Politecnico di Torino

Computer Engineering

A.Y. 2023/2024

Graduation session of October 2024

From Images to Code: Leveraging Computer Vision and Large Language Models for Front-End Automation

Supervisors:
Luigi De Russis
Fabio Raimondi

Candidate:
Domenico Manuardi

Abstract

Recent advances in Artificial Intelligence have catalyzed innovation across diverse domains, including software engineering. Traditionally, applications of modern Large Language Models (LLMs) have been centered on creating code-writing assistants that generate code from a given context or textual description of the desired outcome. This thesis presents a novel approach, focusing on the development of a coding assistant that harnesses the multi-modal capabilities of state-of-the-art LLMs to facilitate front-end development from a graphical representation, in the form of an image. The proposed system employs sophisticated computer vision techniques to detect and analyze graphical elements, subsequently utilizing a multi-modal LLM to translate these elements into an intermediate, language-agnostic format, which is then converted into the desired target language. This research also encompasses the practical implementation of the solution, realized as a web application designed for internal usage at Blue Reply. By bridging the gap between design and code through advanced AI technologies, this project aims to significantly streamline the front-end development process, offering a seamless transition from visual design to functional code.

Table of Contents

List of Figures V

1 Introduction 1
1.1 Context and Motivation . . . 2
1.2 Overall Thesis Structure . . . 4

2 General Process and System Structure 6
2.1 State of the Art . . . 6
2.2 Overview of the Proposed Solution . . . 8
2.3 Application Structure and Workflow . . . 10
2.4 Implementation Strategy . . . 13

3 Computer Vision and Optical Character Recognition for the Initial Image Analysis 14
3.1 Adopted Computer Vision Techniques . . . 14
3.2 Edge Detection for Relevant UI Elements . . . 15
3.2.1 The Problem of Edge Detection . . . 15
3.2.2 Edge Detection Methods . . . 16
3.2.3 The Canny Edge Detection Algorithm . . . 19
3.2.4 Pre-processing Steps for Improved Results . . . 23
3.3 Optical Character Recognition for Textual Content Detection . . . 24
3.3.1 The Problem of Optical Character Recognition . . . 24
3.3.2 Modern Techniques for OCR . . . 25
3.3.3 The Tesseract OCR Engine . . . 27
3.3.4 Pre-processing Steps for Improved Results . . . 29
3.4 Post-processing Techniques . . . 30
3.4.1 Detected Edges Post-processing . . . 30
3.4.2 OCR Results Post-processing . . . 35
3.4.3 Edge Detection and OCR Results Integration and Post-processing . . . 36
3.5 Implementation Tools . . . 38

3.5.1 OpenCV: A Powerful Open-Source Computer Vision Library 38
3.5.2 PyTesseract: Integrating Tesseract into the Python Ecosystem 39
3.6 Results of the Element Detection Process 40

4 Analysis of UI Elements with Multi-Modal Large Language Models 42
4.1 Overview of Large Language Models (LLMs) 43
4.1.1 The Transformer Architecture 44
4.1.2 The Concept of Multi-Modality in LLMs 46
4.2 Analysis of UI Elements 48
4.2.1 Using GPT-4o for Standardized UI Element Descriptions . . 48
4.3 Standardizing the Description of UI Elements 50
4.3.1 Prompt Engineering Techniques for Higher Quality Generation 51
4.3.2 Definition of the Scope 52
4.3.3 Adopted Description Format 53
4.4 Overall Analysis Process 54
4.4.1 Organizing the Inference Process for Fast Image Analysis . . 54
4.4.2 Optimizing Costs with Design Decisions and Trade-offs . . . 56
4.5 Results of the Element Analysis Process 58

5 Layout Generation Process 59
5.1 Web Layout Utilities as a Starting Point 59
5.1.1 CSS Flexbox Module 60
5.1.2 CSS Grid Layout Module 60
5.1.3 Bootstrap Grid Layout Utilities 61
5.2 Using a Grid Structure for Page Layout 61
5.2.1 Layout Generation Algorithm 62
5.2.2 Post-processing for Human-like Layout Organization 63
5.3 The Final Result: General Page Description 64

6 Modules Integration and Overall Application Architecture 66
6.1 Modules Functionality Integration 66
6.2 Applying Recursive Processing to Containers 67
6.2.1 Recursive Analysis of Container Elements 67
6.2.2 Pitfalls of the Recursive Approach and Adopted Solutions . 68
6.3 Translation Modules 69
6.3.1 The Bootstrap 5 Translation Module 70
6.3.2 The Angular/PrimeNG Translation Module 71
6.4 Application Architecture 71
6.4.1 Technology Stack 72
6.4.2 Data Persistence Layer 73
6.4.3 Backend Layer 75

6.4.4 Frontend Layer 76
6.5 REST API Analysis 77
6.5.1 Core Functionality 77
6.5.2 Secondary Functionalities 78
6.6 Adopted Design Principles and Guidelines 80

7 Evaluation and Results 85
7.1 Objectives and Requirements 85
7.2 Early Application Testing with Real Use-cases 86
7.3 Overall Results Assessment and Observations 90

8 Conclusions 92
8.1 Advantages and Limitations of the Proposed Solution 92
8.2 Possible Future Developments 93
8.2.1 Improving the Element Analysis Accuracy 93
8.2.2 Persisting Information to Reduce Costs 95
8.2.3 Organizing Image Conversions into Projects 98

Bibliography 100

List of Figures

2.1 Flow chart of the general adopted process 9
2.2 Flow chart representation of the application modules and the related workflow 11

3.1 Discrete approximation to LoG function with Gaussian $\sigma = 1.4$, from [23]. 19
3.2 Example usage of the Canny Edge Detection algorithm on a user interface mock-up. 20
3.3 Example of non-maximum suppression, from [28]. 22
3.4 Overview of image pre-processing for Edge Detection. 24
3.5 Architecture of the Tesseract OCR Engine, from [38]. 27
3.6 Overview of the input image OCR process employed in the application, including the related pre-processing steps. 31
3.7 Example of contour/border detection within an image using the Suzuki-Abe Algorithm, from [42]. 33
3.8 Example of the obtained output from the Element Detection process. 41

4.1 The encoder-decoder structure of the Transformer architecture, from [49]. 45
4.2 The Element Analysis process. 55

5.1 Example of a layout produced by the Layout Generation Module. The detected UI elements are organized into nested rows and columns. 62

6.1 Example of the conversion of a screenshot into HTML and CSS using the Bootstrap 5 Translation Module. 70
6.2 Overview of the application architecture. 73
6.3 Overview of the architecture of SQLite, from [68]. 74
6.4 Architecture of SQLAlchemy, from [70]. 76
6.5 Initial interface shown in the *Generate* section. 82
6.6 Example of the visualized preview for the generated code. 82
6.7 Example of the generated layout visual representation. 83

6.8 Interface shown in the *History* section. 84
6.9 Interface shown as overview of a single *History* entry. 84

7.1 “Process Edit” test mockup input (top left) and the rendered Bootstrap-translated output (bottom right). 87
7.2 “Process Search” test mockup input (top left) and the rendered Angular/PrimeNG-translated output (bottom right). 88
7.3 “Vehicle Tracking” test screenshot input (top left) and the rendered Angular/PrimeNG-translated output (bottom right). 89
7.4 “Process Details” test mockup input (top left) and the rendered Bootstrap-translated output (bottom right). 90

8.1 Examples of consecutive image clusters with the same Average Hash. 97
8.2 Flow chart showing an example of the revised Element Analysis process. 98

Chapter 1

Introduction

This thesis explores an innovative application of multi-modal Large Language Models (LLMs) in the realm of front-end development. Specifically, it proposes a system that interprets graphical representations of user interfaces and translates them into corresponding code. This approach leverages advanced computer vision techniques to identify and analyze graphical elements within an image, which are then processed by a state-of-the-art LLM to produce an intermediate, language-agnostic representation. This representation is subsequently converted into the desired target language, facilitating the seamless transition from design to implementation.

The practical implementation of this solution has been realized in the form of a web application developed for internal use at Blue Reply. This application aims to streamline the front-end development process, significantly reducing the gap between visual design and functional code creation. By automating the translation of graphical designs into code, the proposed system enhances efficiency and accuracy in front-end development, enabling developers to focus more on creative and logically complex tasks.

To provide a comprehensive understanding of the research and its contributions, this introduction is divided into two subsections. The first subsection, *Context and Motivation*, delves into the background of the study, outlining the current challenges in front-end development and the potential benefits of integrating multi-modal AI technologies into the workflow. It highlights the motivations behind this research, emphasizing the necessity for tools that can bridge the gap between design and implementation more effectively.

The second subsection, *Overall Thesis Structure*, offers a roadmap of the thesis, summarizing the contents and structure of the subsequent chapters. This overview serves as a guide for readers, ensuring a clear and coherent progression through the various aspects of the research, from the theoretical foundations to the practical implementation and evaluation of the proposed system.

1.1 Context and Motivation

Since the recent advancements in Artificial Intelligence, developers have sought innovative methods to enhance their work and augment their productivity through its use. This pursuit has culminated in the development of numerous AI-powered programming tools, some of which have demonstrated significant efficacy. A notable example of such successes is GitHub Copilot [1], a widely-used Integrated Development Environment (IDE) plugin that enables programmers to receive real-time code completion suggestions. These suggestions are based on the extensive knowledge embedded within a Large Language Model (LLM) and contextual information derived from the existing code base. Moreover, developers have increasingly utilized conversational platforms to engage with AI models for assistance in programming tasks. In this context, prominent AI chat platforms such as ChatGPT [2] have proven to be invaluable resources, enhancing productivity and offering valuable suggestions for problem-solving.

More specifically, the 2024 Stack Overflow Developer Survey has underscored the perceived benefits of AI tools as reported by their users [3]. An overwhelming 82.7% of professional developers who participated in the survey acknowledged that increased productivity is one of the most significant advantages derived from these new tools. Additionally, 58.5% of respondents indicated that their efficiency had improved as a result of utilizing AI technologies. While modern AI tools are designed to assist programmers in various capacities, code generation remains the most prevalent application of such technology, with 82% of survey participants identifying it as their primary use case.

All of the most widely-used AI tools for programming employ a text-based form of interaction, wherein the user conveys intent either through a textual description of the desired functionality or by providing previously-written code with annotations indicating what should be written next (e.g., comments). Such a method has proven effective in numerous contexts but also presents certain limitations, particularly in situations where a graphical reference would serve as the starting point. This scenario is commonly encountered in front-end development, where a mock-up of the user interface is iteratively created in collaboration with the client and subsequently serves as the foundation for the actual development process.

Handling graphical references is particularly interesting and problematic from a front-end developer's perspective because designing a front-end application relies heavily on a preliminary visual representation of the desired interface, which is often either provided by the client or created in close collaboration with them. This visual aspect is crucial, as it directly influences the layout, user experience, and overall functionality of the application. Therefore, incorporating this visual dimension into AI-based code generation could significantly enhance its effectiveness. Furthermore, the field of AI-driven code generation from visual inputs has so far

received limited attention. While some preliminary solutions have been proposed, these are constrained in a way that limits their applicability in professional settings.

To address this limitation, various approaches have been explored. Major AI service providers have developed multi-modal LLMs with vision capabilities [4, 5] that can accept a combination of textual and graphical inputs to generate an output. These capabilities have been integrated into other products, allowing for the incorporation of images in the prompting process. Although this feature has proven valuable for executing more advanced activities, it remains inadequate when applied to programming tasks. This inadequacy is primarily due to the limited precision with which state-of-the-art models can analyze the provided reference and their insufficient knowledge of specific front-end frameworks and libraries.

Moreover, research has led to intriguing developments in code generation from graphical references. Studies have demonstrated that deep learning methods can be employed to transform a graphical user interface screenshot created by a designer into computer code [6]. Further research has focused on comparing this novel approach with traditional computer vision techniques, as illustrated in Alex Robinson's *Sketch2Code* [7]. Despite their theoretical effectiveness, these approaches possess significant drawbacks that hinder their large-scale implementation for professional use:

- Beltramelli's *Pix2Code* [6] project achieves promising results using a Convolutional Neural Network (CNN) with unsupervised feature learning. However, it does so by training on a very specific dataset, concentrating on the recognition of a limited group of UI elements with similar graphical features. Extending this functionality to a more general level would necessitate the collection of a significantly larger dataset and a prolonged training process, both of which would be prohibitively costly given the uncertain return on investment.

- Robinson's *Sketch2Code* [7], although remarkable in its results, is similarly constrained by its high specificity regarding the types of images it is designed to handle and the restricted set of UI elements it can detect.

Given these considerations, the aim of the thesis is to explore an innovative approach to AI-assisted code generation based on images, leveraging a hybrid methodology that incorporates traditional computer vision algorithms and modern image processing based on a multi-modal LLM. Additionally, a concrete implementation of such process is analyzed, in the form of an experimental application designed for internal use at Blue Reply. The output of the application is in the form of front-end code for the target language and framework of choice; the generated code implements all the visual features of the input image using the available components provided by the chosen platform, acting as a foundation on top of which the developer is tasked with adding the business-related functionality. By

employing this combined methodology, the aim is to improve the efficiency and accuracy of the transition from graphical design to code, facilitating a more effective front-end development process.

1.2 Overall Thesis Structure

This thesis is structured to guide the reader through the development and evaluation of the previously-introduced approach to generating front-end code from images, leveraging computer vision and multi-modal LLMs.

Chapter 2 establishes a theoretical foundation by presenting a general overview of the proposed solution, focusing on the technical aspects of generating front-end code from images. It includes a detailed description of the application architecture and workflow, as well as the implementation strategies employed. The integration of various technologies and their combined use is thoroughly explored, setting the stage for a deeper exploration of the specific techniques and methodologies utilized.

Subsequently, Chapter 3 begins the exploration of the core components of the solution with an in-depth analysis of the computer vision techniques used for image processing. Such chapter examines the selected methods, discussing their strengths and limitations, as well as the pre-processing steps necessary to enhance their effectiveness. The implementation details, including the specific libraries and tools used, are also thoroughly covered.

Following this, Chapter 4 shifts the focus to the role of multi-modal LLMs in analyzing the detected graphical elements within images. The analysis process, including prompt engineering techniques to optimize output quality, is discussed in detail. Additionally, the optimizations aimed at accelerating the analysis process and minimizing associated costs are examined, providing a comprehensive view of the LLM's application in this context.

The process of inferring the layout of the page from the extracted information is then examined in Chapter 5, addressing the strategies for organizing the layout and the challenges associated with nested structures. This includes a detailed explanation of the methods used to translate visual elements into a coherent code structure, highlighting the practical considerations involved.

Attention is also given to a practical implementation of the described workflow in the form of a web application. The integration of the various functional modules within it is illustrated, describing the overall architecture and the interactions between different components. In Chapter 6, a functional analysis of the application is presented, emphasizing the design principles and guidelines adopted to enhance user experience and ensure seamless integration. Furthermore, the choice of a web-based architecture and the reasons behind various technical and design solutions are discussed.

Chapter 7 is dedicated to the evaluation and results, assessing the initial objectives and the outcomes achieved. Real use-cases employed during the testing phase are discussed, demonstrating the practical application and effectiveness of the proposed solution. This evaluation provides critical insights into the performance and potential areas for improvement.

Finally, the thesis concludes with a reflection on the advantages and limitations of the proposed solution in Chapter 8, discussing the implications of the findings and suggesting potential future developments.

Chapter 2

General Process and System Structure

This chapter analyzes the current state-of-the-art solutions proposed for image-based code generation based on the latest relevant research, to then describe the adopted approach fo the thesis project. In detail, an overview of the proposed solution will be provided; after that, its actual implementation in the form of a web application will be discussed, highlighting the desired workflow for its primary users and the technical decisions that were taken in its implementation.

2.1 State of the Art

As previously discussed, the core idea of the thesis project is to use graphical input to automatically generate a portion of the code needed by frontend developers to implement sections of an application.

Beyond generating a general representation of the page that can be converted into code, the primary challenge lies in recognizing the User Interface (UI) elements present in the image and their logical arrangement within the page's layout. Existing research on this topic has explored three main approaches to address this problem:

- Computer Vision-based recognition.
- Neural Network-based recognition.
- Hybrid methods that combine Computer Vision and Deep Learning for UI element extraction.

A Neural Network-based solution is exemplified by T. Beltramelli's *Pix2Code* [6]. This approach employs a combination of deep learning models to automate

the generation of code from GUI screenshots. The system's architecture integrates Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), specifically Long Short-Term Memory networks (LSTMs). The CNN extracts spatial features from the input image and encodes them into a fixed-length vector, while the LSTM processes a sequence of tokens representing code in a Domain-Specific Language (DSL). Through end-to-end training using gradient descent, the model learns to generate code token-by-token, predicting each token based on the input image and previously generated tokens [6].

A more modern solution is illustrated by Google's *ScreenAI* project, a vision-language model designed to understand User Interfaces and infographics by leveraging a combination of deep learning techniques [8]. It builds upon the PaLI architecture and incorporates a flexible patching strategy inspired by *pix2struct* [9]. The model processes inputs in the form of both text and images, transforming them into a text-based output through a multimodal encoder-decoder architecture. The vision encoder, a variant of the Vision Transformer (ViT), extracts visual features from an image by dividing it into adaptive patches, accommodating various resolutions and aspect ratios. These image features are then integrated with text embeddings from the mT5 language encoder, enabling the model to interpret both visual elements and their corresponding text annotations. The output format produced by the *ScreenAI* model is a structured text format, often referred to as a *screen schema*, which provides a comprehensive description of the screen's structure and content, enabling further interaction or processing tasks [8].

Deep Learning-based methods can be highly effective, but they face significant limitations due to the substantial resources required for pre-training. These processes demand the creation of extensive datasets and the use of large computational resources. In the case of *Pix2Code*, this challenge was addressed by narrowing the focus to a specific task - analyzing screenshots of Bootstrap-based images. Although this approach is powerful, it is limited by its applicability to only a specific subset of inputs.

A notable research project in this domain is A. Robinson's *Sketch2Code*, which explores various techniques for converting hand-drawn wireframes into HTML code [7]. This project introduces a Computer Vision-based approach, utilizing the Canny edge detector to identify edges within the input image. Following Edge Detection, Contour Detection is applied to segment the image into distinct shapes. Based on contour characteristics such as aspect ratio and solidity, the system classifies different components, such as buttons, text fields, and images [7]. This method achieves efficiency by avoiding the use of Neural Networks; however, the heuristic-based classification leads to a considerable margin of error and limits further analysis of detected elements, such as identifying color or text content.

The same project also proposes a hybrid approach, combining classical Computer Vision techniques with Deep Learning methods. In this approach, Deep Learning is

primarily employed for semantic segmentation, enabling the model to classify each pixel of an image to more accurately identify elements [7]. A Convolutional Neural Network is specifically used for this purpose, allowing the system to detect and segment components based on visual features. Concurrently, classical Computer Vision methods, such as image denoising, edge detection, and contour detection, are applied in preprocessing stages to simplify the image before passing it to the Deep Learning model.

This combination yields more accurate results compared to the purely Computer Vision-based approach [7]. The objective of this thesis is to build upon this foundational concept, using similar Computer Vision techniques alongside more advanced analysis methods based on Large Language Models. The thesis also seeks to develop a more generalizable solution that is not restricted to a narrow set of use cases. By refining the techniques and processes employed, this project aims to create an efficient tool to accelerate frontend development.

2.2 Overview of the Proposed Solution

The proposed solution is based on the possibility of combining traditional computer vision tasks with modern, LLM-based analyses in order to generate front-end code for the desired target platform fast and easily. In particular, the generation process can be subdivided in the following general steps:

1. **Computer Vision Analysis**: in this phase, elements of interest are extracted from the provided image to be analyzed by a multi-modal LLM. To do so, a sequence of activites are performed:

   (a) The image is appropriately pre-processed to increase the quality of the results obtained from the subsequent steps.

   (b) Contour and edge detection techniques are used to associate regions of the image with relevant UI elements.

   (c) An Optical Character Recognition engine is used to scan the image for text; the obtained information is combined with the already-extracted regions of interest.

   (d) A set of post-processing steps are applied to the detected UI elements to reduce errors due to limited detection precision.

2. **Analysis of Detected Elements with Multi-modal LLM**: the list of all the detected elements and their corresponding graphical data is fed to a multi-modal LLM that is instructed to provide a standard description of the element in a well-defined format.

3. **Layout Generation**: after the analysis process, the enriched UI elements are spatially grouped together in lists of rows and columns to organize them within a well-defined page layout; the obtained results consists in the *General Page Document* (GPD), i.e. a generic representation of the page content and style that can be used to generate code for any desired target platform.

4. **Code Generation**: the obtained GPD is converted to a set of artifacts which contain the necessary code to render the content of the page on the desired platform through the use of a dedicated translation module.

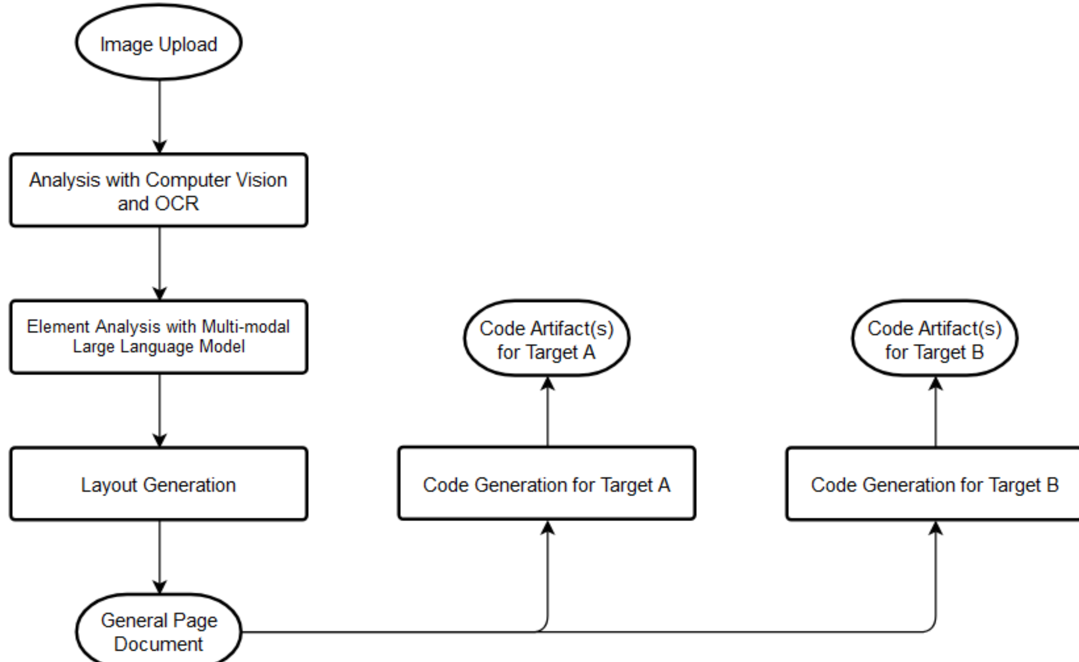


Figure 2.1: Flow chart of the general adopted process

As can be seen, the proposed process allows to obtain a language-agnostic representation of the content of the web page (GPD); this proves significant in a context where different development teams are required to use of different languages and frameworks. The analyzed approach can thus easily be adopted in any development environment, granted that it is possible to develop the needed translation module.

2.3 Application Structure and Workflow

In order to provide the proposed solution in the form of a tool that developers could easily use and integrate in their ordinary workflow, it was decided to build a web application. There were several reasons behind this choice:

- Cross-platform compatibility: the web application functions seamlessly across different operating systems without requiring specific modifications for each platform, ensuring a uniform user experience.
- Integration with other tools: web applications can easily connect with other web-based tools and services, such as version control systems or project management platforms, allowing for further enhancements and efficiency.
- Rapid prototyping: the web application model supports quick development and deployment, enabling developers to rapidly test and refine new features based on user feedback.
- Previous experience: the existing expertise and infrastructure at Blue Reply make it well-equipped to implement and manage a web-based solution efficiently, especially for internal use.

As a result of this choice, the development process was split between the implementation of a front-end and a back-end.

Focusing on the back-end, the application is organized in a set of modules, each one focusing on a specific part of the overall analysis and generation process. This structural choice makes the application modular and allows to isolate smaller parts of the overall functionality in well-defined entities that can be used independently.

The following is the list of modules used to obtain a General Page Document starting from the initial image:

- **Element Detection Module**: this module focuses on extracting a list of UI elements from an mage and their relative locations and sizes. It uses the aforementioned computer vision techniques to do so, applying all the necessary pre-processing steps to increase the quality of the output.
- **Optical Character Recognition Module**: this module focuses on the extraction of text data from the input image. It is used in conjunction with the Element Detection Module to enrich the element list with the detected text elements.
- **Element Analysis Module**: this module is dedicated to the analysis of the detected elements. It establishes and manages a connection with a multi-modal LLM, and queries it with pre-defined prompts to generate a structured description of the received inputs, which is provided as the final output.

- **Layout Generation Module**: this module receives a list of elements and their locations within the image, and is tasked with organizing them into a well-defined layout, with nested rows and columns.

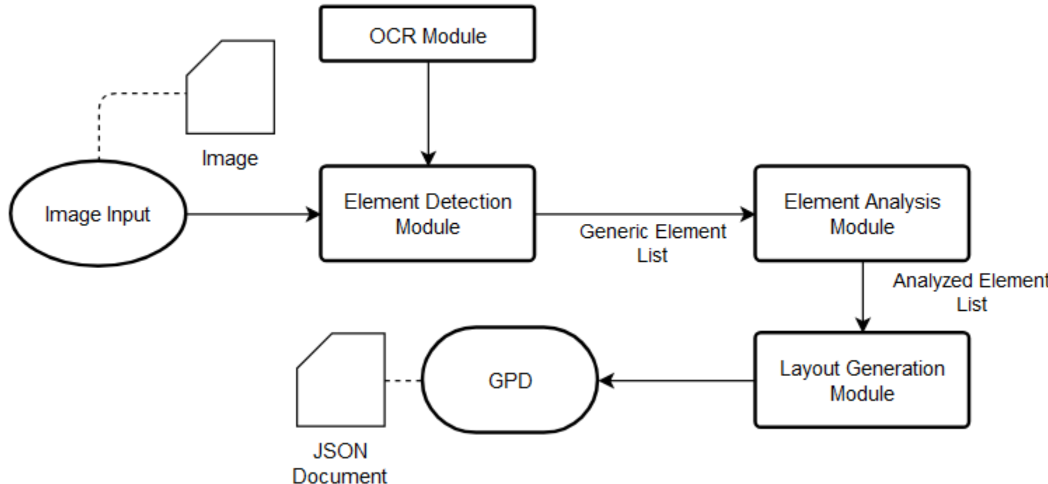


Figure 2.2: Flow chart representation of the application modules and the related workflow

The second major step in the application workflow consists in the transformation of the generated GPD into a set of code artifacts that allow it to be reproduced on the target platform. As such, an appropriate set of Translation Modules needs to be implemented, to allow users to export the obtained results to their platforms of interest.

In the case of Blue Reply, the technology stack often includes **Angular** [10] as the front-end framework of choice. Angular offers several advantages as a web development framework, including a robust MVC architecture that promotes clear code organization and efficiency, as well as the use of TypeScript [11], which enhances code quality by catching errors during development. Furthermore, two-way data binding simplifies synchronization between the model and the view, thereby speeding up development, while Angular's extensive built-in tools and libraries streamline various functionalities. Supported by a strong community and regular updates from Google, Angular represents the primary choice for the development of modern enterprise applications.

More specifically, most Blue Reply products employ a vast ecosystem of libraries aimed at streamlining and speeding up the development process. In the context of building modern and responsive user interfaces, the library of choice is **PrimeNG** [12]. It provides a comprehensive set of over 80 UI components, which significantly

reduces the time and effort required to create visually appealing and functional interfaces. The library's components are highly customizable, enabling developers to tailor them to specific project requirements, while its robust theming system allows for consistent and attractive design across applications. In addition, PrimeNG's active community and frequent updates ensure that it remains compatible with the latest Angular versions and incorporates the latest web standards.

The use of a specific UI library requires the use of a dedicated translation module that is able to effectively employ the provided components to reproduce the interface described by the General Page Document. For this reason, an **Angular/PrimeNG Translation Module** has been included in the project. This module is capable of receiving a GPD and producing a set of three artifacts:

- **TypeScript File**: the component's core, containing the logic, data properties, and methods that define its behavior and interaction with other parts of the application. It includes decorators that link the component to its associated template and style files.
- **HTML File**: the component's template, defining its structure and layout by using Angular's templating syntax to bind data and manage dynamic content.
- **CSS File**: the styling for the component.

In our specific case, particular attention is dedicated to the generation of an HTML template that accurately depicts the user interface described by the starting GPD.

Furthermore, an additional **Bootstrap Translation Module** was implemented, primarily to provide an alternative that the programmer could use to generate code that could more generally be adapted to a different framework of choice.

In this case, the generated code uses the popular **Bootstrap** framework (version 5) [13]. Bootstrap provides an extensive library of CSS and JavaScript utilities facilitates rapid development by offering reusable code for common interface elements, such as navigation bars, forms, buttons, and modals. It is widely adopted across the web development community, ensuring consistent support and updates [14]; its ease of use and versatility allow to use it in combination with all major front-end frameworks.

This module produces two output artifacts:

- **HTML File**
- **CSS File**

The generated HTML and CSS code can then be reused by the programmer for the specific required use case.

2.4 Implementation Strategy

The implementation strategy chosen for the overall application is a classic client-server web architecture. This means that the development effort was divided between the implementation of the front-end, delivered to each user through a web browser, and the back-end, which is in charge of providing the actual functionality and interfacing with the front-end by using an appropriate API.

In regards to the front-end, the framework of choice was Angular [10] in combination with Bootstrap [13]. As previously stated, Angular represents the primary framework of choice at Blue Reply; its usage ensures the possibility of collaboration with other teams within the business unit as well as the availability of additional resources to be allocated to the project for future developments.

The application back-end was subject to more consideration, as multiple languages and frameworks were considered viable options for the project in question. Most Blue Reply products rely on **Spring Boot**, a Java-based application framework that uses Inversion of Control, Dependency Injection and Aspect-Oriented Programming to facilitate the development of web applications. In addition, a large ecosystem of libraries and extensions are available to expand Spring Boot's functionality; this allows to quickly add additional functionality to the web application with minimum configuration from the programmer, in a plug-and-play manner. These factors, as well as the large community around it and the vast amount of available resources, render Spring Boot the best choice for the majority of use cases.

Despite these considerations, the final decision was to use a Python-based backend. Python is widely embraced in the machine learning community due to its simplicity and readability, making it accessible for both beginners and experienced developers. This widespread adoption has led to the creation of a rich ecosystem of libraries, such as TensorFlow, Keras, PyTorch and LangChain, which streamline the development of AI-based programs. These libraries offer powerful tools and pre-built models, enabling developers to efficiently build, train, and deploy machine learning applications. Additionally, many powerful and open-source computer vision and image processing Python libraries are available. These observations ultimately highlighted Python as the most suitable option for our use case.

The development of a web application requires the implementation of a web server that exposes a set of APIs to communicate with the client. There are many Python frameworks dedicated to the implementation of web servers; in order to ensure ease of development and support rapid iteration, we decided to use the **Flask** framework to expose a simple REST API. Flask is a lightweight WSGI web application framework. It is designed to make getting started quick and easy, with the ability to scale up to complex applications [15].

Chapter 3

Computer Vision and Optical Character Recognition for the Initial Image Analysis

This chapter presents a detailed description and analysis of the Computer Vision techniques employed for detecting UI elements within the input image. This task constitutes the initial phase in the process of converting the image into its General Page Description, and the accuracy of this step is crucial for achieving a satisfactory final outcome. Furthermore, the pre-processing and post-processing steps applied to the input image are illustrated in detail.

3.1 Adopted Computer Vision Techniques

To make the detection of elements possible, two main Computer Vision techniques were adopted: **Edge Detection** and **Optical Character Recognition** (OCR). Additionally, **Contour Detection** is used to extrapolate structured topological information from the obtained edges.

Edge Detection is a fundamental concept in image processing, concerned with identifying and locating sharp discontinuities in intensity within an image. These discontinuities, often corresponding to object boundaries or significant changes in surface orientation, are critical for interpreting the structure and features of the visual data. By isolating edges, it becomes possible to simplify the image and focus on its most salient aspects, facilitating further analysis and interpretation [16].

In the case of the described image analysis process, an additional Contour

Detection step is applied. Contour Detection is a technique that extracts contours from a binarized image using one of many available algorithms. This process involves identifying and tracing the continuous boundaries that define the shape of objects within an image, providing a more structured representation of the detected edges. By focusing on the contours, it becomes possible to capture the geometric properties of objects, facilitating their recognition and analysis in Computer Vision applications.

Optical Character Recognition (OCR) is aimed at the automated extraction of textual information from images or scanned documents. By converting visual representations of text into machine-encoded data, OCR enables the efficient digitization, indexing, and retrieval of graphical textual content. This process involves the recognition of individual characters, words, and sentences, making it indispensable for a range of diverse applications.

3.2 Edge Detection for Relevant UI Elements

Edge Detection is the first fundamental step in detecting UI elements within the input image. By leveraging existing Edge Detection algorithms, and applying appropriate pre-processing to the image, it is possible to obtain a binarized output that accurately highlights the presence of relevant UI elements.

3.2.1 The Problem of Edge Detection

Edge Detection is a process that aims to detect and characterize discontinuities in the image domain [17]. In usual Computer Vision tasks, Edge Detection is adopted to highlight differences in either depth, surface orientation, illumination, or reflectance in the portrayed physical objects; however, the capabilities of this process are also well-suited to the detection of graphical elements within a screenshot of a web page or a mock-up of a graphical user interface. The discrete nature of such image representations and the well-defined borders between differently-colored shapes makes the usage of Edge Detection techniques ideal to provide initial spatial and geometric information about the available visual components.

The process of Edge Detection is characterized by a set of challenges that need to be overcome with the use of appropriate smoothing techniques as well as the adoption of proper methods to identify image intensity changes at different scales [16].

In regards to the latter problem, different mathematical approaches and corresponding methods can be followed; more specifically, they can be **search-based** (based on a first-order derivative expression from the image) or **zero-crossing-based** (based on a second-order derivative expression from the image). Such approaches are described more in depth in the following chapter.

3.2.2 Edge Detection Methods

Gradient-based techniques [18] represent one of the most straightforward approaches for edge detection. These methods are based on the computation the first-order derivative of the image intensity function $I(x, y)$ at every image location. The gradient at a point (x, y) in the image is a vector given by:

$$\nabla I = \left(\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y} \right)$$

The magnitude of this gradient vector:

$$|\nabla I| = \sqrt{\left(\frac{\partial I}{\partial x} \right)^2 + \left(\frac{\partial I}{\partial y} \right)^2}$$

indicates the strength of the edge, while the direction of the vector indicates the direction of the steepest intensity change. The computed magnitude can then be compared to a threshold, to verify whether the corresponding location is part of an edge or not.

This approach uses few, simple operations like convolution and difference calculation, making it a simple and effective solution, especially in constrained environments that require a low computational cost. Prime examples of Edge Detection methods based on the first-order derivative are the Roberts Cross detector and the Sobel detector; these methods use convolution kernels to estimate the value of the intensity gradient $I(x, y)$ at every image location to then compute its magnitude.

The **Roberts Cross operator** for Edge Detection is known for its simplicity and speed, and it is especially useful in detecting edges in images with sharp, high-contrast changes [18]. It was first proposed by Lawrence Roberts in 1963 [19]. It computes the gradient of the image intensity function $I(x, y)$ by applying two convolution kernels of size 2×2 to the input image:

$$G_x = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \quad G_y = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

These convolutions are combined to determine the overall gradient magnitude ∇I for each point:

$$|\nabla I| = \sqrt{G_x^2 + G_y^2}$$

The Roberts Cross edge detector is effective when working with high-contrast images; its simplicity and small kernel sizes make it a viable choice in resource-constrained environments, where computational efficiency is crucial. However, it suffers from several issues:

- Limited Edge Detection capability due to the small kernel sizes, which lead to the loss of finer details and broader edge features.

- High sensitivity to noise, which can lead to the detection of several false edges.

Due to its limitations, the Roberts Cross detector is generally superseded by more modern and powerful alternatives, such as the Sobel detector.

The **Sobel operator** is a robust and widely used tool for Edge Detection [20]; it was first proposed in 1968 by Irwin Sobel and Gary Feldman as an alternative to the aforementioned Roberts Cross detector. It is a discrete differential operator that computes the gradient of the image intensity. It uses two convolution kernels, typically 3×3 in size, one for the horizontal direction (G_x) and one for the vertical direction (G_y) [21]:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

These kernels are designed to have the strongest response to edges that run vertically or horizontally along the pixel grid, with one kernel for each of these two perpendicular directions. Like the Roberts Cross detector, the kernels can be applied individually to the input image to measure the gradient in each direction separately. These measurements can then be combined to determine the overall magnitude of the gradient ∇I at each point. The gradient magnitude is computed as:

$$|\nabla I| = \sqrt{G_x^2 + G_y^2}$$

Typically, for a faster computation, the following approximate magnitude is computed instead [21]:

$$|\nabla I| = |G_x| + |G_y|$$

The main advantage of the Sobel operator is its simplicity, due to its approximate method for calculating gradients. However, its major disadvantage is the signal-to-noise ratio. As noise increases, the gradient magnitude of the edges deteriorates, leading to less accurate results [22].

The **Laplacian of Gaussian** is a second-order derivative-based Edge Detection method that uses a combination of two different convolutions to detect regions of the image with rapid intensity change [23]. Given an image with an intensity function $I(x, y)$, the Laplacian of each of its locations $L(x, y)$ is computed as follows:

$$L(x, y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

It is possible to compute an approximated value of the Laplacian for each pixel of the input image using a discrete convolution kernel. An example of a commonly used convolution kernel K is the following [23]:

$$K = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

This kernel approximates a second derivative on the image intensity, making it highly sensitive to noise; to address this, Gaussian smoothing is typically applied to the image before the Laplacian filter is used, reducing high-frequency noise before differentiation. The **Gaussian filter** is a low-pass filter that smooths the image by averaging the intensity values of neighboring pixels, which helps to suppress high-frequency noise.

Since convolution is an associative operation, the Gaussian smoothing filter can be combined with the Laplacian filter first. This combined filter can then be applied to the image, offering two key benefits:

- Both the Gaussian and Laplacian kernels are usually much smaller than the image, reducing the number of arithmetic operations required.
- The Laplacian of Gaussian (LoG) kernel can be precomputed, allowing for a single convolution to be performed on the image at runtime.

The two-dimensional Laplacian of Gaussian (LoG) function centered on zero and with standard deviation σ has the following form [23], obtained by applying the Laplacian operator on the Gaussian filter:

$$\mathrm{LoG}(x, y) = -\frac{1}{\pi\sigma^4}\left(1 - \frac{x^2 + y^2}{2\sigma^2}\right)\exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

This function can be estimated using a discrete convolution kernel. Figure 3.1 shows an example of a LoG convolution kernel for a Gaussian filter with $\sigma = 1.4$ [23].

The LoG filter has the effect of highlighting edges in the image. The produced output contains negative and non-integer values that need to be normalized between 0 and 255.

An efficient approximation of the LoG filter is given by the **Difference of Gaussians** (DoG) filter [24]. It is computed by subtracting one Gaussian-blurred version of an image from another, where each version is blurred with a different standard deviation. This subtraction highlights edges and details by enhancing regions of rapid intensity change, similarly to the LoG filter. However, the DoG filter is computationally less intensive, making it a practical alternative for Edge Detection and other tasks requiring high spatial frequency analysis.

0	1	1	2	2	2	1	1	0
1	2	4	5	5	5	4	2	1
1	4	5	3	0	3	5	4	1
2	5	3	-12	-24	-12	3	5	2
2	5	0	-24	-40	-24	0	5	2
2	5	3	-12	-24	-12	3	5	2
1	4	5	3	0	3	5	4	1
1	2	4	5	5	5	4	2	1
0	1	1	2	2	2	1	1	0

Figure 3.1: Discrete approximation to LoG function with Gaussian $\sigma = 1.4$, from [23].

The **Canny Edge Detection algorithm** offers a more advanced and reliable approach to edge detection, addressing many of the shortcomings found in the previously described methods. This algorithm integrates multiple processing steps to produce a precise and well-defined edge map. Due to its strengths, the Canny algorithm has been selected for use in this thesis project, providing a reliable foundation for further analysis and implementation, as detailed in the following chapter.

3.2.3 The Canny Edge Detection Algorithm

The Canny Edge Detection algorithm is an advanced edge detection methodology proposed by John Canny in 1986 [25]. This algorithm has become a standard tool in the field due to its ability to produce clean and accurate edge maps while being robust against noise; as the most robust and reliable option, it was the edge detection technique of choice for the thesis project. An example of its usage on a realistic example for the purpose of this project is shown in Figure 3.2

The Canny algorithm consists of several sequential steps designed to optimize edge detection while minimizing the influence of noise and irrelevant information. These steps include:

- Gaussian smoothing
- Gradient calculation

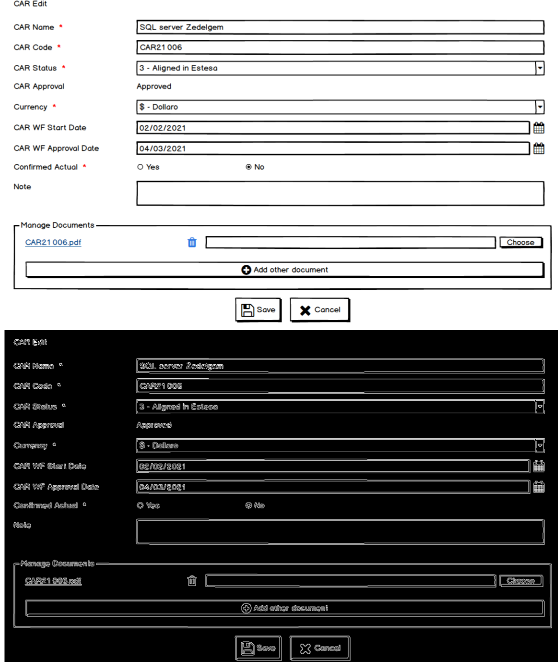


Figure 3.2: Example usage of the Canny Edge Detection algorithm on a user interface mock-up.

- Non-maximum suppression
- Double thresholding
- Edge tracking by hysteresis

The correct execution of each of these steps is crucial to obtain an accurate representation of the image edges.

Gaussian Smoothing

The first step in the Canny algorithm is to smooth the input image to reduce noise and small variations that could lead to the detection of false edges. This is achieved by applying the previously-mentioned **Gaussian filter** to the image. A two-dimensional isotropic (i.e. circularly symmetric) Gaussian with standard deviation σ has the form [26]:

$$G(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

The choice of σ affects the performance of the detector; by increasing its value, the sensitivity to noise of the filter is decreased, but there is also a risk of removing

relevant edges. For most practical applications, a 5×5 size yields good results; however, this will vary depending on specific situations.

Gradient Calculation

After smoothing, the next step involves computing the gradient of the image intensity. The gradient indicates the rate of change in intensity at each pixel, which is crucial for identifying edges. The gradient is calculated in both the horizontal and vertical directions, typically using Sobel operators, which are 3×3 convolution kernels designed to respond maximally to edges running vertically or horizontally [27].

The horizontal (G_x) and vertical (G_y) gradients are combined to compute the magnitude of the gradient ($|\nabla I|$) and its direction (θ) at each pixel:

$$|\nabla I| = \sqrt{G_x^2 + G_y^2}, \quad \theta = \arctan\left(\frac{G_y}{G_x}\right)$$

The magnitude represents the strength of the edge, while the direction indicates the orientation of the edge.

Non-maximum Suppression

Once the gradient magnitude and direction are obtained, the next step is to refine the edges by applying Non-maximum Suppression. This step aims to thin out the edges by preserving only the local maxima of the gradient magnitude along the direction of the gradient. For each pixel, Non-maximum Suppression compares the gradient magnitude with those of the neighboring pixels along the gradient direction. If the pixel's magnitude is not the largest, it is suppressed (set to zero) [25].

An example of such procedure is shown in Figure 3.3, where point A is located on the vertical edge, and the gradient is perpendicular to this edge. Points B and C are positioned along the gradient direction; to determine if Point A represents a local maximum, it is compared with Points B and C. If Point A is a local maximum, it is retained for further processing; otherwise, it is suppressed by setting its value to zero.

This process results in a set of thin, sharp edges, eliminating any broader or thicker edges that may have been detected in the previous step.

Double Thresholding

After Non-maximum Suppression, the resulting edge map may still contain weak edges that are not true edges, as well as strong edges that are more likely to be accurate. To distinguish between these, the Canny algorithm applies Double

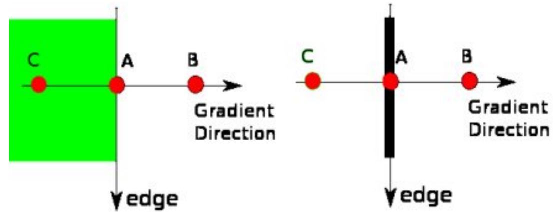


Figure 3.3: Example of non-maximum suppression, from [28].

Thresholding. Two thresholds are used: a high threshold (T_H) and a low threshold (T_L). Pixels with a gradient magnitude greater than T_H are considered strong (certain) edges, while those with a gradient magnitude between T_L and T_H are considered weak (uncertain) edges [25].

Pixels with a gradient magnitude below T_L are discarded as non-edges. This thresholding step effectively separates potential edges from noise and weaker variations.

Edge Tracking by Hysteresis

The final step in the Canny algorithm is Edge Tracking by Hysteresis. This step ensures that weak edges that are connected to strong edges are retained, as they are likely part of the same edge, while isolated weak edges are discarded as noise. Starting from the strong edges identified in the double thresholding step, the algorithm traces along connected pixels, including weak edges that form continuous lines with the strong edges [25].

This process produces the final edge map, which contains only the most significant and continuous edges in the image.

Advantages of the Canny Algorithm

The Canny Edge Detection algorithm offers several advantages over other Edge Detection methods:

- **Robustness to Noise**: the initial Gaussian smoothing step effectively reduces noise, making the Canny algorithm more resistant to false edge detection compared to gradient-based methods like the Sobel or Roberts Cross operators.

- **Edge Localization**: the combination of gradient calculation, non-maximum suppression, and edge tracking ensures that edges are accurately localized, preserving fine details and sharp transitions.

- **Detection of True Edges**: the Double Thresholding and Hysteresis steps help in distinguishing true edges from noise, resulting in a cleaner and more reliable edge map [25].

3.2.4 Pre-processing Steps for Improved Results

As mentioned earlier, a crucial element in any computer vision task is selecting suitable pre-processing steps for the input image to improve the quality of the results. The Canny Algorithm inherently includes some pre-processing through the application of a Gaussian smoothing filter. However, initial experiments with the Edge Detection process have shown that additional pre-processing steps are necessary to achieve optimal results when working with screenshots or UI mock-ups.

A basic initial step is converting the image from the BGR (Blue/Green/Red) color space to grayscale. This conversion is essential because it reduces the color space to a single dimension, which serves as the reference intensity function.

After the conversion to grayscale, **Contrats-Limited Adaptive Histogram Equalization** (CLAHE) is applied to the image to enhance its features and improve contrast between similarly-tinted UI elements.

Histogram Equalization is a fundamental image pre-processing technique designed to enhance the contrast of an image by redistributing its intensity values. In many images, especially those with low or uneven contrast, pixel intensity values may be clustered within a narrow range, leading to a loss of detail in darker or lighter areas. Histogram Equalization addresses this by mapping the original intensity values to new values that are more evenly distributed across the available intensity range. This process enhances the overall contrast, making important features of the image more visible and improving the clarity of the visual information [18].

While traditional Histogram Equalization can be effective, it has notable limitations, particularly in images with large uniform regions or significant noise. The global nature of this technique means that it applies the same transformation across the entire image, which can lead to over-enhancement of noise and the introduction of artifacts.

To overcome these limitations, Contrast-Limited Adaptive Histogram Equalization was developed. Unlike traditional Histogram Equalization, CLAHE operates on smaller, localized regions of the image, known as tiles. By applying Histogram Equalization independently within each tile, CLAHE can enhance contrast in specific areas without affecting the global image appearance. This localized approach is particularly beneficial for images with irregular textures, as it allows for more targeted enhancement where it is needed most [29].

A key feature of CLAHE is its contrast-limiting mechanism, which prevents over-amplification of noise. During the equalization process, CLAHE imposes a clip limit on the histogram, ensuring that any intensity values exceeding this

limit are redistributed. This helps to maintain a balance between enhancing the contrast and preserving the natural appearance of the image, minimizing the risk of introducing artifacts. After equalization, the tiles are seamlessly combined using bilinear interpolation to avoid any blocky artifacts that might result from the independent processing of each tile.

The two pre-processing techniques discussed have proven effective in improving the quality of the Edge Detection results. Figure 3.4 presents an overview of the pre-processing steps applied to the input image.

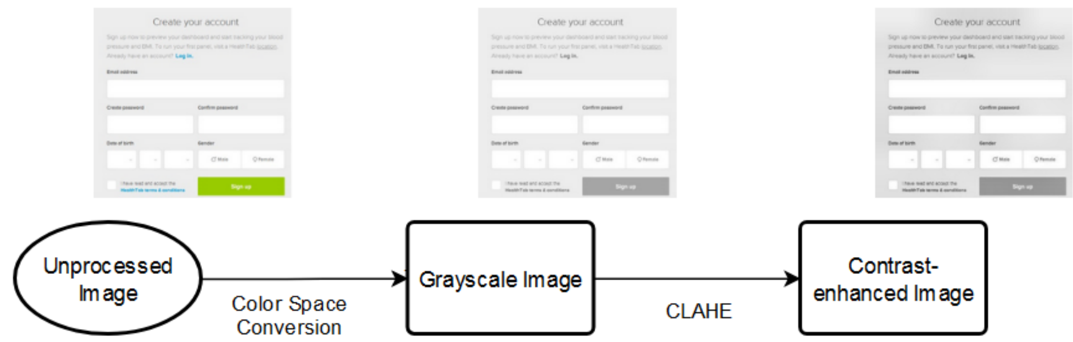


Figure 3.4: Overview of image pre-processing for Edge Detection.

3.3 Optical Character Recognition for Textual Content Detection

Graphical representations of user interfaces typically include a combination of geometric shapes, representing interactive elements like buttons and form inputs, along with textual content. To accurately capture the interface's content, it is essential to scan the input image for text and integrate this information with the previously detected edges. Optical Character Recognition (OCR) techniques are used for this purpose.

3.3.1 The Problem of Optical Character Recognition

Optical Character Recognition (OCR) is a technology that enables the conversion of different types of documents - such as scanned paper documents, PDFs, or screenshots - to machine-encoded text. This capability has widespread applications, including automating data entry, extracting information from graphical sources and enabling digital archiving.

Despite its potential, OCR faces several challenges that impact its accuracy and effectiveness. One major issue is related to the quality of the input image. Low resolution, poor contrast, and noise can significantly degrade the readability of text, making it difficult for OCR systems to accurately recognize characters. Additionally, the variability in text appearance - such as differences in fonts and sizes - complicates the recognition process. The system must be adept at handling diverse textual forms and aligning characters correctly, which is further complicated by the intricacies of different languages and scripts. The complexity of natural language also plays a role, as OCR systems must often interpret characters in context to resolve ambiguities and improve recognition accuracy [30].

Addressing these challenges involves advanced techniques in image pre-processing, machine learning, and natural language processing. Improvements in these areas aim to enhance the robustness and accuracy of OCR systems, making them more effective in handling the diverse text data encountered in real-world applications.

3.3.2 Modern Techniques for OCR

Optical Character Recognition (OCR) faces numerous challenges that impact its effectiveness. Traditional OCR techniques have made significant contributions to the field but also encounter limitations. These methods exploit gradient analysis and convolutional filters, and are based on the same Edge Detection techniques analyzed in Section 3.2.2. These conventional approaches often struggle with issues related to image quality, text variability, and language complexity. Gradient-based methods like the Roberts Cross and Sobel detectors, while useful for detecting edges, do not inherently address the sequential nature of text or the variability in character appearance. Similarly, the Laplacian of Gaussian and Difference of Gaussians methods provide edge detection but fall short in handling complex text data and diverse fonts.

A more effective approach to OCR involves using Recurrent Neural Networks (RNNs) in combination with Convolutional Neural Networks (CNNs). This combination enables thorough analysis of the text in the input image by recognizing key features for character identification and understanding the associations between related graphical elements.

Reccurrent Neural Networks

Recurrent Neural Networks (RNNs) present a more advanced solution to the challenges of OCR by incorporating mechanisms specifically designed to handle sequential data. RNNs are a class of artificial neural networks designed to handle sequential data by leveraging their internal memory. Unlike traditional feedforward neural networks, which process inputs in isolation, RNNs use their internal state

to maintain information across time steps, allowing them to capture dependencies and patterns in sequential data [31]. This is achieved through feedback connections that enable the network to incorporate information from previous inputs into the processing of current inputs.

The ability to retain and utilize historical information allows RNNs to make more informed predictions and decisions based on the entire sequence rather than individual data points. Unlike traditional OCR techniques that may process text in isolation, RNNs can maintain context over sequences of characters, which is crucial for accurate text recognition. This capability enables RNNs to consider the relationships between adjacent characters, improving their ability to decipher text accurately even in the presence of noise or distortion [32].

Long Short-Term Memory

Despite their advantages, standard RNNs face challenges in learning long-term dependencies due to issues such as vanishing gradients during back-propagation. This limitation affects their ability to retain information across long sequences, which is necessary for accurate text recognition in lengthy documents or intricate images. To overcome this, Long Short-Term Memory (LSTM) can be employed.

Long Short-Term Memory networks are a specialized type of Recurrent Neural Network designed to address the limitations of traditional RNNs, particularly the vanishing gradient problem. Introduced by Hochreiter and Schmidhuber in 1997, LSTMs incorporate memory cells and gating mechanisms that enable them to maintain and manage information over extended sequences more effectively [33].

The LSTM architecture is designed to add a module to a neural network that learns when to retain and when to discard relevant information [34]. Essentially, the network learns which information should be preserved for future use in a sequence and when it can be disregarded. A memory cell is a fundamental component of the additional LSTM module, designed to store and manage information over extended periods; it works in conjunction with three key components: the input gate, the forget gate, and the output gate. These gates regulate the flow of information into, out of, and within the memory cell, allowing the network to selectively retain or discard information as needed [33]. This design helps LSTMs capture long-term dependencies and patterns in sequential data, making them highly effective for tasks such as language modeling and sequence prediction where context from previous time steps is crucial.

Convolutional Neural Networks

The effectiveness of RNNs in OCR is further amplified when combined with Convolutional Neural Networks. Convolutional Neural Networks are a class of deep learning models specifically designed to process and analyze grid-like data

structures, such as images. In CNNs, convolutional layers apply a series of filters to input data, extracting hierarchical features like edges, textures, and patterns at multiple levels of abstraction [35]. This ability to capture spatial hierarchies makes CNNs particularly effective for image-related tasks.

In Optical Character Recognition, CNNs play a crucial role by pre-processing image data to identify and extract features relevant to character recognition. They enhance the OCR system's capability to handle variations in font, size, and orientation of text. By learning to recognize complex patterns in text images, CNNs improve the accuracy and robustness of OCR systems, enabling them to accurately interpret and digitize text from diverse and noisy environments [36].

3.3.3 The Tesseract OCR Engine

For the use case of the thesis project, it was decided to use the Tesseract OCR Engine to extract textual content from the input images [37]. Tesseract is available for many operating systems, and is released as free software. It was initially developed as proprietary software by Hewlett-Packard in the 1980s, before being released as open source in 2005. It organizes the image analysis process into a multi-step pipeline [38]:

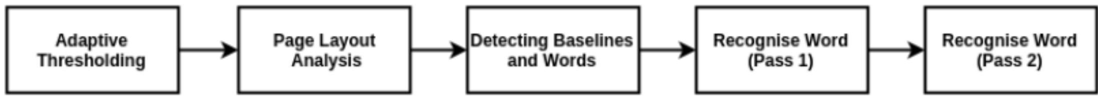


Figure 3.5: Architecture of the Tesseract OCR Engine, from [38].

1. **Adaptive Thresholding**: Tesseract uses Otsu's method [39], a clustering-based algorithm, to determine the optimal threshold for converting an image into black and white. This method minimizes intra-class variance while maximizing inter-class variance, effectively separating the text from the background. Adaptive Thresholding accounts for variations within the image by applying this process to small sections, ensuring better accuracy in text detection.

2. **Page Layout Analysis**: this step involves detecting and removing non-text elements like images and vertical lines using morphological processing. The text regions are identified through connected component analysis, and these components are grouped into column partitions. The page is then segmented into blocks of text, and their reading order is determined using heuristic rules, allowing Tesseract to handle multi-column documents effectively.

3. **Baseline Fitting and Word Detection**: Tesseract performs connected component analysis to detect individual blobs that represent parts of characters.

These blobs are sorted and grouped into rows based on vertical overlap, which helps in determining the skew of the text. A quadratic spline is then fitted to these rows to accurately define the baselines of the text lines. Word detection is done by measuring gaps between these baselines, allowing for precise word segmentation even with variations in character sizes and spacing.

4. **Word Recognition**: Tesseract first determines if the text is fixed-pitch (equal spacing between characters) or proportional. For fixed-pitch text, characters are segmented based on spacing. For non-fixed-pitch text, Tesseract chops the word into smaller blobs and evaluates different segmentations to find the best character matches. A dictionary search is used to refine these matches, with the system iterating over the possible segmentations to improve accuracy.

5. **Character Classifier**: the character classifier operates in two stages:

   (a) **Static Classification**: Uses a pre-trained model with feature vectors derived from the polygonal approximation of character outlines. This method is computationally intensive but effective in handling damaged images.

   (b) **Adaptive Classification**: enhances recognition by training on the document itself during the first pass. It uses the same features as static classification but adapts to the specific fonts and styles in the document, allowing for better accuracy in subsequent passes.

The Tesseract OCR engine, particularly in its more recent versions (starting from version 4.0), uses neural networks extensively as part of its text recognition process:

- **CNNs** are used in Tesseract for feature extraction from the input image. The convolutional layers scan the image for various features like edges, textures, and shapes, which are crucial for identifying individual characters. These features are then passed on to other parts of the network for further processing. The CNN in Tesseract operates on the image at multiple scales and orientations, which allows it to capture a robust representation of the characters, making it effective even for distorted or low-resolution text [40].

- Since text recognition often involves understanding the context in which a character or word appears, **LSTMs** are well-suited for this task due to their ability to maintain information over long sequences. In Tesseract, LSTMs help in recognizing text by considering both the spatial layout of characters and the context provided by surrounding characters. This sequential processing enables the engine to handle variable-length sequences, making it more accurate in recognizing entire words or lines of text rather than just isolated characters.

- After feature extraction and sequence processing, Tesseract uses **fully connected layers** to perform the final classification of the recognized text. These layers map the features extracted by the CNNs and processed by the LSTMs into specific character classes or word outputs. The FCNs are crucial for the final decision-making process in the recognition pipeline, determining what character or word each sequence corresponds to based on the data processed by the previous layers.

These neural networks work together within Tesseract's architecture to provide a more accurate and flexible OCR solution, capable of handling various challenges like different fonts and sizes.

3.3.4 Pre-processing Steps for Improved Results

The OCR Module used in the thesis project employes applies some pre-processing steps in order to optimize the text recognition process.

Image Resizing

Initially, an optional resizing operation is performed. According to the official Tesseract GitHub repository documentation [41], to achieve optimal results, the x-height of approximately 20 pixels is recommended. The x-height refers to the height of the 'x' character in the image. Additionally, a minimum x-height of 10 pixels is advised to prevent information loss during noise reduction [41].

To ensure adequate image dimensions, the image width and height must meet or exceed a minimum size threshold. If the dimensions are below this threshold, the image is resized to meet the requirement while maintaining the original aspect ratio.

Determining the appropriate resizing dimensions for input images is challenging due to the variability in font types and sizes, particularly in UI screenshots and mockups. Therefore, an empirical method was used. This method involved testing a set of reference images that represent typical inputs for the application, with increasing size thresholds. The goal was to identify the minimum size threshold that would ensure reliable text recognition.

The experiment involved 10 reference images, each processed with the OCR procedure at increasing threshold values, ranging from 2000 to 5000 pixels. The OCR results were classified as either correct or incorrect based on whether all text elements were accurately recognized. The results are summarized in Table 3.1.

The findings indicate that a size threshold of 3000 pixels is optimal, as it minimizes the number of incorrect text detections. Therefore, this threshold was selected as the final choice.

Size Threshold (px)	Percentage of Correct Results
2000	50%
3000	70%
4000	60%
5000	60%

Table 3.1: Summary of results of the resize threshold experiment.

Color Space Conversion

Another preprocessing step applied to the optionally resized input image is the conversion from BGR to grayscale. This conversion impacts the initial adaptive thresholding procedure and subsequently affects all following processing steps.

Initial experiments showed that preserving the BGR color space and then converting to grayscale yielded different but complementary results. Consequently, the OCR process was performed on both the grayscale-converted image and the original BGR image.

This approach generates two distinct result sets: one from the BGR image and one from the grayscale image. Each set includes text strings and associated bounding rectangles indicating text locations. To consolidate these results, the BGR-derived set is filtered to exclude text elements where the bounding rectangles overlap with those from the grayscale-derived set. The final result set is created by merging the filtered BGR results with the grayscale results.

3.4 Post-processing Techniques

A series of post-processing steps are implemented to enhance the quality of results from the Edge Detection and Optical Character Recognition processes. Initially, pre-processing is conducted separately on the two result sets. Once combined into a single comprehensive list of detected graphical and textual elements, further adjustments are made. The goal of these modifications is to eliminate false detections caused by image imperfections and noise and to apply heuristics to maximize the extraction of structural information from the data.

3.4.1 Detected Edges Post-processing

Following the application of the necessary pre-processing steps and the Canny Edge Detection Algorithm, a binarized image highlighting the detected edges is produced. However, this outcome does not yet meet the desired objective, which is

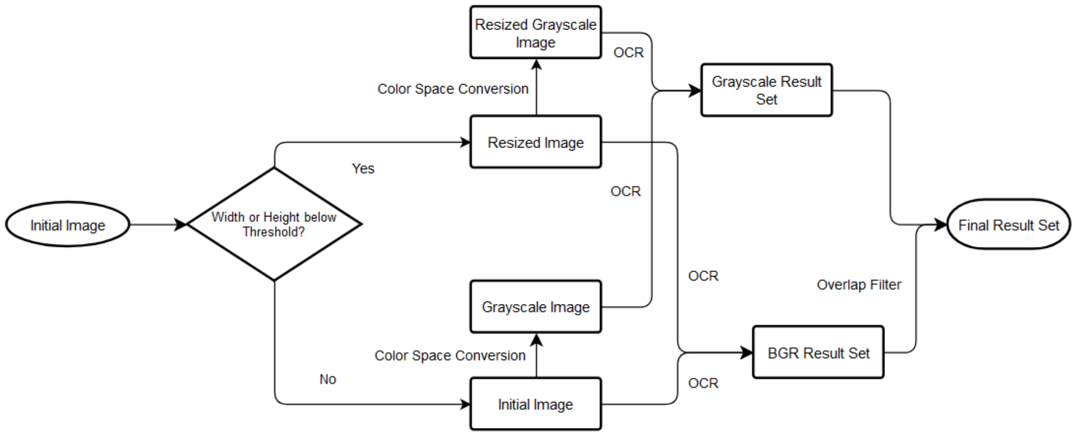


Figure 3.6: Overview of the input image OCR process employed in the application, including the related pre-processing steps.

to generate a list of detected graphical elements represented by bounding rectangles that indicate their locations within the input image. To achieve this goal, two additional operations are performed on the edge map:

1. A Contour Detection Algorithm is applied to the edge map to identify a list of graphical shapes from the edges.

2. The identified contours are then approximated to their corresponding bounding rectangles.

Each of these procedures will be examined in greater detail in the following sections.

From Edges to Contours

A contour is defined as a finite list of points within the image two-dimensional space which identifies an enclosed shape. Contour Detection algorithms allow to extract these lists from the binarized images obtained from Edge Detection procedures. Additionally, it is possible to establish hierarchical relationships between the detected contours, i.e. *parent* contours can enclose one or more *child* contours.

The article “Topological Structural Analysis of Digitized Binary Images by Border Following”, authored by Satoshi Suzuki and Keiichi Abe in 1985 [42], presents an efficient algorithm for Contour Detection in binary images, widely known

as **the Suzuki-Abe algorithm**. It is widely used in the field of image processing and is the algorithm of choice for the presented thesis project. The algorithm operates directly on binary images, which are composed of pixels represented as either foreground (typically denoted by “1” or “true”) or background (denoted by “0” or “false”); this makes it suitable for the analyzed use case, where the input is represented by the binary representation of detected edges.

At the heart of the Suzuki-Abe algorithm is a border-following method that systematically traces the contours of objects within the binary image. The procedure is initiated by scanning the image from the top-left to the bottom-right. Upon encountering an unvisited foreground pixel, the algorithm begins the contour-following process.

This contour tracing is executed in a clockwise direction by evaluating the neighboring pixels in a predetermined order to identify the next contour pixel. As the contour is traced, the coordinates of the contour pixels are recorded, and these pixels are marked to prevent them from being processed again. The process continues until the contour is closed, returning to the starting pixel. Internal contours within the object are detected and recorded in a similar manner [42].

The algorithm also constructs a hierarchical representation of the detected contours, often referred to as a contour tree. In this hierarchical structure external contours act as **parent nodes**, while internal contours, representing holes within objects, serve as **child nodes**. An example of such hierarchical structure is provided in Figure 3.7.

The Suzuki-Abe algorithm is highly efficient in both time and space. By ensuring that each pixel is processed only once during the border-following procedure, the algorithm achieves a time complexity that is linear with respect to the number of pixels in the image [42]. This efficiency makes it well-suited for real-time applications and the processing of large images.

By applying the algorithm to the edge map generated from the earlier edge detection process, a list of all contours within the image is obtained. These contours represent potential regions of interest that may correspond to graphical UI elements within the image.

Approximation and Merging of Detected Contours

Contours can be composed of a large number of points and represent shapes of arbitrary complexity; this representation is not well-suited for our application, in which we are only interested in the extraction of rectangular shapes for later analysis. For this purpose, each detected contour is approximated to its corresponding bounding rectangle, which is easily extracted by simply identifying the minimum and the maximum x and y coordinates from the corresponding list of points.

At this point, a complete list of bounding rectangles is obtained. This will

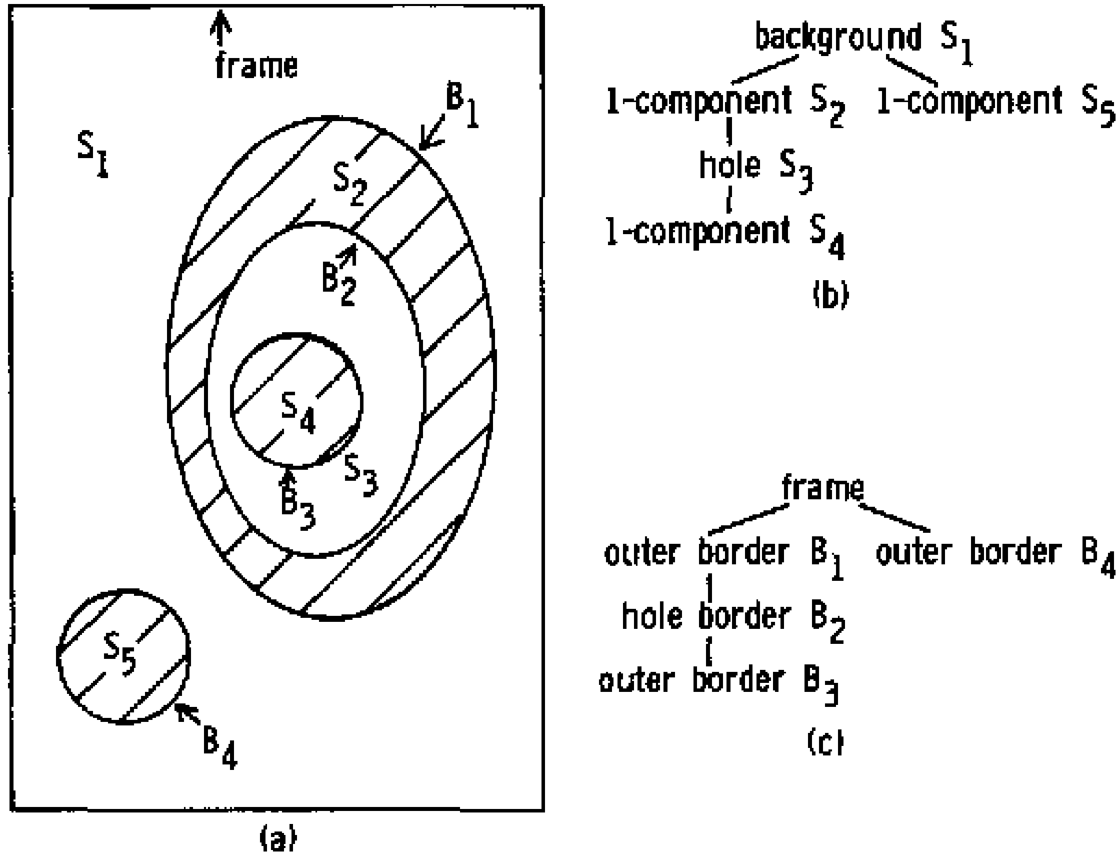


Figure 3.7: Example of contour/border detection within an image using the Suzuki-Abe Algorithm, from [42].

include nested rectangles, and partially overlapping ones. The final goal of the post-processing procedure is to obtain the set of outermost rectangles detected within the image. Furthermore, an outer bounding rectangle may potentially contain a child rectangle, which encloses all the rectangles that fall within the outer one. This information is crucial to obtain, as it is fundamental to allow proper processing of so-called **container** elements (i.e. elements that recursively contain other elements).

To obtain these results, a dedicated algorithm is employed. The goal of this algorithm is to extract the outermost rectangles in an image, which represent potential user interface (UI) elements, while also identifying any child rectangles that these external rectangles may enclose.

The algorithm operates in two main phases:

1. **Determining the smallest parent rectangle**: for each rectangle in the set, the algorithm identifies the smallest rectangle that completely encloses

it, referred to as the *parent* rectangle. If no such parent exists, the rectangle is considered external. The parent is selected based on the smallest area among all enclosing rectangles, ensuring that the hierarchy of containment is accurately represented.

2. **Identifying external rectangles and their children**: Once the parent-child relationships are established, the algorithm proceeds to identify **external** rectangles - those rectangles that do not have any parent. These external rectangles are crucial as they represent the top-level elements within the image. For each external rectangle, the algorithm then determines its **child** rectangle, which is defined as the smallest rectangle that encloses all other rectangles within the external rectangle. The child rectangle is computed through the following steps:

    - Initially, the child rectangle is unassigned.
    - The first rectangle identified as a child (i.e., one that has the external rectangle as its parent) is assigned as the child.
    - Any subsequent child rectangles are merged with the existing child rectangle, expanding its area to encompass all enclosed rectangles.

By the end of this process, the algorithm produces a set of external rectangles that potentially correspond to distinct UI elements within the image. Each external rectangle may also have a corresponding child rectangle, which encapsulates all the nested rectangles within it. This hierarchical structure is essential for accurately representing container elements.

Following this procedure, overlapping external rectangles may be present. Such overlaps typically indicate inaccuracies in the edge or contour detection process, as UI elements are expected to be distinct and non-overlapping. To address this issue, a further post-processing algorithm is employed to eliminate these overlaps.

The post-processing algorithm operates by iterating over the list of external rectangles. For each rectangle, the algorithm checks for overlaps with all other rectangles in the list. If an overlap is detected, the overlapping rectangles are merged into a single rectangle that encompasses the area of both.

In cases where two external rectangles are merged, their respective child rectangles are handled as follows:

- If neither of the merged rectangles has a child, the resulting merged rectangle will also have no child.

- If only one of the merged rectangles has a child, that child rectangle is assigned to the merged rectangle.

- If both merged rectangles have child rectangles, the child with the larger area is selected as the child of the merged rectangle.

Upon completion of these post-processing steps, the algorithm produces a final set of UI element rectangles, each with its corresponding child rectangle if applicable. This refined list is optimized for use in subsequent stages of the image analysis process.

3.4.2 OCR Results Post-processing

The OCR results undergo a sequence of refinement steps designed to enhance the accuracy and coherence of the detected text. Specifically, the post-processing procedures aim to group spatially related textual elements, facilitating the merging of individual words into complete lines of text, and further merging these lines into coherent paragraphs. This process relies on heuristics that consider the spatial arrangement and properties of the detected text elements.

The post-processing procedure applied to text elements consists of two primary phases:

1. Spatially related text elements are merged using dedicated algorithms to form complete lines and paragraphs.

2. The bounding rectangles of the resulting text elements are subsequently scaled down, adjusting for the optional resizing operation performed during the pre-processing steps for OCR.

Merging Related Elements

The merging of spatially-related textual elements is conducted in two sequential steps. The first step focuses on combining horizontally grouped elements, such as individual words, while the second step targets the merging of vertically aligned text elements. The goal of this process is to produce a list of text elements that represent either individual words, short sentences, or entire paragraphs.

The algorithm for merging horizontally grouped words operates by analyzing pairs of text elements and evaluating them against two criteria:

- The text elements must overlap along the vertical axis.

- The horizontal distance between the text elements must be less than or equal to 75% of the height of the taller element.

When both conditions are satisfied, the two text elements are merged into a single element. The bounding rectangle of the new merged element is defined as the minimal rectangle that encompasses both original rectangles. The textual content

of the merged element is derived by concatenating the content of the original elements with a space in between, maintaining the order based on their horizontal positioning.

Following the initial procedure for horizontally grouping text elements, a second algorithm is employed to merge vertically grouped text elements. This algorithm follows a similar methodology to the previously described one, evaluating pairs of text elements against specific criteria to determine whether they should be combined into a single entity. The conditions checked are as follows:

- **X-Similarity**: The difference in the x-positions of the two bounding rectangles is less than or equal to 10% of the maximum width of the two elements.
- **Height Similarity**: The difference in height between the two bounding rectangles is less than or equal to 20% of the maximum height of the two elements.
- **Vertical Proximity**: The vertical distance between the two bounding rectangles is less than or equal to 20% of the maximum height of the two elements.

After evaluating these conditions for each pair of text elements, the algorithm performs a broader grouping operation using a transitive approach. For example, if text elements A and B satisfy all three criteria, and B and C also satisfy them, then A, B, and C will be grouped together.

Once the vertical grouping is completed, the grouped elements are merged similarly to the horizontal merging process. The resulting text element is defined by a bounding box that encompasses all the original rectangles. The textual content is obtained by concatenating the text from each element in the group, separated by new lines and ordered according to their vertical positions.

3.4.3 Edge Detection and OCR Results Integration and Post-processing

After applying the necessary post-processing steps to the results from both Edge Detection and OCR, the outputs are merged into a unified and comprehensive result set. The integration process addresses two key challenges:

1. The differing formats of the elements in the two result sets.
2. The potential duplication of textual content detected by both the OCR and Edge Detection processes.

To standardize the format, elements from both sources are converted into a common data structure that includes the following information:

- The coordinates of the top-left and bottom-right corners that define the bounding rectangle of the element.
- The type of the element. Elements originating from the Edge Detection process are initially assigned the type "unknown", to be refined during subsequent analysis. Elements derived from OCR are labeled as "text".
- Text content, applicable only to elements derived from OCR.

To eliminate duplication of text elements, a straightforward two-step approach is employed. First, each element detected through Edge Detection is compared with each OCR-detected element to check for overlap. Any element from the Edge Detection process that overlaps more than 75% of its area with an OCR-detected element is removed from the result set. This prioritization ensures that OCR results, which are generally more accurate for textual content, take precedence when both methods detect the same text.

In the second step, the process is repeated with the roles of the two sets of elements reversed. This step removes redundant text elements that may correspond to textual content within other graphical elements, such as input fields or buttons.

Small Elements Removal

A potential source of error in the result set is the presence of spurious elements, which may arise from erroneous detections due to imperfections or noise within the image. To mitigate this issue, a filtering operation is applied to the result set, removing elements that meet any of the following three criteria:

- The area of the element is below a specified threshold t_a.
- The width of the element is below a specified threshold t_w.
- The height of the element is below a specified threshold t_h.

The values for these thresholds were determined based on observations from testing the procedure with realistic input examples.

Size Assignment

Each element in the result set is assigned a size category, which can be "small", "medium" or "large". This classification is crucial for subsequent stages of the image analysis process.

The size classification is determined based on the height of the element:

- If the height of the element is less than or equal to a threshold t_{small}, the element is categorized as "small".

- If the height exceeds t_{small} but is less than or equal to a threshold t_{medium}, the element is categorized as “medium”.
- If the height exceeds t_{medium}, the element is categorized as “large”.

Horizontal Element Alignment

Upon completion of the aforementioned post-processing steps, the resulting list of elements provides a reliable and comprehensive representation of the regions of interest for the subsequent element analysis stage. However, an additional adjustment is required to ensure the effectiveness of later stages in the image conversion process.

The bounding rectangles for the detected elements often exhibit a degree of irregularity due to inherent margins of error, even when advanced computer vision techniques are employed. This irregularity can obscure the spatial relationships and alignments among UI elements, potentially leading to a loss of crucial information.

To address this issue, a heuristic approach is applied to refine the x-coordinates of the bounding rectangles. This involves using a set of simple rules to adjust the start (x_1) and end (x_2) positions of the bounding rectangles. Specifically, if two elements have x_1 values that differ by less than a specified threshold t_align, they are assigned the same x_1 value. The same procedure is applied to the x_2 values. The original x_1 and x_2 values are preserved as additional fields within the element's data structure, as they are required for the subsequent image cropping procedure essential for the element analysis.

3.5 Implementation Tools

In order to implement all the described image processing steps in the form of Python code, two crucial packages are used:

- **OpenCV**: an open-source computer vision and machine learning library in Python, widely used for image processing, object detection, and real-time video analysis.
- **PyTesseract**: a Python wrapper for the Tesseract-OCR engine.

These are analyzed more in depth in the following sections.

3.5.1 OpenCV: A Powerful Open-Source Computer Vision Library

OpenCV, an acronym for Open Source Computer Vision Library, is a highly versatile and widely adopted library that facilitates the development of applications in the

domains of computer vision and image processing. Originally developed by Intel in 1999, it has since evolved into an open-source project, attracting contributions from developers worldwide [43]. The library is written primarily in C and C++, but its Python bindings have gained significant popularity due to the simplicity and expressiveness of the Python programming language [44].

In the context of Python, OpenCV provides a robust interface for handling a wide array of image and video processing tasks. OpenCV's extensive functionality is complemented by its ability to integrate seamlessly with other scientific computing libraries in Python, such as NumPy [43].

One of the key strengths of OpenCV is its optimization for real-time applications. It supports hardware acceleration, making it suitable for applications requiring high performance [45]. The library also offers comprehensive documentation and an active community, which further facilitates learning and collaboration among users [44].

OpenCV provides a vast array of powerful functions that allow to easily perform various Computer Vision tasks. The following is an overview of the usage of OpenCV within the Element Detection Module of the application:

- **Image Conversion to Grayscale**: OpenCV is used to convert the input image from its original color format to grayscale.
- **Contrast Enhancement Using CLAHE**: the module applies the Contrast Limited Adaptive Histogram Equalization technique to the grayscale image using OpenCV's dedicated functions.
- **Edge Detection Using Canny Algorithm**: OpenCV's Canny function is employed to detect edges in the contrast-enhanced image.
- **Contour Detection**: a dedicated OpenCV function is used to detect the contours within the edge-detected image.

3.5.2 PyTesseract: Integrating Tesseract into the Python Ecosystem

PyTesseract is a Python wrapper for the Tesseract-OCR engine. This library allows users to apply OCR to various types of image data and extract textual information with high accuracy.

PyTesseract supports multiple languages and offers features such as image preprocessing to improve OCR results, thereby making it a versatile tool for applications ranging from document digitization to automated data extraction [40]. Its integration with Python facilitates seamless usage within data processing pipelines, leveraging other libraries such as OpenCV for image manipulation and NumPy for numerical operations.

Moreover, PyTesseract's ease of use and extensive documentation make it an attractive choice for those who need reliable text extraction capabilities without extensive configuration. The library's design emphasizes simplicity and efficiency, which aids in rapidly developing prototypes or implementing solutions where text extraction from images is a critical component.

The OCR Module of the image analysis application makes use of PyTesseract to perform the extraction of textual data and its corresponding spatial information within the image.

3.6 Results of the Element Detection Process

The final result that is obtained from the initial image analysis is a list of detected UI elements, each represented via a well-defined data structure; this element representation contains spatial information, useful to locate the element within the input image, as well as a preliminary classification, which categorizes the element either with type "unknown" or "text". It also includes the computed size The following is an example of the aforementioned data structure:

```
{
  "x": 29,
  "y": 222,
  "w": 411,
  "h": 44,
  "x2": 435,
  "y2": 266,
  "type": "unknown",
  "child": {
    "x": 26,
    "y": 224,
    "w": 407,
    "h": 40
  },
  "size": "large",
  "cropX1": 24,
  "cropX2": 435
}
```

As can be seen, the horizontal coordinates of the element are expressed in two forms:

- x and $x2$: theese coordinates represent the horizontal boundaries of the element **after** the horizontal alignment post-processing step.

- $cropX1$ and $cropX2$: these coordinates represent the horizontal boundaries **before** the post-processing step; they are useful for the later Element Analysis process, in which the image needs to be accurately cropped to extract the exact image area that corresponds to the element.

An example of the output that is obtained from the Element Detection Module is shown in Figure 3.8.

(a) Example input.

(b) Graphical representation of the detected elements. Blue rectangles are used for the "text" type, while green ones are used for "unknown".

Figure 3.8: Example of the obtained output from the Element Detection process.

Chapter 4

Analysis of UI Elements with Multi-Modal Large Language Models

The Element Detection and OCR Modules process the input image provided by the user, generating a structured list of detected elements. Each element is represented by a data structure containing the following information:

- The coordinates of the two points defining the element's bounding box within the image.
- The adjusted bounding box coordinates resulting from the Horizontal Alignment post-processing step, which are essential for the subsequent Layout Generation process.
- The coordinates of the bounding box for any child elements within the image, if present.
- The type of the element, initially classified as either "unknown" or "text".
- The size of the element, categorized as "small", "medium", or "large".

The next phase of the application involves employing a Multi-modal Large Language Model (LLM) to perform a visual analysis of each detected element, enhancing the data structure with additional information. Specifically, this step aims to accurately categorize elements initially marked as "unknown" and to append supplementary fields to the data structure. This comprehensive description of the UI element facilitates a thorough and structured representation of its content and styling.

There are multiple factors that justify the use of LLMs for this task over simpler classification neural networks:

- An LLM can generate a structured JSON representation of the UI element depicted in the image, offering more than a simple selection from predefined categories.
- LLMs enable quick and straightforward adjustments to the analysis process through prompt modifications, significantly enhancing system flexibility and simplifying change management.

The subsequent chapters will provide an overview of Multi-modal LLMs and their role in the Element Analysis Module to deliver a comprehensive and structured description of the detected UI elements.

4.1 Overview of Large Language Models (LLMs)

Large Language Models (LLMs) represent a significant advancement in the field of artificial intelligence and natural language processing. These models are designed to understand and generate human-like text based on the vast amounts of data they have been trained on.

The development of LLMs can be traced back to the early days of natural language processing (NLP). Initial models were rule-based and relied on manually crafted rules. These early systems, such as ELIZA in the 1960s, demonstrated the potential of computers to engage in simple text-based interactions [46].

The next significant advancement came with the introduction of statistical methods in the 1990s. Models such as Latent Semantic Analysis (LSA) and Hidden Markov Models (HMMs) allowed for the analysis of text based on statistical patterns rather than explicit rules [47]. These models improved the ability to handle ambiguity and variability in natural language.

The emergence of neural networks marked a transformative period in NLP. Early neural network-based models, such as Feedforward Neural Networks and Recurrent Neural Networks, began to outperform traditional methods in various NLP tasks [48]. However, it was the introduction of LSTM networks that significantly advanced the capabilities of neural models in handling sequential data [33].

The advent of Transformer models in 2017 marked a pivotal moment in the evolution of LLMs. The Transformer architecture, introduced by Vaswani et al., leveraged attention mechanisms to handle long-range dependencies in text more effectively than previous models [49]. This architecture enabled the development of more sophisticated models capable of understanding and generating text with greater coherence and accuracy.

4.1.1 The Transformer Architecture

The Transformer architecture, introduced in "Attention Is All You Need" by Vaswani et al. [49], has revolutionized the field of natural language processing. Unlike RNNs and CNNs, which process input sequentially, the Transformer leverages a novel attention mechanism to capture dependencies between input elements simultaneously. This enables the model to effectively handle long-range dependencies and achieve state-of-the-art performance on various NLP tasks.

Encoder-Decoder Structure

The Transformer architecture consists of two primary components: the encoder and the decoder [49]. A graphical representation of such structure is available in Figure 4.1.

The encoder processes the input sequence, transforming it into a meaningful representation. It comprises multiple identical layers, each containing two sub-layers:

- **Multi-Head Self-Attention**: this layer calculates attention weights for each input token relative to all other tokens in the sequence. Multiple attention heads are employed to capture different dependencies. The output of this layer is a weighted sum of the input tokens, where the weights reflect the importance of each token for understanding the current token.

- **Position-wise Feed-Forward Network**: This layer applies the same feed-forward neural network to each position of the input sequence independently. It consists of two linear transformations with a non-linear activation function in between.

The decoder generates the output sequence, conditioned on the encoder's output. It also consists of multiple identical layers, each containing three sub-layers:

- **Masked Multi-Head Self-Attention**: similar to the encoder's self-attention, but with a mask to prevent the model from attending to future tokens during training.

- **Encoder-Decoder Attention**: this layer calculates attention weights between the output of the decoder and that of the encoder. This allows the decoder to focus on relevant parts of the input sequence while generating the output.

- **Position-wise Feed-Forward Network**: identical to the encoder's feed-forward network.

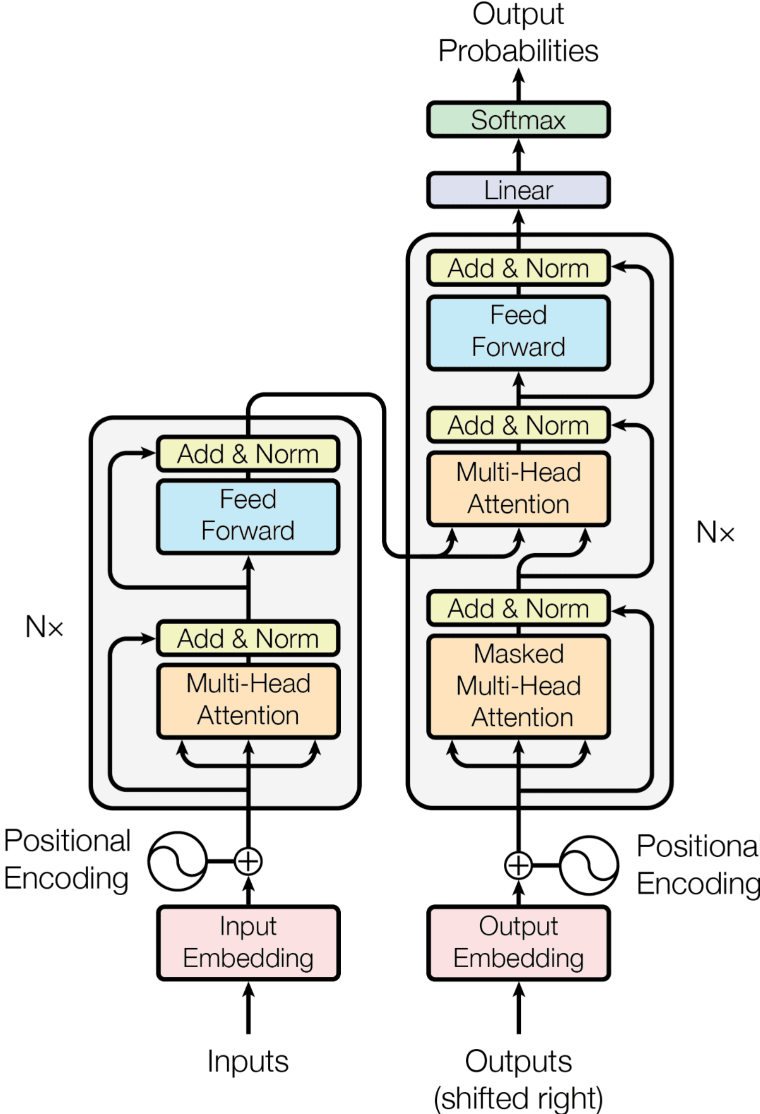


Figure 4.1: The encoder-decoder structure of the Transformer architecture, from [49].

Attention Mechanism

The attention mechanism is the core component of the Transformer. It allows the model to weigh the importance of different parts of the input sequence when processing a specific position. The calculation involves three steps:

1. **Query, Key, and Value Vector Projection**: each input token is projected

into three vectors - *query*, *key*, and *value*.

2. **Attention Score**: the attention score is calculated by taking the dot product of the query vector of the current position with the key vectors of all positions. These scores are then normalized using the *softmax* function to obtain attention weights.

3. **Weighted Sum**: the value vectors are multiplied by the attention weights and summed to produce the output of the attention layer.

Positional Encoding

Since the Transformer does not inherently consider the order of input tokens, positional encoding is added to the input embeddings to provide information about the relative and absolute positions of tokens in the sequence. This is crucial for capturing sequential dependencies.

Layer Normalization and Residual Connections

To improve training stability and performance, the Transformer incorporates layer normalization and residual connections in each layer.

- **Layer Normalization**: this technique normalizes the inputs of each layer to have zero mean and unit variance, independent of the batch size. It helps to stabilize training and accelerate convergence.

- **Residual Connection**s: these connections allow the model to learn identity mappings, enabling the gradient to flow more easily through the network. They help to alleviate the vanishing gradient problem and facilitate training of deeper models.

The Transformer architecture has demonstrated remarkable success in various NLP tasks, including machine translation, text summarization, and question answering. Its ability to capture long-range dependencies and process input in parallel has contributed to its widespread adoption and continued development.

4.1.2 The Concept of Multi-Modality in LLMs

The concept of multimodality in Large Language Models represents a significant evolution in artificial intelligence, allowing these models to process and generate content across multiple data types, such as text, images, audio, and video. This advancement stems from the foundational capabilities of LLMs, which have primarily focused on text processing. The development of Multimodal Large Language

Models (MLLMs) extends these capabilities, offering a more holistic approach to data interaction and interpretation, particularly relevant in fields that require the integration of diverse data formats [50].

MLLMs build upon the principles of multimedia learning theory, which posits that combining multiple modalities—such as text and images—can enhance knowledge acquisition by enabling the integration of information into a more coherent and multi-faceted mental model. This approach is grounded in cognitive science, where the processing of different forms of information through various channels enhances learning effectiveness [50]. For instance, in science education, the integration of textual descriptions with visual aids like diagrams or models allows for a more comprehensive understanding of complex concepts, as learners can process and relate information more effectively when it is presented through multiple sensory modalities.

Multimodal Large Language Models (MLLMs) are capable of processing inputs in various formats—such as text, images, and audio—through advanced architectures that integrate multiple neural networks, each specialized in handling different data types. Central to this capability is the use of encoders and decoders, which transform raw input data into a shared latent space, enabling the model to effectively combine and interpret information across different modalities. For example, when processing an image, a convolutional neural network (CNN) typically serves as the encoder, extracting visual features and converting them into a vector representation that the model can understand. This vector is mapped into the same latent space as text data, which might be processed by a transformer-based network, thereby allowing the model to seamlessly integrate information across modalities [51].

The training process of MLLMs involves large-scale datasets that include paired examples of different modalities, such as images with corresponding captions or videos with associated audio descriptions. Through exposure to these multimodal datasets, the model learns to establish correspondences between modalities, enabling it to generate structured outputs from varied inputs. For instance, by training on both text describing an object and images of that object, the model learns the underlying associations between visual and textual representations. This multimodal learning process ensures that the model can generalize across formats, effectively processing and generating outputs even with new combinations of input data. The use of attention mechanisms within the transformer architecture further enhances the model's ability to focus on the most relevant aspects of each modality, improving its capacity to extract and synthesize information from diverse inputs [52].

A Multimodal Large Language Model can be effectively utilized to extract structured information from an input image by converting the visual data into a predefined format, such as JSON. This process involves the model interpreting the image to identify and categorize relevant elements, such as objects, text, or patterns,

and then mapping these elements into structured data fields. For instance, an image of a business card could be processed by the MLLM to extract information such as name, contact details, and company, organizing these details into a JSON object with keys corresponding to each type of information. This capability is particularly valuable in scenarios requiring the automation of data extraction from visual inputs, enabling efficient data processing and integration with other systems, while maintaining a high level of accuracy and consistency in the interpretation of complex visual data [50].

4.2 Analysis of UI Elements

Leveraging the advanced capabilities of state-of-the-art MLLMs, this application follows a structured workflow to analyze graphical elements detected within an image:

- The data structure for each detected element includes the coordinates of two points that define its bounding rectangle. These coordinates are used to crop the region of interest from the original image.

- The cropped image of the element is then processed by an MLLM, which is tasked with generating a JSON output that provides a standardized description of the image content and its properties.

- The relevant information from the output generated by the MLLM is extracted and integrated with the existing data about the element.

A single image may contain numerous detected elements, potentially resulting in a high number of generation tasks. This can lead to increased costs and processing time. To address these challenges, various optimization strategies can be implemented. The overall procedure and the adopted optimization techniques will be discussed in detail in the following sections.

4.2.1 Using GPT-4o for Standardized UI Element Descriptions

A crucial consideration in the design of the Element Analysis Module was the selection of the appropriate Large Language Model. Recently, several leading providers have developed and released advanced LLMs, available either through API-based services or as open models that can be independently deployed and managed.

- **OpenAI** is a leading entity in the field of LLMs, renowned for its GPT (Generative Pre-trained Transformer) series. The latest model, GPT-4, is offered in various configurations, including a multimodal version capable of processing both text and images. It demonstrates enhanced reasoning, contextual understanding, and complex problem-solving abilities compared to its predecessors. OpenAI provides access to these models via the OpenAI API [4].

- **Google**'s DeepMind division has introduced the Gemini series (formerly known as Bard), positioned as a competitor to OpenAI's GPT models. The most recent version, Gemini 1.5, exhibits improved performance across various tasks, particularly in reasoning, code comprehension, and specialized knowledge areas [5].

- **Anthropic** specializes in the development of LLMs with a strong focus on safety and alignment with human values. Their latest model, Claude 3.5 Sonnet, exemplifies this approach [53].

- **Meta** has developed the LLaMA (Large Language Model Meta AI) series as part of an open research initiative aimed at democratizing access to LLMs. LLaMA 3, the most recent release, offers significant advancements in efficiency and performance, making it accessible for a broad spectrum of applications [54].

- **Mistral** is a newer entrant in the LLM field, with a focus on developing lightweight and efficient models. Their Mistral 7B model is engineered to be compact yet powerful, making it ideal for applications that require lower computational resources [55].

After considering the available options, it was decided to use **GPT-4o**, the latest flagship model released by OpenAI as part of the GPT-4 model series, as the model of choice for the implemented application. GPT-4o has demonstrated significant improvements in accuracy across a variety of tasks, particularly in vision-related applications. According to the comprehensive evaluation conducted by Shahriar et al. [56], GPT-4o excels in image classification and object recognition tasks, showcasing its ability to accurately interpret and process visual data. This high level of accuracy is crucial for vision-based applications where precision is paramount, such as the covered project.

One of the key benchmarks where GPT-4o's accuracy stands out is its performance in standardized vision tasks, which assess the model's capacity to correctly classify and interpret complex visual inputs. For example, in comparison to its predecessors and contemporary models like Google's Gemini and Anthropic's Claude

3, GPT-4o consistently performs at a high level, with fewer errors and greater consistency in results. This accuracy not only enhances the reliability of vision-based applications but also reduces the need for extensive human oversight, making it a more autonomous and efficient choice [56].

In addition to its accuracy, GPT-4o is recognized as one of the fastest LLMs available for advanced applications. The model's design prioritizes efficiency without compromising on performance, allowing it to process and generate responses more quickly than other models. This speed is particularly beneficial in vision-based applications where real-time processing is often required. For instance, in applications such as real-time video analysis or interactive robotics, the rapid processing capability of GPT-4o ensures that visual data can be interpreted and acted upon almost instantaneously, which is critical for maintaining the functionality and safety of these systems.

Moreover, the speed of GPT-4o does not detract from its accuracy. The model manages to achieve a balance between fast processing times and high precision, making it suitable for a wide range of vision-based applications. This efficiency, combined with its advanced vision capabilities, positions GPT-4o as a leading model for applications that demand both high accuracy and fast response times [56].

GPT-4o is available through an API-based service provided by OpenAI. This simplifies the process of integrating the generation tasks within the application workflow.

4.3 Standardizing the Description of UI Elements

The UI element analysis process involves creating a standardized, structured representation of each element, facilitating subsequent translation into front-end code. Establishing a consistent format for these descriptions is crucial for efficient code generation.

The OpenAI API offers the capability to generate output in JSON format [57], which is essential for this research. This standardized output structure enables systematic extraction of relevant information.

The API also permits adjustment of the temperature parameter during generation [57], which controls the randomness of the text output. Higher temperatures increase output diversity, while lower temperatures enhance predictability. For this study, the lowest possible temperature is employed to ensure reliable and reproducible results.

The aforementioned OpenAI API is the key tool used in the Element Analysis process:

- A **multimodal input** is provided, consisting in the cropped image that

contains the UI element to analyze, and a textual prompt that instructs the model to provide the standardized description and the desired output format.

- A **textual output**, corresponding to the JSON description of the element, is obtained.

The input image is a cropped section of the original image, encompassing the target element with a small border for context. The accompanying textual prompt is critical for determining the format and quality of the output. A range of prompt engineering techniques is applied to maximize the quality of results.

In regards to the provided textual prompt, two seperate versions are used, depending on the type of element that is being analyzed:

- If the element is currently of type “unknown”, a generic prompt is provided, with the objective of assigning a type to the element and extracting all its relevant information.
- If the element is of type “text”, a prompt dedicated to the extraction of the styling features of the text element is used instead. This shorter and simpler prompt takes advantage of the already-available type information, and allows to enrich the description of the text element with the usage of fewer input tokens.

4.3.1 Prompt Engineering Techniques for Higher Quality Generation

Prompt Engineering is the iterative process of designing, refining, and implementing prompts or instructions that guide the output of LLMs to optimize their performance on various tasks. It involves modifying or changing the prompting techniques used to elicit desired responses from the model, often through both automated methods and manual adjustments by users [58].

For the Element Analysis process, it was decided to adopt a **few-shot** prompting technique. Few-shot prompting utilizes a few labeled examples from a training dataset to inform the model about the desired output format or content. The prompt template is constructed to include these exemplars, which serve as a reference for the model during inference [58].

Few-shot prompting offers several advantages. First, it improves model performance on specific tasks by providing illustrative examples. This approach typically surpasses zero-shot prompting, which relies solely on the model’s general knowledge. Second, few-shot prompting is adaptable as prompts can be generated based on the data at hand, making it suitable for various applications. Lastly, it is particularly useful when training data is limited, as the model can effectively learn from a small number of examples [58].

In the **Generic Element Analysis** prompt, a long list of examples is provided. The examples are organized in the following form:

- A textual description of the UI element that in analyzed in the example.
- The corresponding JSON description of the element.

Another possibility could be to provide examples with fully-fledged image inputs and the corresponding outputs; however, this approach would dramatically increase the cost of inference, as it would greatly inflate the number of input tokens.

The list of examples is chosen in order to cover all the possible element types that should be recognized by the model; a total of **17 examples** are provided, to provide the LLM with all the necessary information to correctly carry out the description task.

The **Text Element Analysis** prompt makes use of the one-shot prompting technique; this is a simplified approach compared to the few-shot one, where a single example is provided. This is done due to the simpler nature of the task and the smaller scope that it is defined within [58].

4.3.2 Definition of the Scope

The Element Analysis process aims to describe the detected UI elements comprehensively, capturing all relevant characteristics necessary for accurate code reproduction. The detail level of this description should be maximized to ensure precision. However, the extent of information utilized for translation may vary depending on the target platform.

Nevertheless, increasing the description's detail beyond a certain threshold presents significant drawbacks. First, highly detailed descriptions may exceed the processing capabilities of MLLMs, especially under the constraints of few-shot prompting. Second, more complex prompts and responses result in increased costs due to the higher token count in input and output.

Given these considerations, it is essential to establish an optimal scope for the analysis procedure that balances precision and efficiency. This process begins by identifying the different types of UI elements the model is expected to recognize. Subsequently, specific content and style-related properties are selected for extraction for each element type. The final version of the Generic Element Analysis prompt includes the following recognizable UI element types:

- Text
- Icon
- Button

- Button Group
- Text Input
- Dropdown Input
- Date Input
- Search Bar
- Radio Button
- Check Button
- Table
- Image
- Container
- Nothing

The "Container" type is applied when the model detects multiple UI elements of different types within the input cropped image. Accurate identification of this element type is crucial, as the application must handle cases where a container encapsulates a subset of UI elements within the image.

The "Nothing" type is designated for instances of erroneous detections resulting in empty input images. When no UI elements are identified, the model is instructed to use this classification.

4.3.3 Adopted Description Format

As previously mentioned, the format chosen for element description is JSON. Below is an example of the output generated for a Button-type element:

```
{
    "type": "button",
    "icon": "plus",
    "text": "Add",
    "textColor": "white",
    "backgroundColor": "blue",
    "borderColor": "red"
}
```

The “type” property, which is included in every description generated by the model, serves to identify the specific kind of UI element being described. This property is followed by a list of additional properties that vary depending on the element type.

Several properties are common across multiple element types. For instance, the “textColor” and “backgroundColor” properties are also found in the descriptions of other elements, such as Text. Additionally, some properties are optional; for example, a Button element might not include a “borderColor” property.

4.4 Overall Analysis Process

The Element Analysis process for each detected element is structured into the following steps:

1. The coordinates defining the bounding rectangle of the element are used to crop the corresponding region from the original input image. A small offset is applied to these coordinates to obtain a slightly enlarged rectangle during cropping. This adjustment enhances the graphical representation of the element, improving the quality of the output.

2. The appropriate Element Analysis prompt is selected based on the element type. For “text”-type elements, the Text Element Analysis prompt is employed, whereas the Generic Element Analysis prompt is used for elements classified as “unknown”.

3. The selected prompt and the cropped image are then provided as input to the MLLM, which generates a suitable description. The output is subsequently parsed into a JSON object.

4. The information contained in the generated description is combined with the previously available data for the element (such as bounding rectangle, size, etc.) to produce the final element representation, which will be utilized in later stages of the application processing.

An overview of the Element Analysis process is illustrated in Figure 4.2.

4.4.1 Organizing the Inference Process for Fast Image Analysis

When analyzing an input image, a typically large number of detected UI elements must be processed through inference via the available Large Language Model (LLM) API. This API-based interaction with the LLM constitutes a significant performance bottleneck in the overall application, due to two primary factors:

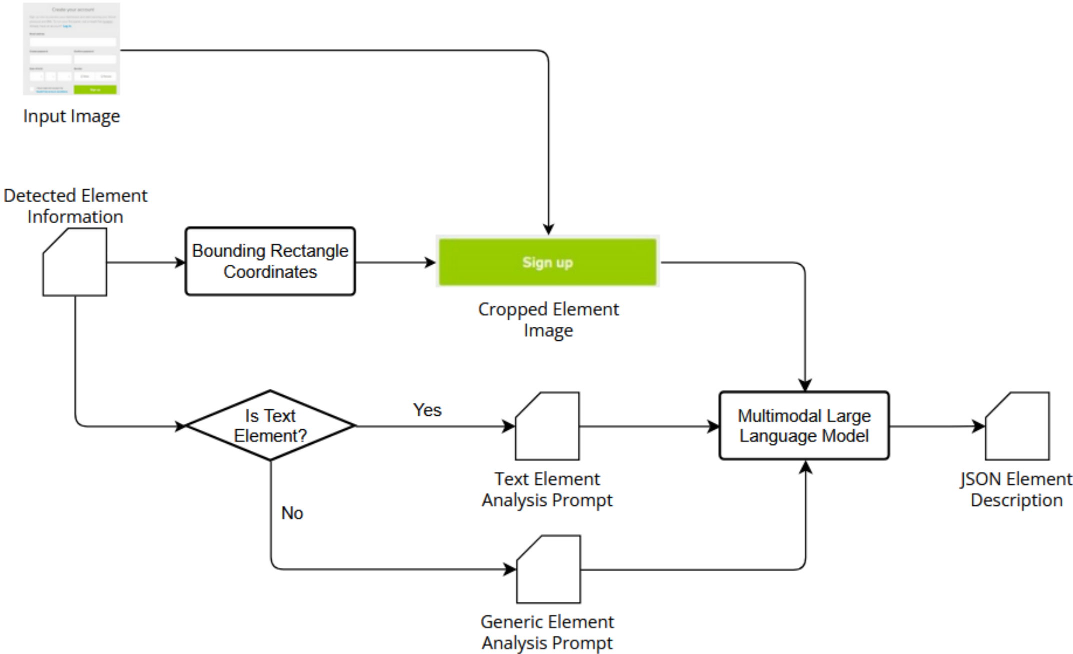


Figure 4.2: The Element Analysis process.

1. Large Language Models often require considerable time to generate responses, particularly when presented with lengthy and complex inputs. Although GPT-4o is notable for its inference speed, it still introduces a noticeable delay before a response is available.

2. Interaction via HTTP consistently incurs additional time costs associated with the request's Round Trip Time.

Given these considerations, it is crucial to minimize the time required for the analysis process to prevent disruptions in the user experience and avoid slowing down the development workflow.

To achieve this, it is advantageous to recognize that each UI element within the input image can be processed independently of the others. The Element Analysis process relies solely on information from the individual element being analyzed and is not influenced by the analysis of other elements. Consequently, all detected elements can be analyzed in parallel.

The Element Analysis Module utilizes Python's high-level concurrency framework to achieve parallel execution of tasks. By managing a pool of threads, the framework allows multiple tasks to be processed simultaneously. This is particularly effective when the tasks are I/O-bound, such as those involving network communication or file system interactions.

The core mechanism relies on distributing individual tasks across multiple threads within a thread pool. Each thread in this pool operates independently, processing its assigned task in parallel with others. This parallelism is orchestrated by a mapping function, which systematically assigns tasks to threads and ensures that all tasks are processed concurrently. The result is a significant reduction in overall execution time, as multiple tasks are completed simultaneously rather than sequentially. In this specific case, the assigned task corresponds to the Element Analysis procedure, which applies the necessary steps based on the element's properties.

4.4.2 Optimizing Costs with Design Decisions and Trade-offs

One of the biggest drawbacks of the approach that is adopted for the described application is the large number of API requests that need to be made in order to carry out the analysis process for all the detected elements within the image.

The OpenAI API follows a pricing model that is based on the number of input and output tokens for each request [57]; in the Element Analysis procedure, in addition to the image, either the full Generic Element Analysis prompt or the Text Element Analysis prompt needs to be sent as input. This substantially increases the number of input tokens. The number of output tokens for each request is generally limited, due to the length-constrained JSON format that is adopted.

It is thus in the interest of the company to try to minimize the number of requests that are made by the Element Analysis Module. For this purpose, the following optimizations are applied:

- The images are provided through the API with a low detail level; this reduces the number of input tokens without compromising the quality of the results.
- Only a subset of the available text elements is analyzed, to obtain general styling information about text with different sizes.

Image Detail Level

The OpenAI API allows the provision of inputs of different types; for image-type inputs, it is possible to choose between *high* and *low* detail level. The "detail" setting within the OpenAI API is a crucial parameter that influences how the model interprets and processes visual inputs. It offers three distinct options - *low*, *high*, and *auto* - that determine the resolution and depth of the model's image analysis. By default, the *auto* setting is employed, allowing the model to assess the input image size and autonomously decide whether to use the *low* or *high* setting

[57]. This flexibility ensures that the model can dynamically adjust its processing approach based on the specific characteristics of the input image.

In scenarios where efficiency and speed are prioritized over detailed image analysis, the *low* setting is particularly advantageous. When this option is selected, the model receives a downscaled version of the image at 512px by 512px, allocating a budget of 85 tokens to represent the image. This approach significantly reduces the input tokens required, enabling faster response times and conserving computational resources [57]. Consequently, the *low* setting is ideal for applications where high-resolution image processing is unnecessary, allowing for a more efficient use of the API while still providing sufficient detail for the Element Analysis task, in which such setting is used.

Optimized Text Elements Analysis

To reduce the number of API requests made during the Element Analysis process, it is advantageous to leverage the repetitive nature of textual elements typically found within input images. During the post-processing phase, each detected element is assigned a "size" property, classifying it as either *small*, *medium*, or *large*. For most input images, which are expected to represent user interfaces, it is reasonable to assume that text elements of similar sizes will exhibit consistent styling features, such as font color.

Given this assumption, only a select subset of text elements is chosen for the Element Analysis process. Specifically, a "text"-type element is extracted from each size category present within the input image. The analysis is then applied exclusively to this subset. The styling information obtained from these representative elements is stored as a *general style* directive corresponding to each text size. Subsequently, each text element can be associated with styling information derived from two potential sources:

- Element-specific styling information, which may be present if the element has been individually analyzed.
- The *general style* information, obtained from the analysis of the representative subset of text elements.

During the code translation process, these sources are checked sequentially; if element-specific information is not available, the *general style* is applied.

This optimization strategy, while effective in reducing API calls, results in a slight loss of precision in the representation of image content. To mitigate this limitation, the strategy is implemented as an optional feature, allowing users to decide whether to utilize it. Nonetheless, it is enabled by default due to its significant benefits and the minimal quality degradation observed in the final output.

4.5 Results of the Element Analysis Process

The result of the previously described process is a list of elements represented with a data structure that expands the foundation described in Section 3.6. More specifically, additional styling and content information is introduced, allowing for a more accurate representation in the final General Page Description. The following is an example of such enriched format, showing the started data structure provided by the Element Detection Module and its enriched counterpart obtained via the Element Analysis Module:

```
{
  "x": 414,
  "y": 608,
  "w": 156,
  "h": 61,
  "x2": 570,
  "y2": 669,
  "type": "unknown",
  "child": {
    "x": 437,
    "y": 631,
    "w": 109,
    "h": 15
  },
  "size": "large",
  "cropX1": 414,
  "cropX2": 570
}
```

```
{
  "x": 414,
  "y": 608,
  "w": 156,
  "h": 61,
  "x2": 570,
  "y2": 669,
  "type": "button",
  "child": {
    "x": 437,
    "y": 631,
    "w": 109,
    "h": 15
  },
  "size": "large",
  "cropX1": 414,
  "cropX2": 570,
  "text": "Submit form",
  "textColor": "white",
  "backgroundColor": "teal"
}
```

As can be seen, the previously obtained information is enriched with styling- and content-related information; the “type” property is finally assigned, and additional information such as “text”, “textColor” and “backgroundColor” is introduced.

Chapter 5

Layout Generation Process

The final step in generating the General Page Description from the input image involves structuring the resulting set of detected elements into a coherent and well-defined layout. Properly arranging these UI elements to reflect the original page layout is essential and presents a complex challenge, requiring the application of various algorithms and techniques. The primary goal of the Layout Generation process is to accurately reproduce the visual organization of the elements as they appear in the input image, while avoiding any imperfections or artifacts that may arise due to the automated nature of the procedure.

The subsequent sections will provide a detailed explanation of the rationale behind the chosen technical approaches and their implementation.

5.1 Web Layout Utilities as a Starting Point

The first step in designing the Layout Generation Module is deciding what kind of layout structure to adopt for organizing the UI elements. Given that the discussed application has front-end development for web applications in mind as its primary use case, it seems natural to take the existing web layout utilities as a starting point for such design decision.

When considering existing layout utilities for web development, the following options are considered the most robust and common:

- CSS Flexbox Module
- CSS Grid Layout Module
- Bootstrap Grid Layout Utilities

5.1.1 CSS Flexbox Module

The Flexbox CSS layout module is a framework for organizing elements within a single dimension, allowing for precise control over spacing, alignment, and distribution of items within a container. Unlike other layout systems that operate on two dimensions simultaneously, Flexbox focuses on a single axis at a time, either horizontally or vertically, making it particularly well-suited for applications where the alignment of items along one dimension is the primary concern.

At its core, Flexbox operates on two axes: the main axis and the cross axis. The main axis is determined by the direction in which the content is intended to flow, while the cross axis runs perpendicular to it. By manipulating these axes, one can control how space is distributed between items, as well as how the items are aligned within the container. The system supports both growth and shrinkage of items, enabling them to adapt dynamically to the available space, which is especially useful in responsive design scenarios [59].

One of the primary advantages of the Flexbox layout is its flexibility. It allows for the automatic distribution of available space among items, ensuring that they can expand or contract based on the container's size. This adaptability is further enhanced by properties that control alignment, such as justifying content along the main axis or aligning items along the cross axis. Such control facilitates the creation of complex layouts that remain robust across different screen sizes and orientations, making it a powerful tool in the toolkit of designers and developers [59].

However, the conceptual simplicity of Flexbox also introduces certain limitations. While it excels at organizing elements along a single axis, it may not be the most efficient solution for layouts that require simultaneous control over multiple dimensions. Additionally, the reliance on a predefined direction for the main axis can impose constraints on designs that require a more fluid and non-linear arrangement of elements. The need to manage alignment and spacing across different axes independently can also introduce complexity when dealing with more intricate layouts, potentially leading to an increased cognitive load during the design process.

5.1.2 CSS Grid Layout Module

The CSS Grid Layout Module introduces a two-dimensional grid-based layout system to CSS, offering significant flexibility in designing both large-scale web page layouts and smaller user interface elements. This system allows web developers to structure content into columns and rows, thus enabling precise control over the placement of items within a web page. Unlike traditional layout methods, which primarily rely on floats and positioning, CSS Grid provides a robust solution for creating complex layouts with ease [60].

The CSS Grid Layout module introduces a sophisticated two-dimensional grid system that allows developers to define and manipulate the structure of web pages with precision. By organizing content into rows and columns, it provides a powerful framework for creating complex layouts that were previously challenging to achieve using CSS alone. The grid layout is highly flexible, enabling both fixed and flexible track sizes, precise item placement, and control over overlapping content [60].

However, the grid system does not come without some disadvantages. One drawback is the potential rigidity in design, where the strict alignment of elements within grid lines may limit creative freedom, leading to uniformity that might not suit all design intentions. Additionally, the complexity of managing a grid layout can increase with more intricate designs, making it difficult to maintain a balance between functionality and visual appeal. The predefined grid structure may also constrain dynamic content that requires fluid adaptability across varying screen sizes and orientations. These challenges highlight the need for careful planning and consideration when utilizing grid layouts to ensure that they enhance rather than restrict the overall user experience.

5.1.3 Bootstrap Grid Layout Utilities

Bootstrap, one of the most widely used CSS frameworks on the web, offers a robust grid system that simplifies the creation of responsive and consistent layouts across various devices. Central to this system is the use of containers, rows, and columns, which work together to structure content effectively. Containers provide a base for alignment and padding, while rows organize columns and ensure even spacing through customizable gutters.

The grid's flexibility is evident in its 12-column layout, allowing designers to allocate different proportions of space to content blocks. This system is responsive, adapting to various screen sizes through predefined breakpoints that ensure the layout remains cohesive. Advanced features like nested columns and variable-width content further enhance the grid's versatility, making it a powerful tool for modern web design.

5.2 Using a Grid Structure for Page Layout

Following a thorough analysis of prevalent layout representation solutions in web development, a custom layout system was created to address the specific requirements of representing the spatial organization of detected UI elements. This system was designed with a focus on generality, ensuring that the General Page Description remains a generic, language- and framework-agnostic representation of a user interface. This emphasis on generality is particularly reflected in the layout representation.

The layout system is fundamentally based on a grid structure composed of rows and columns. Rows divide the available space vertically, while columns partition it horizontally. Drawing inspiration from the flexbox CSS module, each column is assigned a *flex* value, a numerical range from 0 to 1 that specifies the percentage of the available width the column occupies.

The layout structure is organized in a recursive manner: each row can contain a set of columns, and each column can house a set of rows. At the lowest level of this nested hierarchy, specialized row and column elements, referred to as leaves, contain the actual elements of the page. An illustration of this layout structure is provided in Figure 5.1.

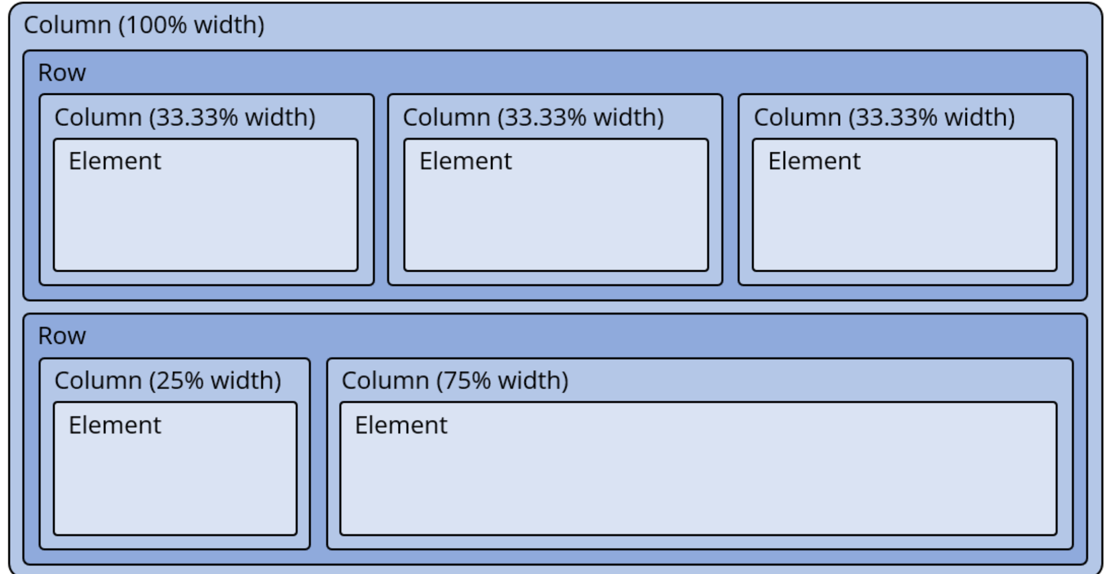


Figure 5.1: Example of a layout produced by the Layout Generation Module. The detected UI elements are organized into nested rows and columns.

5.2.1 Layout Generation Algorithm

The output of the Element Analysis Module comprises a list of detected elements, each with known positioning and dimensions within the page. Building on this foundation, an algorithm can be defined to iteratively group elements into rows and columns, ultimately constructing the final page structure.

The Layout Generation algorithm is based on two principal operations:

- **Row Splitting**: starting from a row, elements are grouped horizontally into columns. Horizontal groups are identified by the overlapping elements on

the horizontal axis. If multiple groups are identified, they are translated into corresponding columns, to which the column splitting process is recursively applied. If only one group is identified, the row is marked as a leaf, and the splitting process concludes.

- **Column Splitting**: starting from a column, elements are grouped vertically into rows. The grouping is performed by identifying vertically overlapping elements, with each group being translated into a row. Each resulting row then undergoes its own splitting procedure. If only one vertically overlapping group is identified, the column is marked as a leaf, ending the recursive process.

At the onset of the Layout Generation process, where no initial rows or columns exist, both row splitting and column splitting are applied to the initial set of elements. The operation that produces the initial division dictates the subsequent steps, leading to the full layout.

This recursive grouping of elements continues until leaf rows and columns are obtained, each containing no more than one UI element. The resulting nested structure mirrors the logical organization a front-end developer might employ to replicate the spatial arrangement of the input image.

Following the splitting process, all generated columns are assigned a *flex* value, calculated as the ratio of their width to the width of the parent row.

5.2.2 Post-processing for Human-like Layout Organization

Post-processing steps are essential to refine the layout generated by the automated process, ensuring it aligns more closely with the strategies employed by human front-end developers. The automated nature of the Layout Generation process can introduce artifacts that degrade the quality of the output and, subsequently, the code produced during translation. To address these issues and enhance the layout structure, a series of post-processing steps are applied.

Columns Width Adjustment

The initial post-processing step involves adjusting the widths of the generated columns within each row, following these procedures:

- Round the flex value of each column to the nearest multiple of 0.05.
- Sum the newly computed flex values for all columns within a row. If their total deviates from 1, adjust the flex of the last column to correct the discrepancy.
- Remove columns that have a flex value of 0.

This adjustment process improves the layout by:

- Normalizing flex values in a manner consistent with typical developer practices.
- Eliminating unnecessary columns generated due to minor gaps between elements, which are typically managed through margin and padding. This reduction results in cleaner, more maintainable code.

Related Consecutive Columns Merging

Further simplification of the layout is achieved by merging consecutive related columns based on heuristic rules. This post-processing procedure scans each row containing leaf columns from left to right and applies the following rules:

- Merge a “text”-type column with the next column if it is empty.
- Merge a “text”-type column with the previous column if it is empty.
- Merge a “text”-type column containing a single word with the previous “text”-type column if the next column is empty.

This step reduces the number of columns while preserving the functional integrity of the layout, thereby simplifying the structure and improving the quality of the generated code.

5.3 The Final Result: General Page Description

The Layout Generation process is the final step in extracting all relevant information from the initial image. Upon completion, the following information is obtained:

- A list of elements detected within the image, including their specific content and styling details.
- The organizational structure of these elements, presented as nested rows and columns.
- Additional “general style” information, which serves as a fallback in cases where specific element data is unavailable.

This information can be used to generate front-end code that reproduces the content of the original image with a high degree of accuracy. To facilitate this, the information is organized within a well-defined, language-independent data structure, which serves as the foundation for the code translation process. This generic structure is referred to as the **General Page Description** and is formatted as follows:

- It contains a **General Style** data structure, which includes generic styling information for the page:
  - The background color of the page.
  - Text colors for small, medium, and large-sized text.

  This data is derived from the Element Analysis Module.
- It includes a **Layout** data structure, corresponding to the output of the Layout Generation Module, which also encompasses the outputs from the Element Detection and Element Analysis Modules.

The General Page Description is a fundamental concept within the developed application, serving as the definitive reference for all subsequent code generation steps. Its primary advantage is its reusability; with an appropriate translation module, it can be utilized as a blueprint to target any platform of interest to the user.

Chapter 6

Modules Integration and Overall Application Architecture

As discussed in Section 2.3, the application is organized into a series of modules, each responsible for implementing distinct aspects of its functionality. This structure adheres to a recognized software design pattern [61], which advocates for a modular approach where each module addresses a specific part of the system's functionality and interacts with other modules via well-defined interfaces. This approach enables the adoption of a *divide-et-impera* strategy, allowing the developer to concentrate on solving smaller, more manageable problems individually.

Beyond examining the functionality of each module in isolation, it is essential to emphasize the integration between these modules within the broader context of the application's design. Building on this foundational concept, the overall architecture of the application is explored, not only considering the core functions of individual modules but also their role within the complete system, which is realized as a web application.

6.1 Modules Functionality Integration

The application's components are implemented as **Python modules**, which serve as the fundamental units for code organization and reuse. A Python module typically consists of a single file containing code, including functions, classes, and variables, which can be imported and utilized by other modules or scripts. This modular approach facilitates code reuse and organization. Within a module, elements can be designated as public, allowing access by other modules, or private, by prefixing

names with an underscore, indicating they are intended for internal use only. This distinction between public and private elements helps protect the module's internal logic from unintended interference, thus supporting the development of well-encapsulated software components [62].

In this application, each module comprises private methods that encapsulate its internal functionality, alongside a limited set of public methods that serve as interfaces for other modules. This approach maximizes the benefits of modular architecture while providing a level of abstraction that conceals implementation details.

The interaction between modules is structured as follows:

- The Element Detection Module uses a function provided by the OCR Module to extract text data from images, integrating this information with the results from Edge Detection and Contour Detection processes.
- The Element Detection Module, Element Analysis Module, and Layout Generation Module provide functionality utilized by the higher-level Image Analysis Module.

The Image Analysis Module encapsulates the application's core workflow. It interfaces with all other modules to perform the analysis process, from input image processing to the generation of the General Page Description. As such, it functions as the entry point for the core application functionality, managing both the process structure and overall configuration.

Given that the application is a web-based system, the back-end, which houses the entire application logic (i.e., the described modules), must expose a set of APIs for front-end interaction. In this project, the REST API leverages the Image Analysis Module to execute the core functionality, converting an input image into a JSON object representing the General Page Description. Subsequent sections will address the additional translation functionality, which enables the conversion of this generic representation into code.

6.2 Applying Recursive Processing to Containers

6.2.1 Recursive Analysis of Container Elements

One unresolved issue pertains to the processing of special *container* elements. These elements are identified when the employed MLLM detects multiple UI components within a corresponding image segment. An example of a case in which such element type is detected is in the presence of a "card" section, in which a subset of UI elements is encapsulated and conceptually distinct from the remainder of the page. Merely recognizing these elements is insufficient for their accurate representation in

the General Page Description; additional processing is necessary to analyze their internal structure.

The proposed solution involves recursively applying the image analysis process - previously described for the entire input image - to the cropped image corresponding to the identified *container* element. The extraction of this internal image crop is facilitated by the presence of *child* elements, as detailed in Section 3.4.1. Specifically, the bounding rectangle of the container's *child* element is utilized to crop the image, thereby isolating the container's internal components. This cropped image is then subjected to the complete Image Analysis workflow.

Once a General Page Description object is generated for the *container* element, its internal layout information is extracted and incorporated into the content field of the original container within the General Page Description of the original image. Through this recursive approach, all relevant information is eventually consolidated into a single GPD, providing a comprehensive description of the entire image content. Given the recursive nature of this process, a *container* element may itself contain other containers, which are processed similarly.

The recursive processing of containers is implemented in the high-level Image Analysis Module. While this recursive strategy is well-suited to the task, it also introduces certain risks that require careful consideration and intervention.

6.2.2 Pitfalls of the Recursive Approach and Adopted Solutions

Recursion is a fundamental concept in computer programming, where a function calls itself directly or indirectly to solve a problem. This technique is particularly powerful for tasks that can be broken down into smaller, similar sub-problems, as it allows for a concise and elegant solution [63]. Common examples of problems suited to recursive approaches include traversing hierarchical data structures, such as trees, or performing computations like calculating factorials or Fibonacci numbers. In these cases, recursion simplifies the logic by allowing the programmer to focus on solving the smaller sub-problems, with the assumption that the base case will eventually be reached, terminating the recursive calls.

Despite its elegance, recursion introduces several risks and pitfalls that require careful consideration. One of the primary concerns is the potential for **excessive memory usage**, as each recursive call typically consumes stack space to store the function's state. In cases where the recursion depth is significant, this can lead to a **stack overflow**, causing the program to crash. This issue is particularly pronounced in languages with limited stack size, where deep recursion can rapidly exhaust available resources, leading to system instability [64].

Another risk associated with recursion is the possibility of **infinite recursion**, which occurs when the base case is either incorrectly defined or never reached.

Infinite recursion results in the function repeatedly calling itself without termination, ultimately causing the program to run indefinitely or until a stack overflow occurs. This is a common pitfall for novice programmers, who may overlook the importance of properly defining and ensuring the base case [64].

The risk of infinite recursion is particularly significant in the context of this project, as each recursively processed container incurs a notable cost due to the various API calls required to analyze its internal elements. Errors that result in excessive or infinite recursion can lead to a substantial increase in operational costs and severely disrupt the user experience. Therefore, implementing a robust safety mechanism to prevent such occurrences is essential.

The solution adopted involves the use of a **configurable recursion depth threshold**. The Image Analysis Module monitors the current recursion depth as it applies recursive analysis. If the depth exceeds the predefined threshold, the process is immediately halted, and placeholder elements are returned as the output of the analysis. While this mechanism effectively prevents the issues associated with deep recursion, it also imposes limitations on the depth at which the Image Analysis Module can operate. Consequently, selecting an appropriate threshold value is critical to achieving a balance between accuracy and robustness.

Initial testing of the application revealed that a threshold value of 2 is sufficient to support the majority of use cases while offering substantial protection against potential recursion-related errors.

6.3 Translation Modules

The ultimate goal of the application is to produce code that targets a desired front-end platform and allows the user to accelerate the development process. In order to transform the General Page Description into code, appropriate Translation Modules are defined, each targeting a specific platform.

For the purposes of developing an initial application prototype, it was decided to target two specific formats:

- Angular, the prevalent front-end framework in use at Blue Reply, for concrete applicability in the majority of the company's current and future projects. Support for the PrimeNG UI component library was also taken into account, as it provides a diverse set of ready-made UI components and is used in all Angular-based projects at the company.

- Simple HTML and CSS with the use of Bootstrap 5, the most widely used CSS framework on the web. This target was chosen to provide a simple way to generate generic front-end code that can be easily adapted for usage with any other framework of choice (e.g. React).

6.3.1 The Bootstrap 5 Translation Module

The Bootstrap 5 Translation Module starts from the input General Page Description JSON document and generates an output that is composed of CSS code and HTML code. The HTML section defines the content and the structure of the genereated page, and uses a mix of ready-made Bootstrap style classes as well as custom classes that are defined in the attached CSS code. A well-defined naming convention is used for the custom CSS classes; whenever possible, the same CSS class is re-used to style multiple components that share aesthetic features.

As mentioned in Section 4.4.2, the Element Analysis Module applies optimizations that may imply the absence of precise styling attributes for a subset of “text”-type elements. In such case, the General Style portion of the General Page Description document is used to compensate for such absence. The use of these general styling attributes for text elements is not only useful due to the optimization of the analysis process, but it also avoids inconsistencies in the styling of text elements due to inaccuracies of the visual analysis by the LLM.

An example of translation of a screenshot into HTML/CSS Bootstrap 5 code can be seen in Figure 6.1.

(a) Input image.

(b) Render of the generated HTML and CSS.

Figure 6.1: Example of the conversion of a screenshot into HTML and CSS using the Bootstrap 5 Translation Module.

6.3.2 The Angular/PrimeNG Translation Module

The Angular/PrimeNG Translation Module is tasked with converting the input General Page Description into a corresponding Angular component. This module is motivated by the large use of the Angular framework at Blue Reply for the majority of web application projects. Furthermore, the module makes use of the PrimeNG component library, which provides a diverse array of ready-made and customizable UI components [12]. Within the company, it is used in all Angular-based projects as a powerful tool that enables increased consistency and development speed.

Angular is a component-based framework, and as a result, the corresponding Translation Module must be able to produce a fully-fledged component that can be easily inserted into the application. In Angular, a component is the foundational building block used to create applications. It organizes the application into modular, maintainable, and scalable parts, each with a clear responsibility. Every component consists of a few key elements: a TypeScript class that manages the component's behavior (such as handling state, user interactions or data fetching), an HTML template that defines what gets rendered in the DOM and a CSS selector that defines the styling of the component. Components can be organized with HTML and CSS in separate files for better code management [10].

In the case of the application's Translation Module, the output is divided into three parts:

- The Typescript class that defines the basic component behaviour, which in this case is only used to define basic component properties and does not actually include any front-end logic.
- The HTML template, which is generated starting from the General Page Description.
- The CSS selector, which contains all the custom styling that is derived from the GPD and is used to enrich the representation of the template.

Following this strategy, the aesthetic part of the component is derived from the GPD, while its functional and interactive part is added manually by the programmer.

6.4 Application Architecture

The application employs a client-server architecture, a widely adopted approach in web programming. In this model, the client and server function as separate entities that communicate over a network. The client, typically a web browser, sends requests to the server, where the application logic is hosted. The server processes these requests, retrieves the necessary data, and returns the appropriate response

to the client. This interaction enables the client to display content, execute actions, and facilitate user interaction. The client-server model is fundamental to modern web applications, as it allows for the separation of concerns, leading to scalable and efficient systems [65].

The modules previously described, including the Translation Modules, are integrated within the application's back-end. These services are made available to the front-end through a REST API. The REST architecture, introduced by R. T. Fielding in 2000 [66], emphasizes a stateless, client-server communication model using standard HTTP methods. It is inherently designed around the client-server paradigm, promoting a stateless approach where each request contains all the information required for processing, without relying on stored session data. Moreover, REST facilitates a uniform interface between the back-end and front-end, allowing each to evolve independently [66]. This adaptability is particularly advantageous for an experimental project like the one discussed, where the application's internal structure may change rapidly.

As is common in client-server systems [65], the back-end relies on a database to store persistent data. This includes:

- The system's configuration (e.g., the version of the MLLM to be used, the content of analysis prompts, etc.).
- User information, including roles.
- A history of analyzed images and generated GPDs for each user.

Finally, the system makes use of the external OpenAI API to leverage the GPT-4o model for multimodal inference. This API enables communication with a remote server to obtain outputs from available OpenAI models, following a token-based pricing scheme [57]. This remote interaction is a critical component of the system's architecture, as it plays a central role in the Element Analysis Module.

The overall system architecture is further illustrated in Figure 6.2.

6.4.1 Technology Stack

To implement the described architecture, the application utilizes a technology stack that is well suited for rapid iteration and adaptation, which aligns with the experimental nature of the tool. The stack incorporates platforms that are already familiar and widely used within Blue Reply. This choice facilitates easier comprehension of the system's structure for existing employees and streamlines the onboarding process for future project members. A detailed description of each component of the application's stack and the reasons behind its adoption is provided in the following Sections.

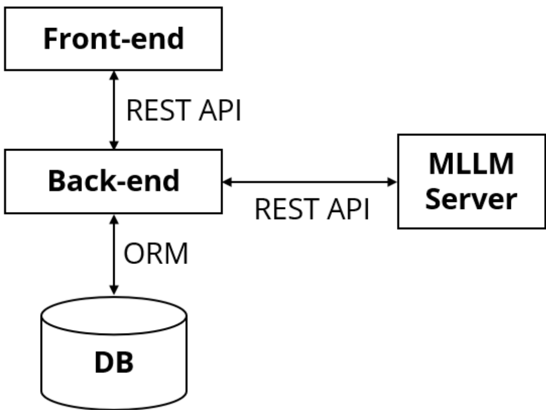


Figure 6.2: Overview of the application architecture.

6.4.2 Data Persistence Layer

The data persistence layer for the application is implemented using **SQLite**. SQLite is a lightweight, serverless, and self-contained database engine that is well-suited for experimental web applications [67]. It is embedded directly within the application, eliminating the need for a separate server process and reducing setup complexity. This simplicity, combined with its zero-configuration nature, makes SQLite an ideal choice for early-stage projects where ease of use and quick deployment are critical. Additionally, its file-based architecture ensures that the database is highly portable, facilitating smooth transitions between development environments. Despite its compact size, SQLite offers full SQL support [67].

SQLite's architecture is characterized by a modular design that efficiently integrates its core functions while maintaining simplicity and portability. At a high level, SQLite is structured into four primary groups of modules. The core modules manage the ingestion and execution of SQL statements, utilizing a virtual machine to process these instructions. The SQL compiler modules translate SQL commands into bytecode, which the virtual machine executes. Backend modules handle data storage and interaction with the operating system, ensuring that data is consistently and reliably managed. Additionally, SQLite includes accessory modules that provide various utilities and extensive testing capabilities, which contribute to its reliability and widespread adoption [68].

This modular architecture allows SQLite to operate as an embedded database within the host application, eliminating the need for a separate server process. The system's architecture is further optimized for OnLine Transaction Processing (OLTP) while remaining flexible enough to handle a broad range of use cases, from simple key-value storage to complex analytical queries [68].

In the context of developing an experimental tool, it is crucial to select software

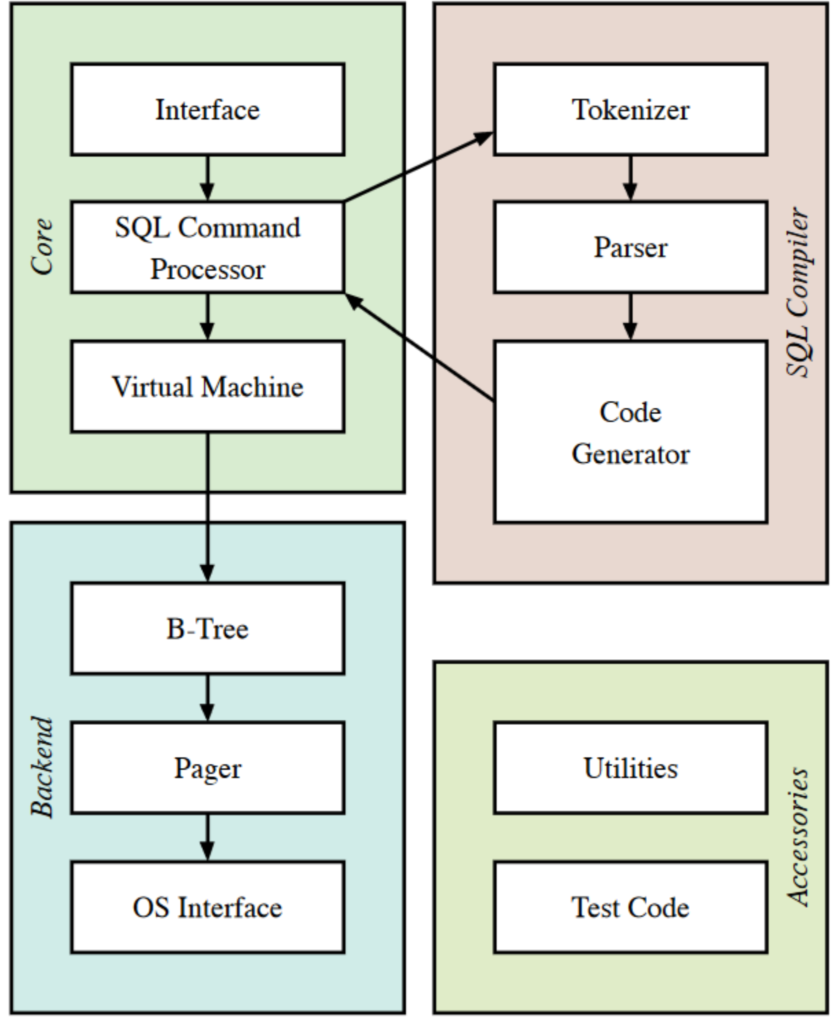


Figure 6.3: Overview of the architecture of SQLite, from [68].

components that can adapt to a rapidly evolving environment, interface seamlessly with various external systems, and require minimal deployment complexity. Based on these considerations, SQLite was chosen as the database engine for the persistence layer. Its widespread adoption and robust community support provide stability, while its self-contained, file-based structure minimizes complexity and accelerates development.

A significant limitation of using SQLite for the persistence layer in a web application is represented by its constrained concurrency model, which relies on a locking mechanism that may lead to performance bottlenecks. These limitations can adversely impact the scalability of the application. Nonetheless, given the internal nature of this tool, which demands low scalability to accommodate only a

limited number of concurrent users, the choice of SQLite remains justified despite these drawbacks.

Another limitation of SQLite is its lack of advanced features, such as user management, fine-grained access control, and built-in replication, which are commonly found in more sophisticated database systems. However, the simplicity of the web-based tool described here negates the need for such advanced capabilities.

6.4.3 Backend Layer

The application backend is built on top of the Image Analysis Module and the Translation Modules described in Sections 6.1 and 6.3. As previously mentioned, it encapsulates their functionality in a REST API that is implemented using **Flask**, a lightweight Python web framework.

Flask is a framework designed to be simple and lightweight, making it an ideal choice for small-scale applications and rapid prototyping. Its minimalist approach allows developers to implement only the essential components, granting significant flexibility and control over the application architecture. Flask's simplicity also makes it easy to learn and use. However, this simplicity can be a drawback for more complex applications, as Flask lacks many built-in features, such as authentication and database management, which developers must implement themselves or add via extensions [69].

In order to make the framework suitable for the specific use case of the application, two extensions were used to expand its capabilities:

- Flask-CORS: an extension for configuring and automatically handling Cross Origin Resource Sharing (CORS).
- Flask-SQLAlchemy: an extension that allows to integrate the SQLAlchemy library into the Flask ecosystem.

SQLAlchemy is a versatile toolkit for integrating Python applications with databases, offering a comprehensive set of tools that can be used independently or in combination. Its primary components are the Core and the Object Relational Mapper (ORM). The Core provides the foundation for SQLAlchemy's functionality, featuring the SQL Expression Language, which allows for the construction and execution of SQL expressions through composable objects. This system enables the management of database operations such as inserts, updates, and deletes. On top of the Core, the ORM facilitates the interaction with a database using a domain object model, automating tasks like data persistence through a unit of work pattern [70].

In the case of the described application, SQLAlchemy acts as the communication mechanism between the Python-based backend layer and the SQLite-based persistence layer; it thus represents a key component that allows to retrieve and modify

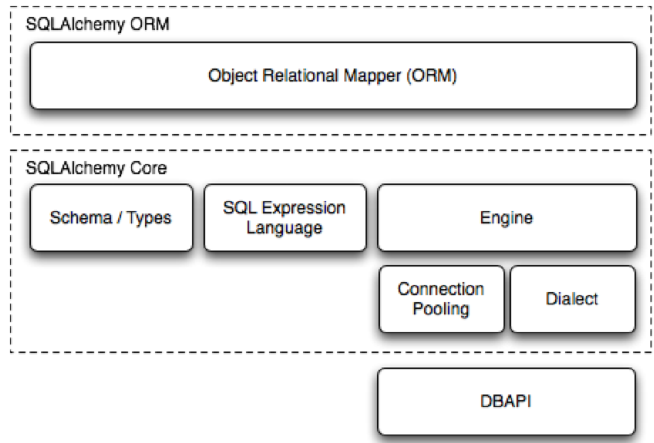


Figure 6.4: Architecture of SQLAlchemy, from [70].

the state of the system, allowing the use of domain models at the application level to interface with the underlying relational data.

6.4.4 Frontend Layer

The frontend of the application is implemented as a **Single Page Application** (SPA). SPAs are designed to enhance user experience by loading a single HTML page and dynamically updating content on the browser as the user interacts with the application [71]. This approach differs from traditional web applications, which require full page reloads for each user action. Instead, SPAs utilize asynchronous requests to fetch only the necessary data from the server, resulting in a more fluid and responsive user interface. However, this architecture may lead to an increased initial loading time compared to Multi-Page Applications (MPAs). Beyond this trade-off, SPAs promote modular development, allowing independent updates to components without impacting the entire application. This modularity enhances code maintainability and development efficiency [71].

For this application, the decision to implement the frontend as a SPA was primarily driven by the need for an improved user experience, particularly after the initial application bundle is loaded. Given the high interactivity required by the tool, especially for real-time image analysis and GPD translation processes, the SPA architecture was a logical choice.

In modern web development, various frameworks support the development of SPAs. For this project, **Angular** was selected as the framework of choice. Angular, developed by Google, is widely used for building dynamic SPAs. It facilitates the creation of responsive web interfaces through a component-based architecture,

where each component is composed of HTML, CSS, and TypeScript files.

The selection of Angular was based on several factors:

- It is a robust and mature framework, supported by a large community and regularly updated.
- Its design aligns well with a modular approach, enabling simultaneous and independent development of different parts of the application.
- It has a vast ecosystem of packages that allow for the easy integration of additional functionality into the application.
- It has been successfully used at Blue Reply for several years, demonstrating its reliability and scalability across a wide range of use cases; furthermore, the development team at the company has extensive experience with it.

The Angular frontend application communicates with the backend using the built-in *HttpClient* service, which facilitates the sending of HTTP requests and the processing of responses. The interaction with the backend API is organized into services, which group logically related API interfaces and map them to corresponding methods. These services and their methods are then utilized by the application's components to handle backend interactions.

6.5 REST API Analysis

The application backend's REST API is structured into distinct parts, with each one dedicated to delivering a specific subset of the system's overall functionality.

6.5.1 Core Functionality

The core functionality of the application centers on the Image Analysis and Code Translation workflow. This process involves two primary steps in transforming an image into functional code for the target platform:

1. The Image Analysis Module processes the input image and generates a General Page Description (GPD) in a standardized JSON format.
2. The appropriate Translation Module utilizes the GPD to produce platform-specific code.

To facilitate the Image Analysis process, a dedicated ***imageToJson*** API has been implemented. This endpoint allows authenticated users to upload an image file and receive the corresponding GPD as a response. The API also provides

information on the number of input and output tokens used during the Element Analysis step, enabling users to monitor OpenAI API usage and associated costs.

The ***imageToJson*** API incorporates Optical Character Recognition (OCR) processing during the Element Detection phase. Furthermore, it initiates a series of internal API calls from the application backend to the OpenAI servers, leveraging the GPT-4o multimodal model for Element Analysis. These two components of the Image Analysis process represent significant performance bottlenecks, resulting in slower response times compared to other endpoints in the REST interface. Consequently, it is crucial to manage user expectations by providing clear feedback and appropriate handling of the extended waiting time on the frontend.

The GPD Translation step is executed through the ***convert*** API. This endpoint enables authenticated users to submit a JSON document containing a General Page Description and specify a target platform for code generation. If the requested target platform (i.e., the corresponding Translation Module) is defined in the backend, the API responds with the platform-specific implementation.

6.5.2 Secondary Functionalities

To complement the application's core functionalities, the backend API incorporates auxiliary features. These additional components enhance the overall user experience and system capabilities. Specifically, the API includes separate endpoint groups dedicated to two key aspects:

- Session-Based Authentication
- User Generation History

These supplementary features work in synergy with the primary system processes to provide a comprehensive and secure user interface.

Session-Based Authentication

Session-Based Authentication is a well-known mechanism in web applications for managing user identity and access control. Upon successful login, the server generates a unique session identifier, which is stored on the client side as a cookie. This identifier accompanies each subsequent request, enabling the server to recognize the user and maintain their authenticated state throughout the session. The server associates this session ID with a specific user account and tracks pertinent data, such as permissions and activity.

The application under analysis utilizes the Flask framework's session functionality to implement a temporary form of Session-Based Authentication. This interim solution employs salted password hashes stored in the local database for each user. While salted hashes enhance security to some degree, it is important to emphasize

that this approach is not recommended for long-term implementation. The storage of any authentication-related information in a local application database, even in hashed form, is generally discouraged due to potential security vulnerabilities.

It is thus crucial to underscore that this authentication method serves solely as a provisional measure for initial prototyping and testing purposes. Pre-defined user accounts with randomly generated passwords were assigned to employees participating in the testing phase. **These accounts do not contain any sensitive personal information**. A more robust, externally provided authentication system is planned for implementation in the near future, following the conclusion of the initial experimental testing phase.

Within this temporary framework, each user is assigned one of two roles: “dev” (a standard user) or “admin” (an administrator with access to all users’ generation history and corresponding statistics).

The authentication flow is facilitated through a set of dedicated endpoints:

- The *login* endpoint accepts an email and password. Upon successful authentication, it stores the Session ID in an HTTP-only cookie.

- The *userInfo* endpoint allows the frontend to retrieve user information, such as name and role.

- The *logout* endpoint enables users to terminate the active session and delete the corresponding browser cookie, effectively exiting the application.

User Generation History

A critical feature for enabling the reuse of previously generated code and minimizing costs associated with redundant requests is the User Generation History. This history can be accessed by authenticated users to retrieve previously generated General Page Descriptions (GPDs) and efficiently convert them into code artifacts via a dedicated Translation Module. The history functions as a chronologically-ordered archive of generated GPDs, accessible at any time to produce code for any available target.

This historical data is stored in a dedicated table within the system’s database. Additionally, the graphical reference associated with each generation is preserved. However, the storage of numerous images in the backend presents a potential challenge. To mitigate this, input images are scaled down before storage, thereby reducing memory usage while still providing users with a suitable visual reference.

The history table is automatically populated with each call to the *imageToJson* endpoint. Access to this data is facilitated through the ***history*** endpoint, which can be utilized differently depending on the user’s role:

- A "dev" user must provide a user ID, which is used to filter the history entries before they are returned. If the supplied ID does not match the user's own ID, an error is generated.
- An "admin" user can provide an arbitrary user ID to filter the entries, or omit the ID altogether, in which case all history entries for all users are returned.

The *history* endpoint includes paging functionality, allowing users to specify an offset and a limit to retrieve a specific subset of the history. This feature is particularly useful for reducing loading times when viewing the data on the frontend. Additionally, time-based filtering is supported, enabling users to define a time window for the results by specifying a *startDate* and an *endDate*.

6.6 Adopted Design Principles and Guidelines

When designing the user experience for a user-facing application, it is essential to consider the various factors that contribute to ensuring a satisfactory level of usability. In the field of Human-Computer Interaction, substantial research has been devoted to identifying effective user interface design solutions and patterns for such systems.

While extensive literature addresses techniques for improving the user experience with traditional interactive systems, the recent widespread adoption of Large Language Models and their integration into user-facing applications have introduced a new set of challenges. Among these, the most pertinent to this thesis are as follows:

- The non-deterministic nature of Large Language Models introduces an inherent unpredictability in their output, necessitating the implementation of robust error-handling and retry mechanisms to ensure system resilience against potential failures.
- The generation delays often associated with AI-based products, particularly those leveraging LLMs, result in extended waiting times for users. Providing immediate and effective visual feedback during these interactions is crucial, though it can be challenging to achieve in a meaningful way.
- Users often expect to comprehend the rationale behind system decisions. Offering clear and informative explanations for AI behavior presents difficulties, particularly when the underlying processes are complex and not easily understood by users.

In 2019, a research team at Microsoft conducted a comprehensive analysis of Human-AI interaction challenges, which culminated in the development of a set

of *AI Design Guidelines* [72]. These guidelines were synthesized from over 150 recommendations sourced from academic literature, industry reports, and public articles. The guidelines underwent further validation through iterative testing [72], which included:

- Heuristic evaluations of existing AI products using the guidelines.
- User studies involving 49 participants, focused on applying the guidelines to popular AI-based applications.
- Expert evaluations from Human-Computer Interaction (HCI) professionals.

The AI Design Guidelines address all phases of user-AI interaction. Ideally, this interaction begins by setting clear expectations, with the system explicitly communicating its capabilities and limitations upfront. Ensuring that users understand the system's functions and the reliability of its performance fosters realistic expectations and trust from the outset [72].

In cases where the AI produces errors, users should be provided with easy mechanisms to correct mistakes or dismiss undesired actions. The AI system should offer explanations for its decisions to help users understand its behavior. This approach enhances transparency and facilitates effective error resolution [72].

These guidelines were consistently considered during the design of the user interface. Although the tool incorporates a complex internal workflow, the user interaction remains straightforward:

1. The user uploads an image file for analysis.
2. The system generates a General Page Description and presents the user with an overview of the result.
3. The user selects a target platform for code generation.
4. The system employs the relevant Translation Module and displays the generated output to the user.

Additionally, the user can access the History feature, which allows the retrieval of previously generated code and related information.

The application is structured into two main sections:

- The *Generate* section, where users can upload an image and follow the described process to generate code.
- The *History* section, where users can review a list of previous code generations, filtered by a selected time window. "admin" users are also able to filter by individual users or view the history of all users.

The *Generate* section features a straightforward interface, as depicted in Figure 6.5, enabling users to promptly upload an image and initiate the conversion process. During the execution of this process, a loading screen is displayed to provide feedback to the user.

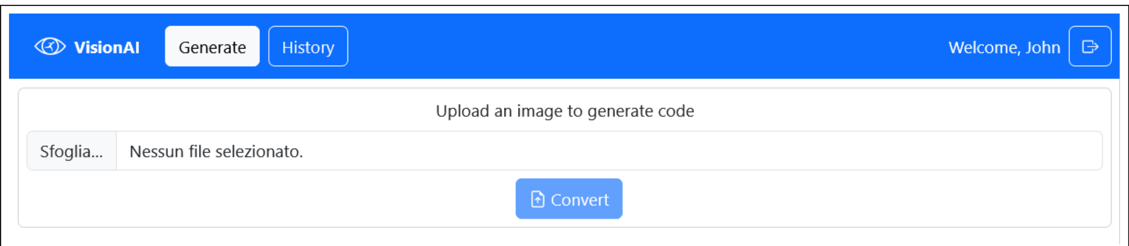


Figure 6.5: Initial interface shown in the *Generate* section.

Upon completion of the processing, the final result is displayed, as illustrated in Figure 6.6. A visual preview appears at the top of the page, offering immediate feedback on the accuracy of the translation. Users can switch between available target platforms using a drop-down selector located at the top-right corner of the preview box. As different platforms are selected, the preview is updated accordingly.

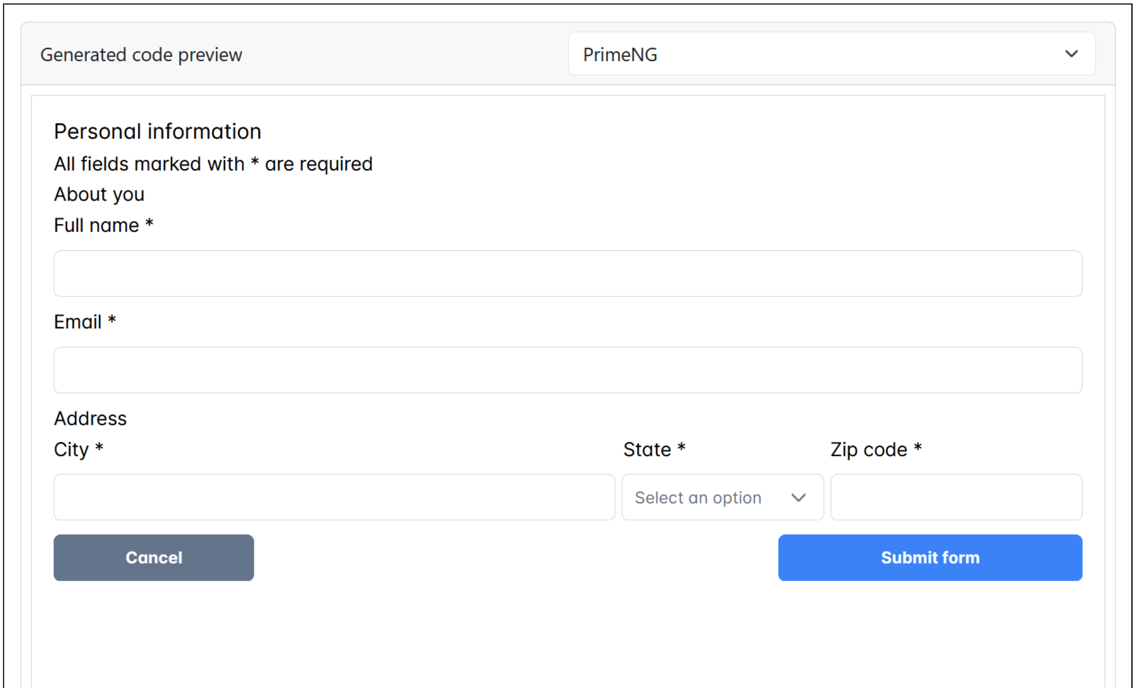


Figure 6.6: Example of the visualized preview for the generated code.

Beneath the preview box, two accordions provide additional information:

- A visual representation of the generated layout, offering a detailed graphical overview of the detected elements and their respective locations on the page. An example of this visualization is shown in Figure 6.7.
- The generated code, which the user can easily copy with a single click.

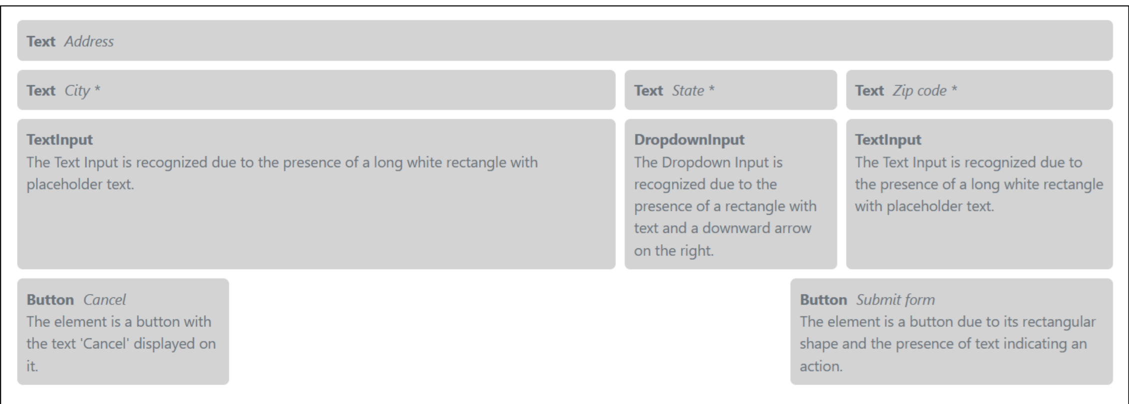


Figure 6.7: Example of the generated layout visual representation.

As demonstrated in the provided example, a brief description is displayed for each non-textual element detected, offering an explanation of why a specific type was assigned during the Element Analysis process. This feature was introduced to inform users about the rationale behind the choices made during the AI-driven steps of the process, in line with the *AI Design Guidelines* [72].

At the bottom of the page, a *Regenerate* button allows users to quickly retry the code generation process using the same reference image. This functionality addresses the inherent unpredictability of the Element Analysis process, which arises from the non-deterministic nature of the Large Language Model output.

The *History* section, depicted in Figure 6.8, offers a paginated view of the user's past code generations, organized chronologically from the most recent to the oldest entries. An expandable *Search filters* section enables users to filter results using a traditional form, as previously described. By selecting a specific history entry, users can access detailed information, including the visual layout overview and generated code for each target platform. An example of this view is shown in Figure 6.9. If the code for a particular platform has not been generated previously, the translation process occurs in real time and the output is displayed as the corresponding section is expanded.

As illustrated, the design prioritizes simplicity and usability, with an interface that is intuitive and predictable in its behavior. Special attention was given to providing immediate feedback for each action through pop-up notifications and clearly explaining the reasoning behind the system's decisions and behavior.

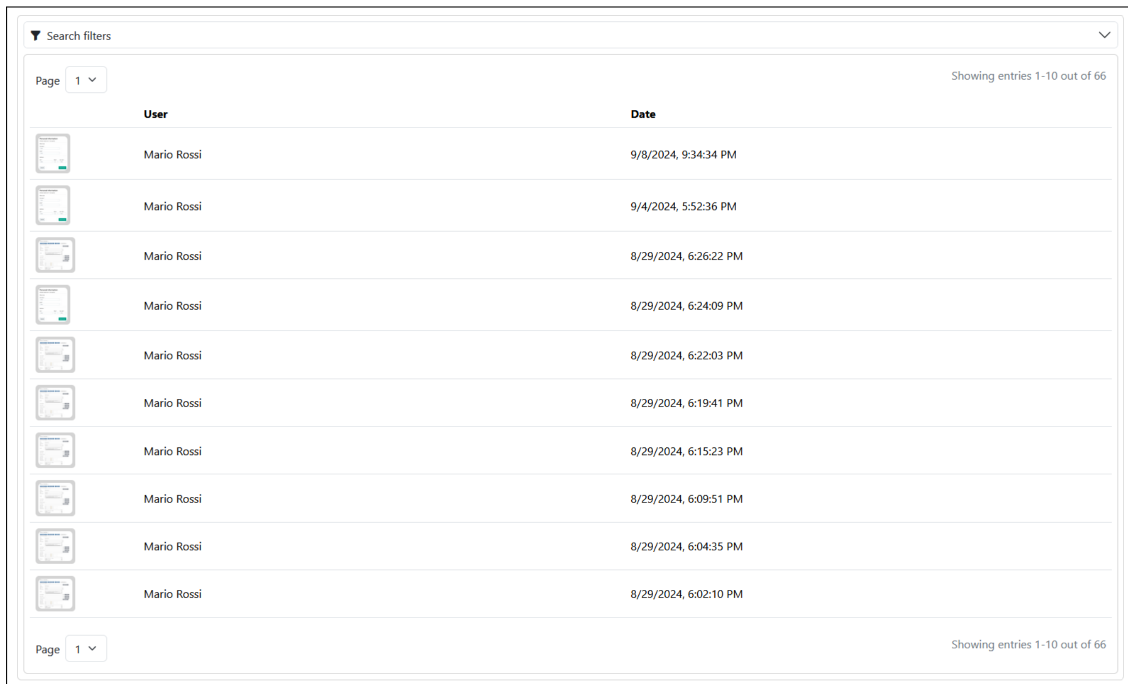


Figure 6.8: Interface shown in the *History* section.

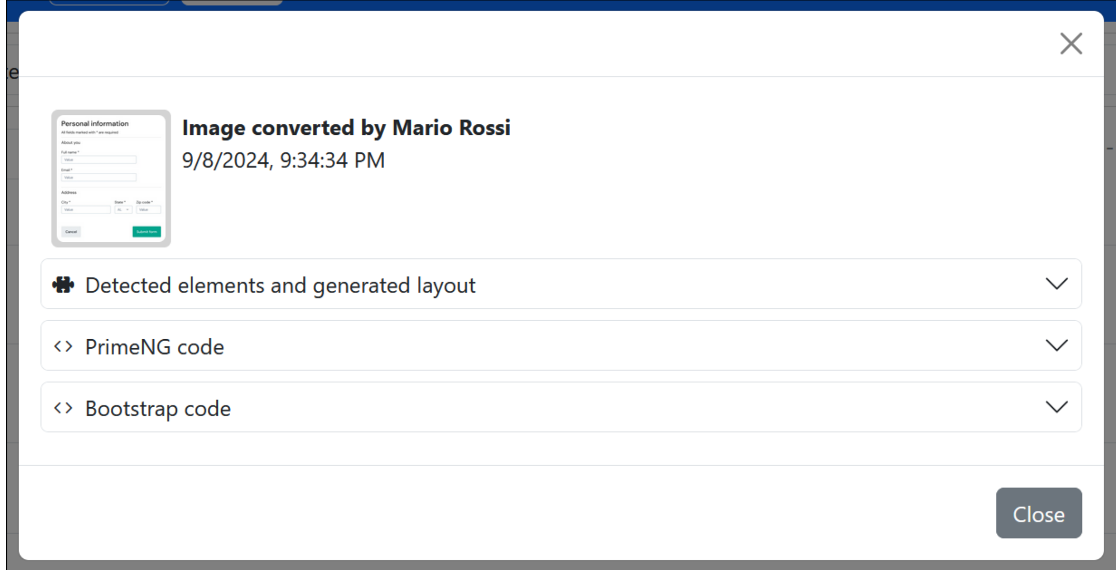


Figure 6.9: Interface shown as overview of a single *History* entry.

In the event of errors, users have the option to either retry the generation process or manually adjust the generated code to align with the visual specifications.

Chapter 7

Evaluation and Results

Upon completing the prototype implementation of the application, a series of tests were conducted to assess the quality achieved and the tool's practical usefulness for internal development tasks. The following sections outline the initial expectations and project requirements, followed by a discussion of the actual results and their evaluation.

7.1 Objectives and Requirements

Following the initial research phase, during which the foundational aspects of the image conversion process were established, a set of requirements was compiled based on the expectations of Blue Reply's development team. These requirements served as the primary reference throughout the subsequent development activities. Below is a summary of those requirements:

- **User Authorization & Authentication**.
- **Image Conversion to the General Page Description**: The user must be able to upload an image (in PNG or JPEG format) and convert it into a generic format, called the General Page Description (GPD), which provides a comprehensive, language-independent representation of its content.
- **Conversion of the GPD to HTML and CSS Code using the Bootstrap 5 Framework**.
- **Conversion of the GPD to an Angular Component using the PrimeNG UI Framework**.
- **Generation History**: Users must have access to their code generation history, allowing them to easily retrieve previously generated code.

By the end of the development phase, all requirements were met. However, meeting the initial specifications does not necessarily indicate the overall success of the project. The ultimate goal of the company's experiment was to assist the web development team by improving their productivity in creating front-end interfaces.

To achieve this goal, two key metrics were considered:

- **Conversion Time**: The time elapsed from the initial image upload to the moment the generated source code is delivered to the user. This should be minimized to maximize the time saved by using the tool.
- **Element Analysis Accuracy**: The percentage of correctly classified elements within an image by the Element Analysis Module. Ideally, this should reach 100%, eliminating the need for user intervention.

Using this framework, a set of mockups and screenshots provided by the development team served as a baseline for evaluation. For each, the relevant metrics were collected and analyzed.

7.2 Early Application Testing with Real Usecases

To assess the tool's effectiveness, the Blue Reply development team provided a set of 24 mockups and screenshots for testing. The mockups were taken from design documents of previous development activities on a large, ongoing project, while the screenshots came from a legacy web application slated for migration to a new Angular-based architecture.

A subset of these references, along with screenshots of the unmodified generated code, is presented below. To protect sensitive information, some text labels and placeholder content have been altered; however, the images' structure, graphical features, and layout remain unchanged.

Figure 7.1 illustrates a typical use case, in which the user provides a mockup of an input form for editing a system resource. Most enterprise systems developed and maintained by Blue Reply involve managing and storing large amounts of data, making input forms a common front-end development task. In this example, the tool successfully identified the majority of UI elements, requiring minimal manual intervention.

Figure 7.2 illustrates the translation of a search page, where a table displays a large number of entries filtered based on user input. The example shows the Angular/PrimeNG translation, which relies less on the specific styling details contained in the General Page Description and adopts a more generic, consistent styling approach. While styling information is not ignored (e.g., different background

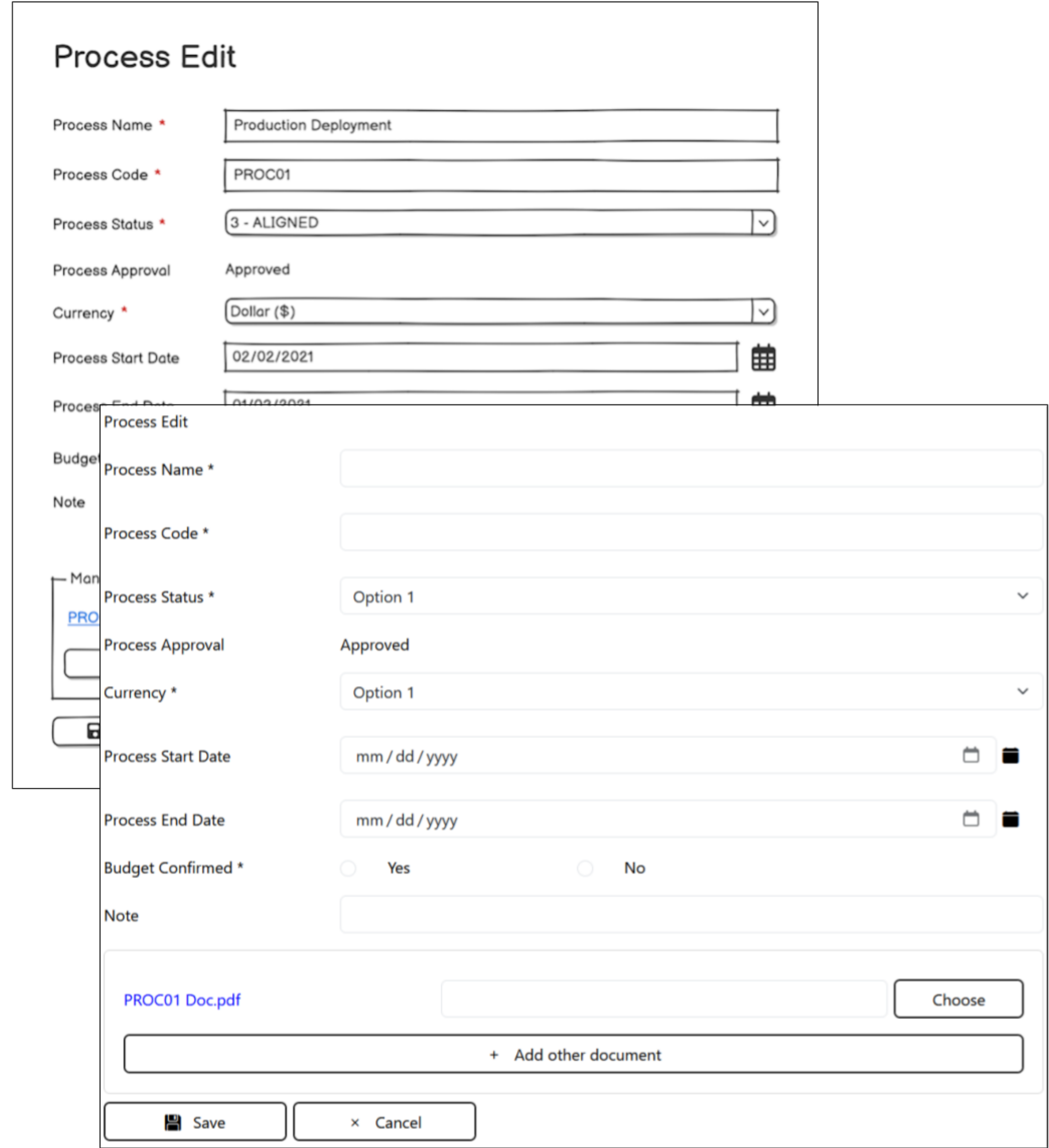


Figure 7.1: "Process Edit" test mockup input (top left) and the rendered Bootstrap-translated output (bottom right).

colors for buttons are translated into specific PrimeNG button variants), it is used as a reference upon which the UI library's utilities are applied.

When translating the table from the "Process Search" screen in Figure 7.2, the tool uses the content of the first row as a template and replicates it for the remaining rows, matching the row count in the original image. This approach

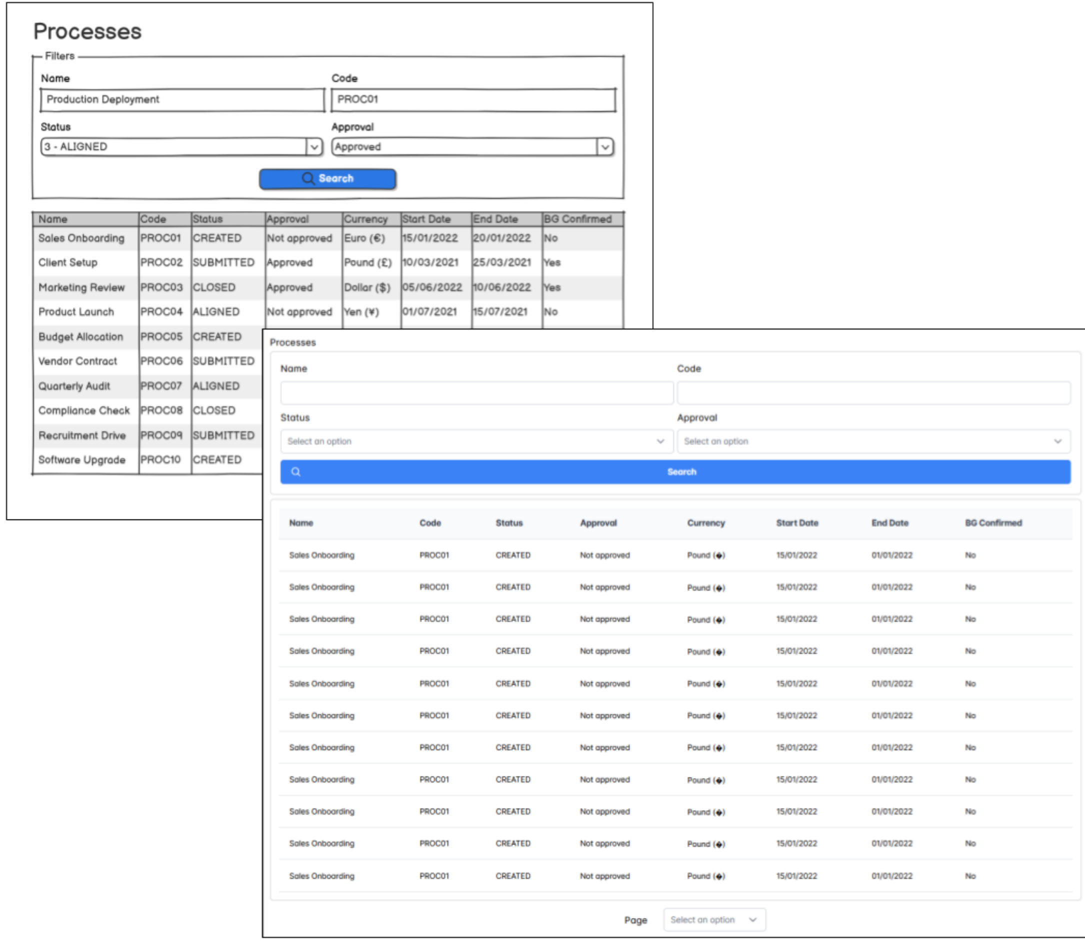


Figure 7.2: "Process Search" test mockup input (top left) and the rendered Angular/PrimeNG-translated output (bottom right).

reflects the fact that the logic for extracting and displaying data in the table is left to the developer, who is responsible for populating the interface with meaningful data. As a result, the table is filled with placeholder values to provide a structural reference.

Figure 7.3 presents a different example, where a screenshot of a legacy web application is used as input for the tool. While the result is generally accurate, there are some inconsistencies in the translation of certain UI elements.

Finally, Figure 7.4 illustrates a more complex use case, where the input image contains three cards displaying information about a selected resource. The tool produces a fairly accurate result, though with minor inconsistencies in the generated layout and UI elements.

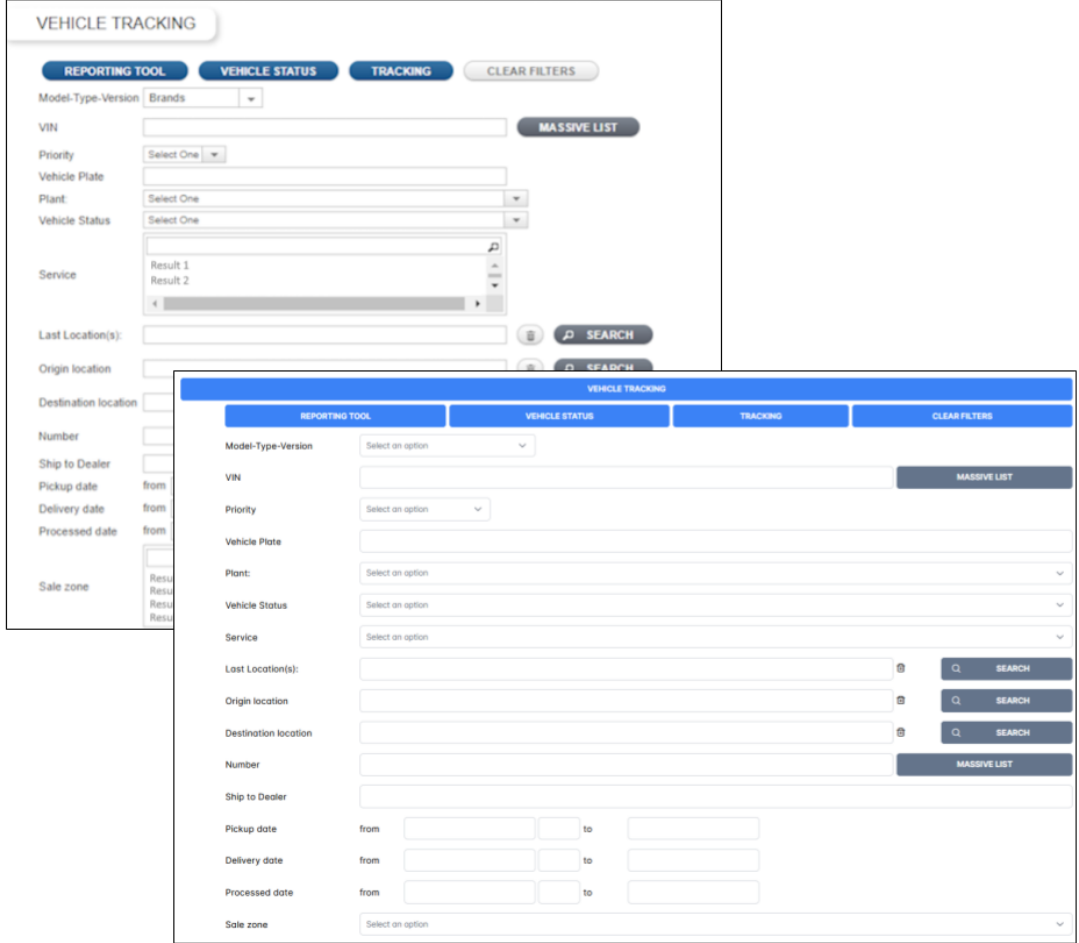


Figure 7.3: "Vehicle Tracking" test screenshot input (top left) and the rendered Angular/PrimeNG-translated output (bottom right).

As shown in these examples, the tool is capable of closely resembling the input reference image, though occasional inaccuracies require manual adjustments by the developer. During testing, the following types of errors were frequently observed:

- Page and section titles were often not properly highlighted, requiring the programmer to manually adjust their styling.
- Small icon buttons were frequently misinterpreted as icons by the MLLM.
- Text and icon elements with colors different from the rest of the page were not styled correctly, necessitating manual corrections.

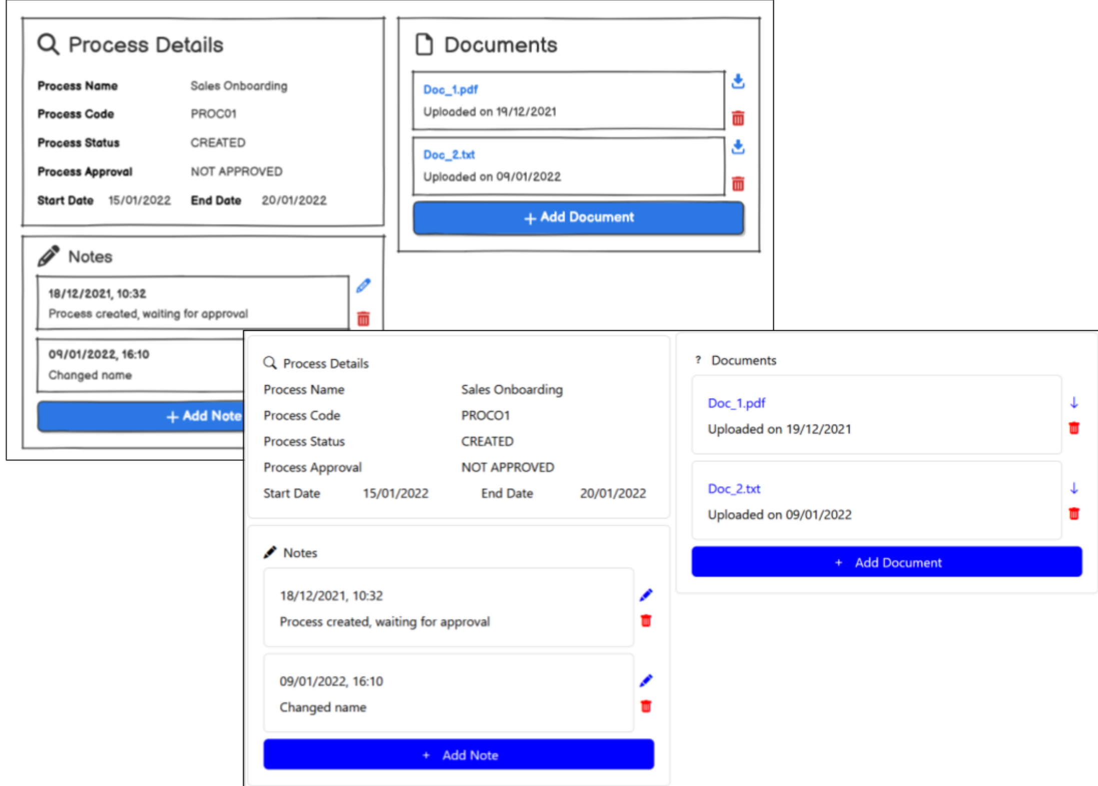


Figure 7.4: "Process Details" test mockup input (top left) and the rendered Bootstrap-translated output (bottom right).

- Unusual input field designs, such as the separate calendar icon next to the date input in Figure 7.3, led to errors, as the tool was unable to logically associate spatially separated elements.

7.3 Overall Results Assessment and Observations

For all the processed test images, the Conversion Time and Element Analysis Accuracy metrics were calculated using well-defined procedures:

- The Conversion Time was measured as the total waiting time reported by the web browser for the *imageToJson* and *convert* API calls. Since the *convert* API offers two possible targets, both were tested, and the worst case (i.e., the longest waiting time) was considered.
- The Element Analysis Accuracy was evaluated by assigning a score based on how accurately each UI element was analyzed:

- Each element that was correctly categorized and styled added 1 point to the score.
- Each element that was correctly categorized but styled incorrectly added 0.25 points.
- Elements that were incorrectly categorized or missed entirely during the Element Detection process did not contribute to the score.

The total score was then divided by the maximum possible score (the expected number of elements in the image) and multiplied by 100 to obtain a percentage value.

It is important to note that these tests were performed on a temporary machine provided by the host company, so the recorded Conversion Time values are likely to improve with a dedicated cloud-based or on-premise solution. Table 7.1 summarizes the metrics for the reported test examples, and the average metrics are provided for both the reported subset and the full test set.

Test Subject	Conversion Time	Element Analysis Accuracy
"Process Edit" screen	28.25 s	85.94%
"Process Search" screen	40 s	94.64%
"Process Details" screen	43.03 s	84.3%
"Vehicle Tracking" screen	11.23 s	81.14%
Average (reported examples)	30.63 s	86.5%
Average (entire test set)	43.24 s	82.11%

Table 7.1: Overview of the achieved performance metrics during testing.

The results demonstrate that the tool is able to effectively assist programmers with common front-end development tasks. The average Conversion Time indicates significant potential for enhancing productivity, and it is expected to improve further with the adoption of a dedicated deployment solution. The Element Analysis Accuracy, however, is less satisfactory, with an average value of 82.11% achieved across the test images. The presence of errors introduces inefficiencies, requiring programmers to identify and correct problematic elements. Potential solutions to this issue will be discussed in Section 8.2.

Chapter 8

Conclusions

This chapter reflects on the completed thesis project, starting from the obtained results, and offers suggestions to address current limitations of the tool while improving its effectiveness in supporting users in their development activities. Given its experimental nature, the project underwent several iterations at both the conceptual and technical levels. It is, therefore, important to highlight the challenges encountered with the chosen solution and propose directions for future improvements in the project's development.

8.1 Advantages and Limitations of the Proposed Solution

During the final testing phase, the developed tool demonstrated itself as an effective solution for enhancing frontend developers' productivity. After analyzing the recorded metrics and discussing the test results with the host company, the following key points were identified:

- The average Conversion Time recorded during the tests, while improvable through a more powerful and reliable deployment solution, highlighted the tool's capability to reduce the time developers spend on repetitive tasks, thereby increasing their productivity and enabling them to focus on more complex and demanding activities.
- The adoption of the General Page Description as a language- and framework-agnostic representation format enhances the tool's versatility, making it applicable in a wide range of contexts and usable by different teams.
- The deterministic Translation Modules address the issue of unpredictability often associated with other LLM-based development tools, ensuring stylistic

consistency across projects.

- From a user's perspective, the workflow is straightforward, making the tool easy and intuitive to use.

However, several issues were identified that pertain to specific aspects of the workflow, which will require further refinement before the prototype can be considered a finished product.

A critical step in the application's workflow is the Element Analysis process. The final test results indicate that the adopted Large Language Model, GPT-4o, is not fully reliable, often producing partially or entirely inaccurate outputs in a significant percentage of cases. This limited accuracy stems from both the intrinsic limitations of this technology, which cannot guarantee consistent and reliable results, and the use of a general-purpose model that is not specifically tailored to the use case at hand.

Moreover, the analysis of all elements within an input image requires a significant number of token exchanges through the external LLM API, which can result in high operational costs. To mitigate these expenses in the long term, strategies should be implemented to reuse previously obtained information, minimizing redundancies in the Element Analysis phase.

Additionally, the application currently lacks functionality related to the grouping of related history entries. When managing multiple projects, a developer may need to access the generation history specific to one project and use a pre-configured workflow dedicated to it (e.g., automatic selection of the target platform for code generation). The final product would benefit from the inclusion of such functionality to efficiently assist developers in handling various types of activities simultaneously.

8.2 Possible Future Developments

Starting from the obtained prototype, with its previously discussed functionality and implementations, several possibilities for further enhancement and improvement arise.

8.2.1 Improving the Element Analysis Accuracy

The primary goal for future improvements of the application should be to enhance the accuracy of the Element Analysis step. To significantly boost productivity, it is crucial that the results contain as few errors as possible, minimizing the need for manual intervention by the programmer. Several potential approaches can be considered to achieve this.

One possible method involves a substantial restructuring of the Element Analysis process by employing a *divide et impera* strategy. Rather than directly providing a

Large Language Model with the image of the element, the analysis could be split into two stages:

1. An initial classification model examines the element's image to identify the type of UI element.

2. The image, along with the identified type, is then passed to a Large Language Model, which focuses solely on providing style information, assuming the type has been correctly identified.

By incorporating a dedicated classification model to identify the type of element, an increase in accuracy is expected. This is because, compared to the general-purpose MLLM which would later be used for extracting stylistic information, this component would be specifically designed for this task. Additionally, a classification model is both a faster and more cost-effective solution compared to a Multimodal Large Language Model. The reduced number of tokens exchanged through the LLM API would also contribute to lower operational costs.

The classification model would be implemented using a Convolutional Neural Network, designed to process images and assign them to a pre-defined set of types. A promising approach would involve fine-tuning an existing pretrained model. This would allow the benefits of the pretrained model's robustness and accuracy to be harnessed and adapted to the specific requirements of this task.

Several state-of-the-art pretrained models for image classification are currently available. One notable open-source option is ResNet, a family of models known for their high accuracy and adaptability to various classification tasks [73]. The ResNet (*Residual Network*) architecture employs a residual learning framework to tackle challenges encountered in training deep networks, particularly the vanishing gradient problem. Instead of each layer learning a completely new representation, ResNet learns the residuals - the difference between the input and the target output - using shortcut connections that bypass certain layers. These connections facilitate easier learning by ensuring the gradient flows smoothly through deeper layers [73].

An alternative approach to enhancing the accuracy of the Element Analysis process, without altering its structure, would involve utilizing a different Multimodal Large Language Model, fine-tuned to achieve the required performance levels for recognizing and describing UI elements. Many LLM providers offer users the option to fine-tune a foundational model through a technique called Low-Rank Adaptation (LoRA).

Low-Rank Adaptation is a technique used to fine-tune LLMs with reduced computational costs and memory requirements [74]. Instead of updating the entire set of model parameters, LoRA introduces trainable low-rank matrices that approximate the weight changes necessary for task-specific adaptation. This approach effectively reduces the number of parameters that need to be adjusted,

while preserving the model's general capabilities. By only modifying a subset of parameters, LoRA enables efficient adaptation to specific tasks without the need for extensive retraining or the storage of multiple model copies, making it a practical choice for task-specific fine-tuning [74].

As a practical example, the OpenAI Developer Platform offers fine-tuning for several leading LLMs, including GPT-4o [75]. While this process involves an initial financial investment, a fine-tuned model could significantly enhance performance, improving the quality of the generated output. Moreover, such a model would likely eliminate the need for a few-shot prompting technique, which would reduce the number of tokens exchanged and, consequently, lower costs over time.

8.2.2 Persisting Information to Reduce Costs

One of the primary limitations of Large Language Models is their inability to retain information from previous interactions. To provide context for a conversation, the entire content must be included as input for generating the next response. To address this issue, leading AI service providers have developed more advanced strategies to carry over learned information between interactions.

An example of such a solution was introduced by OpenAI in ChatGPT, which allows users to interact with models through a chat interface. In 2024, OpenAI introduced a feature called *memories*, which consists of facts learned over time during conversations with the user. These memories can be reused in future interactions, enabling more contextually aware and informed responses [76]. Other LLM providers, such as Anthropic, have also begun experimenting with similar features [53].

Although the technical details of these solutions have not been fully disclosed, it is evident that they are based on two key concepts:

- The user's past interactions with the Large Language Model are analyzed, and relevant facts are extracted and stored in a structured format.
- In subsequent interactions, the user's queries are analyzed, and relevant facts are retrieved from the stored information (e.g., using techniques such as Retrieval-Augmented Generation [77]) and incorporated into the context window.

By selectively adding only relevant information in a predefined format, the model is equipped with the necessary context to provide accurate responses. This approach eliminates the need for the user to repeatedly provide the same information, improving the overall experience and reducing the number of input tokens required.

In the context of the developed tool, this strategy cannot be applied directly; however, it provides a foundation for devising functionalities that allow the system to "remember" previously generated information and avoid redundancies.

A significant limitation of the current process is its inability to retain information about previously analyzed elements. This can result in considerable resource waste. For example, if two identical images are processed consecutively, they are treated as entirely separate inputs, and the analysis is repeated for each element, starting from scratch.

To address these inefficiencies, it is essential to store information about previously analyzed elements. This could be achieved by adding an additional table to the relational schema of the application's database, which would store key-value pairs representing the analyzed images and their corresponding JSON-formatted descriptions. This table, acting as an "Element Cache", would be consulted during each Element Analysis process to check whether the image has already been analyzed. If a match is found, the corresponding description can be retrieved directly, bypassing the need to involve the LLM.

Given this solution, the analysis process for a detected UI element would follow this structure:

- The image corresponding to the detected element is extracted.
- A lookup is performed in the dedicated table to determine whether the extracted image has already been processed:
  - If the image has been processed, the pre-generated description is retrieved and assigned to the element.
  - If not, the element is analyzed through the use of the MLLM.

One important detail that has been overlooked in the proposed solution is the need for a suitable representation of the elements' images in the dedicated table. A basic approach would involve treating the image as a BLOB field and storing all its binary content within the table. However, this method would be highly inefficient in terms of both storage usage and lookup costs.

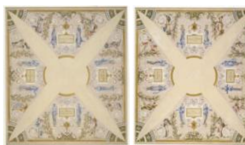
A more effective approach would be to use a hashing function on the element image, with the resulting hash serving as the primary key for the Element Cache. In this scenario, the selection of an appropriate hashing function is critical for the system's proper behavior. Traditional hashing methods may encounter issues with collisions, where unrelated images produce the same hash value. This could lead to some elements being assigned entirely inaccurate descriptions. To mitigate these inaccuracies, Perceptual Hashing can be employed.

Perceptual Hashing is a method used to create a concise, representative fingerprint of an image based on its visual characteristics. Unlike traditional hashing, which produces completely different outputs for even minor changes, perceptual hashing is designed to account for slight variations, such as differences in brightness or small distortions. This makes it useful for identifying images that share similar

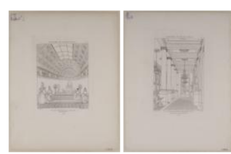
content. By generating a perceptual hash, the method captures essential visual features such as edges and textures.

An example of a Perceptual Hashing algorithm is the Average Hash (aHash); it works by reducing an image to capture its essential structure, allowing for quick comparisons between visually similar images. First, the image is resized to 8x8 pixels, removing high-frequency details while preserving its basic features. It is then converted to grayscale, simplifying the color information. The average color value of the 64 pixels is calculated, and each pixel is compared to this average. If a pixel is above the average, it is assigned a 1; if below, it is assigned a 0. These bits are combined into a 64-bit hash. The algorithm is robust to changes in scale, aspect ratio, brightness, and contrast, as similar images will produce similar hashes. The degree of similarity is measured by comparing the hashes using Hamming distance, with fewer differences indicating closer visual similarity. Figure 8.1 shows examples of clusters of images that share the same aHash.

Cluster 00423c3c3c3c5a00



Cluster 0042c3c3c3c7ffff



Cluster 00767e7e7e7e7c7c

Figure 8.1: Examples of consecutive image clusters with the same Average Hash.

In conclusion, a form of cross-generational *memory* can be implemented in the system through an **Element Cache**, which consists of a table that stores information about previously analyzed elements as key-value pairs. The key would correspond to the element image's Average Hash, while the value would consist of its LLM-generated JSON-formatted description. This description should exclude any references to the element's spatial properties, which are already available from the Element Detection phase. When a new element is analyzed, the Average Hash

for its corresponding image is computed, and if a match is found in the cache, the stored description is used, thereby avoiding the need for the LLM.

Furthermore, the hashing procedure can be utilized to further minimize LLM intervention by applying it to each element prior to the analysis process. Elements sharing the same Average Hash value can be grouped into clusters, with only one element subjected to the Element Analysis process. Once the description for this element is obtained - either through LLM-based generation or from the Element Cache - it can be applied to all remaining elements within the cluster. The revised analysis process that incorporates Average Hashing and Element Caching is illustrated with an example in Figure 8.2.

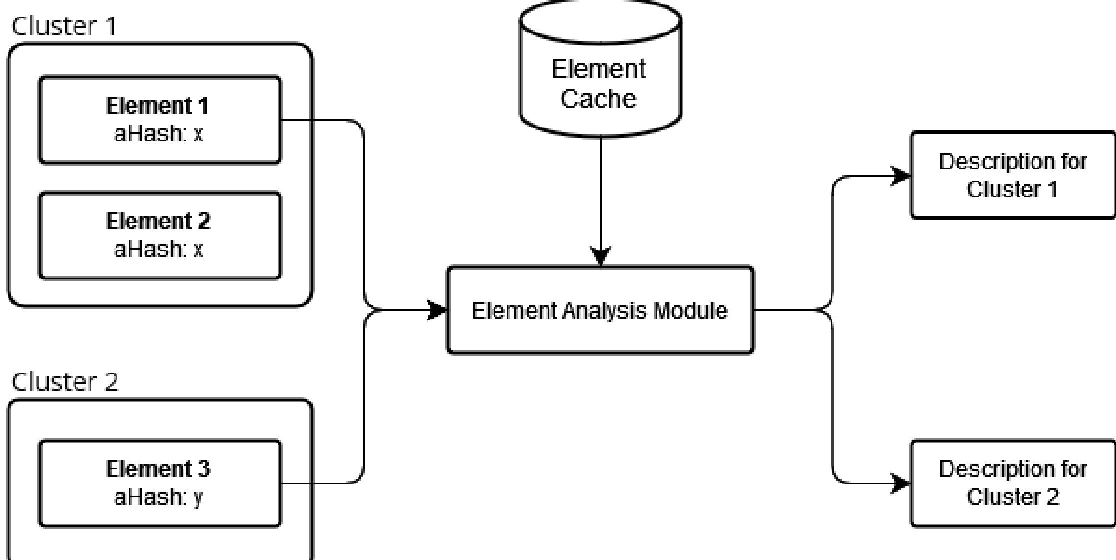


Figure 8.2: Flow chart showing an example of the revised Element Analysis process.

8.2.3 Organizing Image Conversions into Projects

An additional functionality that would significantly enhance the quality of the tool's user experience is the ability to track multiple development projects. This would allow users to define custom default settings for the image analysis and corresponding code translation processes.

Organizing the user's activities into different projects could prove highly beneficial, enabling effective work organization and easy access to previously generated code based on their current activities. Additionally, projects would offer opportunities to automate certain aspects of the workflow:

- A project could be assigned a Translation Module, which would be used automatically to generate the desired code without requiring an explicit choice from the user.
- Projects could also be linked to shared repositories and utilize Version Control Systems to automatically push newly generated code to them.

Furthermore, this functionality could be expanded to promote collaboration among developers. Shared projects could be made accessible to multiple users, allowing each member to access code generated by all project participants, as well as the predefined settings that would enable them to effectively use the tool from the outset.

Overall, the introduction of **Shared Development Projects** could enhance the user experience, facilitate collaboration among team members, and further streamline the development workflow through integration with external systems. While this feature may increase the application's complexity, it represents an important step toward creating a more useful and comprehensive tool.

Bibliography

[1] GitHub. 2023. URL: `https://github.com/features/copilot` (cit. on p. 2).

[2] OpenAI. 2022. URL: `https://openai.com/index/chatgpt/` (cit. on p. 2).

[3] Stack Overflow. 2024. URL: `https://survey.stackoverflow.co/2024` (cit. on p. 2).

[4] OpenAI et al. *GPT-4 Technical Report*. 2024. arXiv: `2303.08774 [cs.CL]`. URL: `https://arxiv.org/abs/2303.08774` (cit. on pp. 3, 49).

[5] Gemini Team et al. *Gemini: A Family of Highly Capable Multimodal Models*. 2024. arXiv: `2312.11805 [cs.CL]`. URL: `https://arxiv.org/abs/2312.11805` (cit. on pp. 3, 49).

[6] Tony Beltramelli. *pix2code: Generating Code from a Graphical User Interface Screenshot*. 2017. arXiv: `1705.07962 [cs.LG]`. URL: `https://arxiv.org/abs/1705.07962` (cit. on pp. 3, 6, 7).

[7] Alex Robinson. *Sketch2code: Generating a website from a paper mockup*. 2019. arXiv: `1905.13750 [cs.CV]`. URL: `https://arxiv.org/abs/1905.13750` (cit. on pp. 3, 7, 8).

[8] Gilles Baechler et al. *ScreenAI: A Vision-Language Model for UI and Infographics Understanding*. 2024. arXiv: `2402.04615 [cs.CV]`. URL: `https://arxiv.org/abs/2402.04615` (cit. on p. 7).

[9] Kenton Lee et al. *Pix2Struct: Screenshot Parsing as Pretraining for Visual Language Understanding*. 2023. arXiv: `2210.03347 [cs.CL]`. URL: `https://arxiv.org/abs/2210.03347` (cit. on p. 7).

[10] Angular Team. *Home - Angular*. Accessed on August 7, 2024. URL: `https://angular.dev/` (cit. on pp. 11, 13, 71).

[11] Gavin Bierman, Martín Abadi, and Mads Torgersen. «Understanding TypeScript». In: *ECOOP 2014 – Object-Oriented Programming*. Ed. by Richard Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 257–281. ISBN: 978-3-662-44202-9 (cit. on p. 11).

[12] PrimeNG Team. *PrimeNG - Angular UI Component Library*. Accessed on August 7, 2024. URL: `https://primeng.org/` (cit. on pp. 11, 71).

[13] Bootstrap Team. *Bootstrap - The most popular HTML, CSS and JS library in the world*. Accessed on August 7, 2024. URL: `https://getbootstrap.com/` (cit. on pp. 12, 13).

[14] Majida Laaziri, Khaoula Benmoussa, Khoulji Samira, Kerkeb Larbi, and Abir Yamami. «Analyzing bootsrap and foundation font-end frameworks: a comparative study». In: *International Journal of Electrical and Computer Engineering* 9 (Feb. 2019), pp. 713–722. DOI: `10.11591/ijece.v9i1.pp713-722` (cit. on p. 12).

[15] Flask Team. *Welcome to Flask - Flask Documentation*. Accessed on August 7, 2024. URL: `https://flask.palletsprojects.com/en/3.0.x/#` (cit. on p. 13).

[16] E Hildreth and D Marr. «Theory of edge detection». In: *Proceedings of Royal Society of London* 207.1167 (1980), pp. 187–217 (cit. on pp. 14, 15).

[17] Tony Lindeberg. «Edge detection». In: *Encyclopedia of Mathematics* (2011) (cit. on p. 15).

[18] R.C. Gonzalez and R.E. Woods. *Digital Image Processing*. Prentice Hall, 2008. ISBN: 9780131687288. URL: `https://books.google.it/books?id=8uGOnjRGEzoC` (cit. on pp. 16, 23).

[19] Lawrence Roberts. «Machine Perception of Three-Dimensional Solids». PhD thesis. Massachusetts Institute of Technology, 1963. ISBN: 0-8240-4427-4 (cit. on p. 16).

[20] Irwin Sobel. «An Isotropic 3x3 Image Gradient Operator». In: *Presentation at Stanford A.I. Project 1968* (Feb. 2014) (cit. on p. 17).

[21] Manoj K Vairalkar and SU Nimbhorkar. «Edge detection of images using Sobel operator». In: *International Journal of Emerging Technology and Advanced Engineering* 2.1 (2012), pp. 291–293 (cit. on p. 17).

[22] Radhika Chandwadkar, Saurabh Dhole, Vaibhav Gadewar, Deepika Raut, and SA Tiwaskar. «Comparison of edge detection techniques». In: *6th Annual Conference of IRAJ*. Vol. 8. 2013 (cit. on p. 17).

[23] R. Fisher, S. Perkins, A. Walker, and E. Wolfart. *Spatial Filters - Laplacian/Laplacian of Gaussian*. Accessed on August 13, 2024. URL: `https://homepages.inf.ed.ac.uk/rbf/HIPR2/log.htm` (cit. on pp. 17–19).

[24] L Assirati, N R Silva, L Berton, A A Lopes, and O M Bruno. «Performing edge detection by Difference of Gaussians using q-Gaussian kernels». In: *Journal of Physics: Conference Series* 490 (Mar. 2014), p. 012020. ISSN: 1742-6596. DOI: 10.1088/1742-6596/490/1/012020. URL: http://dx.doi.org/10.1088/1742-6596/490/1/012020 (cit. on p. 18).

[25] John Canny. «A Computational Approach to Edge Detection». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8.6 (1986). DOI: 10.1109/TPAMI.1986.4767851 (cit. on pp. 19, 21–23).

[26] R. Fisher, S. Perkins, A. Walker, and E. Wolfart. *Spatial Filters - Gaussian Smoothing*. Accessed on August 13, 2024. URL: https://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm (cit. on p. 20).

[27] Muthukrishnan R. «Edge Detection Techniques For Image Segmentation». In: *International journal of computer science and information technology* (Dec. 2011). DOI: 10.5121/ijcsit (cit. on p. 21).

[28] OpenCV Team. *OpenCV: Canny Edge Detection*. Accessed on August 14, 2024. URL: https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html (cit. on p. 22).

[29] Stephen M Pizer, E Philip Amburn, John D Austin, Robert Cromartie, Ari Geselowitz, Trey Greer, Bart ter Haar Romeny, John B Zimmerman, and Karel Zuiderveld. «Adaptive histogram equalization and its variations». In: *Computer vision, graphics, and image processing* (1987) (cit. on p. 23).

[30] R.G. Casey and E. Lecolinet. «A survey of methods and strategies in character segmentation». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 18.7 (1996), pp. 690–706. DOI: 10.1109/34.506792 (cit. on p. 25).

[31] Jeffrey L Elman. «Finding structure in time». In: *Cognitive science* 14.2 (1990), pp. 179–211 (cit. on p. 26).

[32] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. «Speech recognition with deep recurrent neural networks». In: *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE. 2013, pp. 6645–6649 (cit. on p. 26).

[33] Sepp Hochreiter and Jürgen Schmidhuber. «Long short-term memory». In: *Neural computation* 9.8 (1997), pp. 1735–1780 (cit. on pp. 26, 43).

[34] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. «Learning to forget: Continual prediction with LSTM». In: *Neural computation* 12.10 (2000), pp. 2451–2471 (cit. on p. 26).

[35] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. «Gradient-based learning applied to document recognition». In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324 (cit. on p. 27).

[36] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. «Imagenet classification with deep convolutional neural networks». In: *Advances in neural information processing systems* 25 (2012) (cit. on p. 27).

[37] Anthony Kay. *Tesseract: an Open-Source Optical Character Recognition Engine*. Accessed on August 14, 2024. URL: `https://www.linuxjournal.com/article/9676` (cit. on p. 27).

[38] Akhil S. *Overview of Tesseract OCR engine*. Dec. 2016 (cit. on p. 27).

[39] Nobuyuki Otsu et al. «A threshold selection method from gray-level histograms». In: *Automatica* 11.285-296 (1975), pp. 23–27 (cit. on p. 27).

[40] Ray Smith. «An overview of the Tesseract OCR engine». In: *Ninth international conference on document analysis and recognition (ICDAR 2007)*. Vol. 2. IEEE. 2007, pp. 629–633 (cit. on pp. 28, 39).

[41] Tesseract Team. *Tesseract: Is there a Minimum / Maximum Text Size?* Accessed on August 14, 2024. URL: `https://github.com/tesseract-ocr/tessdoc/blob/main/tess3/FAQ-Old.md#is-there-a-minimum--maximum-text-size-it-wont-read-screen-text` (cit. on p. 29).

[42] Satoshi Suzuki et al. «Topological structural analysis of digitized binary images by border following». In: *Computer vision, graphics, and image processing* 30.1 (1985), pp. 32–46 (cit. on pp. 31–33).

[43] Gary Bradski. «The OpenCV Library». In: *Dr. Dobb's Journal: Software Tools for the Professional Programmer* 25.11 (2000), pp. 120–123 (cit. on p. 39).

[44] Adrian Kaehler and Gary Bradski. *Learning OpenCV 3: computer vision in C++ with the OpenCV library.* " O'Reilly Media, Inc.", 2016 (cit. on p. 39).

[45] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library.* " O'Reilly Media, Inc.", 2008 (cit. on p. 39).

[46] Joseph Weizenbaum. «ELIZA—a computer program for the study of natural language communication between man and machine». In: *Communications of the ACM* 9.1 (1966), pp. 36–45 (cit. on p. 43).

[47] Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. «Indexing by latent semantic analysis». In: *Journal of the American society for information science* 41.6 (1990), pp. 391–407 (cit. on p. 43).

[48] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. «Learning representations by back-propagating errors». In: *nature* 323.6088 (1986), pp. 533–536 (cit. on p. 43).

[49] Ashish Vaswani. «Attention is all you need». In: *arXiv preprint arXiv:1706.03762* (2017) (cit. on pp. 43–45).

[50] Arne Bewersdorff, Christian Hartmann, Marie Hornberger, Kathrin Seßler, Maria Bannert, Enkelejda Kasneci, Gjergji Kasneci, Xiaoming Zhai, and Claudia Nerdel. «Taking the next step with generative artificial intelligence: The transformative role of multimodal large language models in science education». In: *arXiv preprint arXiv:2401.00832* (2024) (cit. on pp. 47, 48).

[51] Alec Radford et al. «Learning transferable visual models from natural language supervision». In: *International conference on machine learning*. PMLR. 2021, pp. 8748–8763 (cit. on p. 47).

[52] Jiayang Wu, Wensheng Gan, Zefeng Chen, Shicheng Wan, and S Yu Philip. «Multimodal large language models: A survey». In: *2023 IEEE International Conference on Big Data (BigData)*. IEEE. 2023, pp. 2247–2256 (cit. on p. 47).

[53] Anthropic. *Introducing Claude 3.5 Anthropic*. Accessed on August 18, 2024. URL: https://www.anthropic.com/news/claude-3-5-sonnet (cit. on pp. 49, 95).

[54] Meta AI. *Introducing Meta Llama 3: The most capable openly available LLM to date*. Accessed on August 18, 2024. URL: https://ai.meta.com/blog/meta-llama-3/ (cit. on p. 49).

[55] Albert Q Jiang et al. «Mistral 7B». In: *arXiv preprint arXiv:2310.06825* (2023) (cit. on p. 49).

[56] Sakib Shahriar, Brady Lund, Nishith Reddy Mannuru, Muhammad Arbab Arshad, Kadhim Hayawi, Ravi Varma Kumar Bevara, Aashrith Mannuru, and Laiba Batool. *Putting GPT-4o to the Sword: A Comprehensive Evaluation of Language, Vision, Speech, and Multimodal Proficiency*. 2024. arXiv: 2407.09519 [cs.AI]. URL: https://arxiv.org/abs/2407.09519 (cit. on pp. 49, 50).

[57] OpenAI. *API Referencee - OpenAI API*. Accessed on August 18, 2024. URL: https://platform.openai.com/docs/api-reference/introduction (cit. on pp. 50, 56, 57, 72).

[58] Sander Schulhoff et al. *The Prompt Report: A Systematic Survey of Prompting Techniques*. 2024. arXiv: 2406.06608 [cs.CL]. URL: https://arxiv.org/abs/2406.06608 (cit. on pp. 51, 52).

[59] Mozilla. *Basic concepts of flexbox - CSS: Cascading Style Sheets*. Accessed on August 19, 2024. URL: https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_flexible_box_layout/Basic_concepts_of_flexbox (cit. on p. 60).

[60] Mozilla. *Basic concepts of grid layout - CSS: Cascading Style Sheets*. Accessed on August 19, 2024. URL: https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_grid_layout/Basic_concepts_of_grid_layout (cit. on pp. 60, 61).

[61] Erich Gamma. *Design patterns elements of reusable object-oriented software*. TPB, 2006 (cit. on p. 66).

[62] Mark Lutz. *Learning python: Powerful object-oriented programming.* " O'Reilly Media, Inc.", 2013 (cit. on p. 67).

[63] Edsger W Dijkstra. «Recursive programming». In: *Numerische Mathematik* 2.1 (1960), pp. 312–318 (cit. on p. 68).

[64] Olivia Gallucci. *Recursion: Algorithmic Paradigms, Complexities and Pitfalls*. Accessed on August 22, 2024. URL: https://oliviagallucci.com/recursion-algorithmic-paradigms-complexities-and-pitfalls/#pitfalls-of-recursion (cit. on pp. 68, 69).

[65] Santosh Kumar. «A Review on Client-Server based applications and research opportunity». In: *International Journal of Recent Scientific Research* 10.7 (2019), pp. 33857–3386 (cit. on p. 72).

[66] Roy Thomas Fielding and Richard N. Taylor. «Architectural styles and the design of network-based software architectures». AAI9980887. PhD thesis. University of California, Irvine, 2000. ISBN: 0599871180 (cit. on p. 72).

[67] SQLite Team. *About SQLite*. Accessed on September 2, 2024. URL: https://www.sqlite.org/about.html (cit. on p. 73).

[68] Kevin P Gaffney, Martin Prammer, Larry Brasfield, D Richard Hipp, Dan Kennedy, and Jignesh M Patel. «Sqlite: past, present, and future». In: *Proceedings of the VLDB Endowment* 15.12 (2022) (cit. on pp. 73, 74).

[69] Devndra Ghimire. *Comparative study on Python web frameworks: Flask and Django*. 2020 (cit. on p. 75).

[70] SQLAlchemy Team. *Overview - SQLAlchemy 2.0 Documentation*. Accessed on September 2, 2024. URL: https://docs.sqlalchemy.org/en/20/intro.html (cit. on pp. 75, 76).

[71] Madhuri A Jadhav, Balkrishna R Sawant, and Anushree Deshmukh. «Single page application using angularjs». In: *International Journal of Computer Science and Information Technologies* 6.3 (2015), pp. 2876–2879 (cit. on p. 76).

[72] Saleema Amershi et al. «Guidelines for Human-AI Interaction». In: *CHI 2019*. CHI 2019 Honorable Mention Award. ACM. May 2019. URL: https://www.microsoft.com/en-us/research/publication/guidelines-for-human-ai-interaction/ (cit. on pp. 81, 83).

[73] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV]. URL: https://arxiv.org/abs/1512.03385 (cit. on p. 94).

[74] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021. arXiv: 2106.09685 [cs.CL]. URL: https://arxiv.org/abs/2106.09685 (cit. on pp. 94, 95).

[75] OpenAI. *Fine-tuning - OpenAI API*. Accessed on September 26, 2024. URL: https://platform.openai.com/docs/guides/fine-tuning (cit. on p. 95).

[76] OpenAI. *Memory and new controls for ChatGPT*. Accessed on September 26, 2024. URL: https://openai.com/index/memory-and-new-controls-for-chatgpt/ (cit. on p. 95).

[77] Patrick Lewis et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. 2021. arXiv: 2005.11401 [cs.CL]. URL: https://arxiv.org/abs/2005.11401 (cit. on p. 95).