/what-is-sdlc.html`.

Appendix **A**

Glossary

ACE Adaptable Costs Evaluator.

API Application Programming Interface.

CI Continuous Integration.

CSS Cascading Style Sheets.

CSV Comma-separated values.

DSL Domain Specific Language.

EJS Embedded JavaScript.

FP Functional Programming.

FR Functional Requirement.

HTML HyperText Markup Language.

HTTP Hypertext Transfer Protocol.

HTTPS Hypertext Transfer Protocol Secure.

I/O Input/Output.

ID Identifier.

IDE Integrated Development Environment.

JS JavaScript.

JSON	JavaScript Object Notation.
JWT	JSON Web Token.
MVC	Model-View-Controller.
RDBMS	Relational Database Management System.
REST	Representational State Transfer.
SaaS	Software as a service.
Saas	Syntactically Awesome Style Sheets.
SCE	Storage Costs Evaluator.
SDLC	Software Development Life Cycle.
SoC	Separation of Concerns.
SPA	Single Page Application.
TB	terabyte.
TDD	Test Driven Development.
UI	User Interface.
UK	United Kingdom.
UX	User Experience.
VM	Virtual Machine.
XML	Extensible Markup Language.

Appendix B

Contents of Enclosed CD

`README.md`	the file with CD contents description
`adaptable-costs-evaluator.zip`	ZIP archive with the Adaptable Costs Evaluator application source code
`DP_Patro_Tomas_2022.pdf`	the thesis text in PDF format
`DP_Patro_Tomas_2022.zip`	ZIP archive with the thesis LaTeX source files