Use of Visual Content for Inference and Response in Q&A Forums

FAIZ AHMED

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

YORK UNIVERSITY
TORONTO, ONTARIO

May 2025

Abstract

This thesis investigates the intersection of visual communication and programming knowledge sharing on Q&A platforms like Stack Overflow. Developers have shown increased use of screenshots to communicate technical problems, due to ease and content relevance of these screenshots. Through four interconnected studies, we explore how image processing and emerging AI technologies can help assist users by utilizing these screenshots. First, we examined image processing for duplicate question detection on Stack Overflow. Second, we evaluated how LLMs can interpret programming screenshots and generate relevant questions. Third, we developed *CORTEx*, a novel method for extracting and preserving structured text from programming screenshots. Fourth, we assessed multimodal LLMs' capabilities in identifying and extracting relevant content from screenshots. This research highlights the gap between developers' visual communication preferences and current system capabilities, exploring the potential of image processing and AI to address this divide and make visual programming content more accessible and usable.

Acknowledgements

I acknowledge with deep appreciation the invaluable contributions of many individuals who made this thesis possible.

First and foremost, I extend my sincere gratitude to my supervisors, Prof. Maleknaz Nayebi and Prof. Suprakash Datta, for their exceptional guidance, scholarly expertise, and unwavering support throughout my graduate studies. Their mentorship has been instrumental in shaping this research and fostering my academic development.

I am profoundly grateful to my family and friends for their steadfast encouragement and understanding during this academic endeavor. Their support provided the foundation that enabled me to pursue and complete this work.

I acknowledge the department for providing the necessary research infrastructure, resources, and academic environment that facilitated this thesis. The institutional support has been essential to the successful completion of this project.

Finally, I thank my lab colleagues for their collaborative spirit, intellectual contributions, and professional camaraderie. Their dedication to research excellence and willingness to share knowledge and insights have significantly enhanced my graduate experience and academic growth.

Table of Contents

List of Figures

List of Tables

Abbreviations & Acronyms

AI	**A**rtificial **I**ntelligence
SO	**S**tack **O**verflow
CORTEx	**CO**n**T**ext aware **R**ecognition from **T**ext and cod**E**
IDE	**I**ntegrated **D**evelopment **E**nvironment
LLM	**L**arge **L**anguage **M**odel
NLP	**N**atural **L**anguage **P**rocessing
OCR	**O**ptical **C**haracter **R**ecognition
Q&A	**Q**uestion **&** **A**nswer
RQ	**R**esearch **Q**uestion

Chapter 1

Introduction

Software development has evolved into an intensely visual discipline [40, 185], with developers spending countless hours interacting with complex integrated development environments (IDEs), interpreting error messages, and debugging multi-layered code structures [22, 159]. While programming languages themselves remain fundamentally textual, the process of creating, understanding, and troubleshooting code increasingly relies on visual elements [84, 12] ranging from color-coded syntax highlighting to graphical debuggers and data visualizations. This visual dimension of programming shapes not only how developers work but also how they communicate about code-related problems.

The visual nature of programming is reflected in how developers seek help when encountering technical challenges. A study by Nayebi and Adams [113, 111] showed that the number of images shared on Stack Overflow doubled between 2013 and 2022, with 69.4% of these images depicting code or terminal outputs. `ImageR` achieved an F1-score of 0.76 in determining when images are needed, with 75% of users finding the recommendations highly valuable [174]. Additionally, 75% of the developers surveyed in `ImageR`'s study found its overall design and recommendations to be practically useful in enhancing bug report clarity and communication efficiency [173].

Stack Overflow has emerged as the leading platform for programming questions and answers, with more than 20 million registered users and more than 24 million questions as of 2022 [165]. The platform is a crucial resource for developers seeking solutions

to programming challenges, offering a vast repository of knowledge on topics ranging from basic syntax questions to complex architectural decisions [203, 74].

While Stack Overflow was designed primarily for text-based content, the platform has seen a significant increase in image usage over time. However, Stack Overflow's guidelines explicitly discourage posting images of code, errors, or data, advising users to include textual representations instead [167]. Genuine concerns drive these guidelines: text can be copied, edited, and searched, while images cannot. Text-based content is accessible to users with visual impairments who rely on screen readers, whereas images may present accessibility barriers [28].

Despite these guidelines, developers continue to post screenshots of code and error messages at increasing rates [113, 111, 112]. This persistent behavior suggests images fulfill essential needs that text alone cannot satisfy. Furthermore, Kuramoto et. al. [86] reported that images or videos have a significantly higher probability of reporting bugs, 77.10% compared to 44.60% in reports without visual content. This trend indicates that images or screenshots are vital in these coding communities.

Even with the rise of use of images on Stack Overflow, it is unclear how these images can help the developer community and the users on this platform. To investigate this, we followed a number of objectives (Obj):

Obj 1: Investigate whether images can reduce bias in the current design of software automation by investigating the case of duplicate post detection on Stack Overflow.
This is addressed in Chapter 2.

Obj 2: Evaluate LLMs' ability to identify and highlight problematic code segments in screenshots, along with inferring questions.
This is the target of Chapter 3.

Obj 3: Investigate methods for enhancing image processing techniques when reading screenshots to better extract text and code compared to LLMs and state-of-the-art methods.
This is discussed in Chapter 4.

The following sections provide an overview of each objective.

1.1 Obj 1: Automating Duplicate Post Detection using Screenshots

Despite the increased use of images in Software Development platforms, automation tasks in social coding platforms remain mainly textual, and there is minimal use of screenshots [30, 113]. Studies show that visual communication plays an increasingly important role in software development processes [194].

To investigate this further, Chapter 2 of the thesis investigates the potential of image processing and the screenshots attached to the posts for identifying duplicate questions on Stack Overflow. Previous studies automated this approach but were heavily reliant on textual content of the posts [4, 208, 172]. By expanding upon the work of Ahasanuzzaman et. al. [4], we explore whether incorporating visual information from screenshots can enhance duplicate detection beyond traditional text-based approaches.

To achieve this, we employed Optical Character Recognition(OCR) to capture the textual content and image captions using `gemini-1.0-pro-vision` to capture the semantics of the image attached. Our findings show that while images provide complementary information to text, current image processing techniques yield only modest improvements (around 1-2%) in duplicate detection accuracy. This work establishes important baselines and identifies challenges in leveraging image content for this task, contributing valuable negative results to the research community.

1.2 Obj 2: Highlighting Problematic Code and Inferring Questions using Screenshots

The widespread adoption of visual communication across social media platforms reflects a fundamental shift in how users interact online. Studies [176, 68] show that there are approximately 1.3 billion users on Instagram and 1 billion users on TikTok. This

trend has naturally extended to software engineering, where the context from different IDE panels and the specific positioning of code elements can provide crucial information for understanding developer workflows and intent [43, 85]. The visual layout and elements in programming screenshots can convey information that may be lost in text-only descriptions, making them valuable resources for technical problem-solving.

Building upon advances in multimodal AI capabilities [144, 88], Chapter 3 explores whether large language models can effectively infer meaningful questions from programming screenshots alone and investigates their ability to identify and extract relevant content, specifically focusing on highlighting problematic code segments. Previous research has primarily focused on using images as supplementary content rather than primary input for technical problem-solving [111, 5].

Our work explores the feasibility of screenshot-only query interpretation by leveraging prompt engineering techniques including in-context learning, chain-of-thought prompting, and few-shot learning to guide LLM models interpretation. Models like `GPT-4o`[65], `Gemini-1.5-pro`[175], and `LLAMA-3.2`[179] have demonstrated impressive capabilities in understanding and reasoning about visual content, including programming screenshots[33, 66]. Our comparative analysis evaluates these three state-of-the-art models to precisely locate and extract specific text segments that relate to underlying programming issues, enabling more focused assistance.

Our findings reveal significant potential and current limitations in screenshot-based programming assistance. For question inference, `GPT-4o` achieves the strongest performance, generating questions with over 60.00% similarity to original posts for 51.75% of tested images, while `GPT-4o` and `Gemini` both achieve developer relevance scores of 0.67-0.68. For content extraction and problematic code identification, `Gemini` achieved the highest F1 score of 0.91 for relevance detection, while `GPT-4o` demonstrated superior text extraction quality with BLEU scores of 0.46 and ROUGE-1 scores of 0.68 when assisted by OCR. However, performance varies significantly based on screenshot complexity, with models excelling on clear, focused content but struggling with multi-window or complex visual inputs. This work establishes important benchmarks for

multimodal AI in developer support, highlighting both the potential and current limitations of screenshot-based programming assistance tools.

1.3 Obj 3: Parsing Screenshots using Enhanced Image Processing Techniques

Recent advances in Large Language Models (LLMs) demonstrate significant potential for bridging visual and textual modalities in programming contexts. Several systems leverage LLMs to generate accurate HTML and CSS from UI screenshots [205, 48], while frameworks like Low-code LLM enable visual programming operations without extensive prompt engineering [33]. This convergence of visual understanding, text generation, and code interpretation in modern LLMs creates unprecedented opportunities for automated Q&A forum assistance [108, 38].

However, despite these advances, current LLMs face critical limitations when processing code-related images from developer forums. State-of-the-art models like `GPT-4o` and `Gemini-1.5-pro` frequently hallucinate non-existent code or omit crucial visible text, creating unreliable extraction results. Traditional OCR systems struggle with code formatting preservation, while specialized methods like `CodeMotion` [77] and `PSC2Code` [19] over-segment static images, producing fragmented outputs unsuitable for Stack Overflow's requirements as they are mainly based programming screencasts and not static screenshots.

To address these limitations, we introduce *CORTEx* (COntext aware Recognition from Text and codE) in Chapter 4, a novel methodology specifically engineered for processing code-related images on Q&A platforms. *CORTEx* employs a five-stage pipeline combining intelligent image segmentation, advanced OCR, and context-aware layout analysis to maintain semantic integrity. Our evaluation demonstrates *CORTEx*'s superior performance:around 15.00% improvement in BLEU and ROUGE scores over existing methods and LLMs, with around 70.00% of developers preferring *CORTEx*'s layout

segmentation over state-of-the-art alternatives. This capability represents a critical step as it helps tackle Stack Overflow's guidelines to not post coding screenshots [168].

1.4 Thesis Overview

This work builds upon the existing body of literature developed over the years [81, 5, 158, 156, 82, 114, 111, 58, 130, 124, 109, 110, 157, 142, 6, 133, 134, 115, 129, 120, 116, 125, 128, 121, 96, 127, 122, 152, 126, 118, 117, 131, 153, 119, 132, 73, 123, 6, 120, 174, 177]. This thesis is manuscript-based and consists of three papers:

The first paper, discussed in **Chapter 2** investigates image-based duplicate detection on Stack Overflow, finding modest improvements (1-2%) over text-based approaches while establishing important baselines.

While, **Chapter 3** presents the second paper, that extends question inference from our MSR paper [6] that addresses question inferring to precise content identification and extraction, with `Gemini` achieving 0.91 F1 for relevance detection and `GPT-4o` demonstrating superior extraction quality.

Lastly, the third paper, discussed in **Chapter 4**, introduces *CORTEx*, a novel method for extracting text from programming screenshots that outperforms existing OCR systems by 15.00% while preserving critical code structure.

Chapter 5 synthesizes key findings and discusses future research directions.

Chapter 2

Negative Results of Image Processing For Identifying Duplicate Questions on Stack Overflow

Abstract

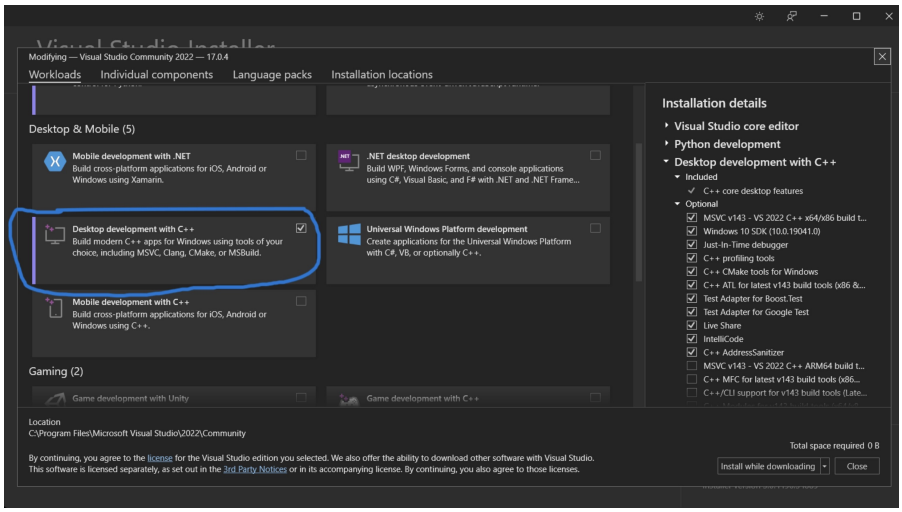
In the rapidly evolving landscape of developer communities, Q&A platforms serve as crucial resources for crowdsourcing developers' knowledge. A notable trend is the increasing use of images to convey complex queries more effectively. However, the current state-of-the-art method of duplicate question detection has not kept pace with this shift, which predominantly concentrates on text-based analysis. Inspired by advancements in image processing and numerous studies in software engineering illustrating the promising future of image-based communication on social coding platforms, we delved into image-based techniques for identifying duplicate questions on Stack Overflow. When focusing solely on text analysis of Stack Overflow questions and omitting the use of images, our automated models overlook a significant aspect of the question. Previous research has demonstrated the complementary nature of images to text. To address this, we implemented two methods of image analysis: first, integrating the text from images into the question text, and second, evaluating the images based on

their visual content using image captions. After a rigorous evaluation of our model, it became evident that the efficiency improvements achieved were relatively modest, approximately an average of 1%. This marginal enhancement falls short of what could be deemed a substantial impact. As an encouraging aspect, our work lays the foundation for easy replication and hypothesis validation, allowing future research to build upon our approach and explore novel solutions for more effective image-driven duplicate question detection.

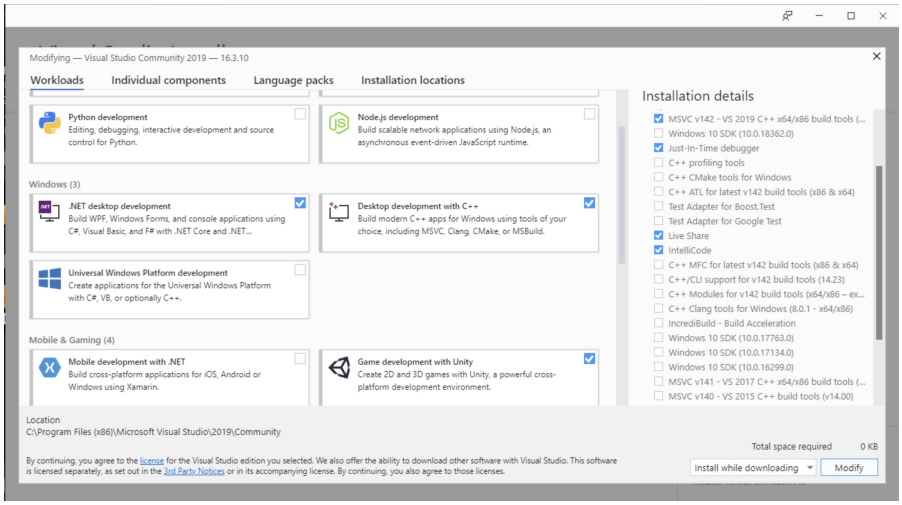
2.1 Introduction

Community-driven question-answering platforms have gained traction for crowdsourcing information from software developers, thanks to the vast amount of data they accumulate through active user involvement. While various platforms cover a wide range of topics, such as Yahoo Answers and Quora, Stack Overflow stands out for its specific focus on software development. Since its inception in 2008, Stack Overflow has become a crucial platform for addressing programming issues. As of March 2022, Stack Overflow boasted over 20 million registered users, with over 24 million questions and 35 million answers [164].

To avoid bloating and better structure the forums, Stack Overflow encourages developers to thoroughly review existing posts before submitting a new question. This practice is designed to prevent the recurrence of queries that have already been raised and potentially resolved.[166] Despite these proactive measures, developers occasionally submit duplicate questions. In this context, duplicate questions refer to inquiries seeking solutions to identical problems. When two questions are identified as duplicates, one is flagged as such and undergoes the closure process, while the other is designated as the master. Experienced community members or users can cast votes to identify duplicates, and when a question garners enough of these votes, it will be closed as a duplicate [166]. Within the research community, to automate this process, some research efforts have been directed toward the identification of duplicate questions, emphasizing textual analysis and Q&A thread interactions [191, 208, 161]. Notably, an examination



(A) Image attached to a Master question of our interest on Stack Overflow



(B) Image attached to a duplicate question similar to our question of interest

FIGURE 2.1: Both the images are very similar even though they are attached to 2 different questions.

in a study conducted by Ahasanuzzaman et al. [4] showed that exact duplicates are infrequent; however, many questions are duplicated but phrased differently.

Fueled by advancements in Machine Learning, software engineers are increasingly interested in the role of images within developer communities, as evidenced by Nayebi et al. [111]. This has led to a surge of studies mirroring observations in social coding platforms, highlighting developers' growing preference for image-based communication. Platforms like Stack Overflow, a popular question-and-answer forum, have seen a

rise in image usage, as documented by Nayebi et al. [111, 112] and Wang et al. [186].

On the other hand, a qualitative survey with software developers showed the complementary nature of these images to the textual information on bug repositories [111]. Often, the images are essential in understanding the change requests, questions, or responses submitted. Images help developers to convey complex information more effectively and efficiently than text alone. The study also pointed out that the majority of the images attached to questions were informative, and developers considered around 87.0% of the text unlikely or very unlikely to comprehend without the image attached to the question [111, 112]. They recognized that decision support systems that overlook image data are inherently biased, as they fail to consider the complementary nature of visual information. Furthermore, they anticipated that advancements in image processing and machine learning would soon surpass the capabilities of systems that rely solely on text-based data on social coding platforms.

Drawing on the findings by Wang et al. [187] and Nayebi and Adams [112], which highlight the complementary nature of images shared alongside text, we propose that image processing can significantly enhance state-of-the-art duplicate detection. The reported 200% increase in the number of images shared on Stack Overflow [111] has motivated us to investigate whether the information in these images can lead to more accurate predictions of duplicate questions. Currently, automated methods for detecting duplicate content do not consider image information in their decision-making processes, prompting us to hypothesize that such decisions, based solely on partial information, are inherently flawed. Consider, for instance, Figure 2.1, which displays two screenshots related to a bug fix. The visual similarities between these screenshots suggest that the associated questions might pertain to the same issue despite differences in their textual content.

Current text-based duplicate detection systems solely analyze question text, potentially overlooking valuable information embedded in attached images. This overreliance on text can lead to biased and incomplete decisions, as crucial visual details might be missed that could be instrumental in determining question similarity. Our

work aims to address this limitation by incorporating image data into the duplicate question identification process and assessing whether the current information systems lean toward biased decisions. By leveraging both textual and visual content, we aim to achieve more comprehensive and accurate duplicate detection on Stack Overflow. Specifically, we put forth the following research questions (RQs) for investigation:

RQ1: To what extent does the text extracted from images attached in Stack Overflow questions enhance the performance of state-of-the-art models for detecting duplicate questions?

What and How: The use of images in identifying redundant issue reports on Stack Overflow has the potential to enhance the accuracy of duplicate detection. Building upon past studies that indicate the complementary nature of images to text on Stack Overflow [111], we incorporate information from these images to evaluate whether it can enhance the performance of current methods for identifying duplicate Q&As. We use Optical Character Recognition (OCR) to this end [105, 37].

RQ2: Can incorporating the semantic meaning conveyed by image captions, alongside the existing focus on question text, significantly enhance the accuracy of duplicate question detection models on platforms like Stack Overflow?

What and How: We investigate how image information improves duplicate question detection in Stack Overflow. By generating captions for attached images using Google's

`gemini-1.0-pro-vision` model and incorporating them with textual features, we aim to create a richer representation for duplicate question identification. This approach leverages the complementary nature of image and text data to potentially enhance the accuracy of existing duplicate detection methods.[39]

RQ3: To what extent can combining image captions and Optical Character Recognition (OCR) for images in Stack Overflow questions improve the performance of duplicate question detection models?

What and How: Expanding on **RQ1** and **RQ2**, we investigate how image data combined with image text can improve duplicate question detection on Stack Overflow. We explore using image captions (via `gemini-1.0-pro-vision`) and OCR (Nanonets) to extract text from images. Both extracted text and captions are incorporated with existing textual features. This evaluation helps us understand the effectiveness of image data for enhancing duplicate question detection accuracy.

Our study focuses on enhancing the current state-of-the-art in detecting duplicate questions on Stack Overflow by incorporating images, extending beyond existing methods that rely solely on text from the questions for duplicate detection. In this chapter, we build upon the well-cited approach of Ahsanuzzaman et al. [4], incorporating title, body description, tag similarity, and code to define and improve duplicate question detection. Our expansion involves incorporating image analysis to address **RQ1**, **RQ2**, and **RQ3**.

In what follows, we start by explaining our baseline study in Section 2.2. We then provide an overview of the empirical data in Section 2.3. Next, we introduce our research methodology, DUPIMAGE, for using images in identifying duplicates and answering our **RQs** in Section 2.4 and present the results in Section 2.5. We then discuss the importance of publishing negative results in Section 2.6 and discuss the related works in Section 2.7. Lastly, we conclude by discussing the threats to validity and providing a conclusion (Section 2.8 & 2.9) and providing the details of our replication package in Section 2.10.

2.2 Baseline for Identifying Duplicate Questions

In 2015, Zhang et al. [208] introduced a model, DUPPREDICTOR, for identifying duplicate questions in StackOverflow. Later, in 2016, Ahsanuzzaman et al. [4] further extended this model into DUPE. For answering our research questions, we employed the DUPE model developed by Ahasanuzzaman et al. [4] as the extensively cited, frequently reused [191, 91], and replicated approach [161]. This choice served as the basis

for our evaluation, where we investigated the potential enhancement of its performance through image processing in identifying duplicate questions on Stack Overflow.

The approaches presented by DUPPREDICTOR [208] and DUPE [4] were evaluated on a dataset containing Stack Overflow questions up to September 2014 [204]. This dataset included 130,888 non-master duplicates and 90,245 master questions.

2.2.1 DUPPREDICTOR Methodology

Zhang et al. [208] proposed an approach to solving duplicate question detection on Stack Overflow. This approach takes a new question as input and returns a ranked set of questions potentially similarly to the input. The DUPPREDICTOR framework comprised two primary phases: (i) model-building and (ii) predicting duplicates.

In Phase (i), historical questions were gathered from Stack Overflow to serve as training data. Step 2 focused on preprocessing, involving extracting title, description, and tags from each question, followed by text tokenization, removal of common English stop words, and stemming using the Porter stemming algorithm [197]. After looking at different parts of questions (title, description, topic, and tags) explained below, the system calculated a score for how similar each question was to the original question. The training learns the best way to combine these four similarity scores. It uses a set of known duplicate questions to figure out how important each type of similarity is. For example, it might learn that title similarity is more important than tag similarity for finding duplicates. This training process automatically finds the right balance between the four scores to get the best results.. This final score was used to rank other questions based on how likely they were duplicates of the original question.

In their approach, Zhang et al. [208] measured similarity across several aspects of questions to identify duplicates. Title similarity considered how many common words question titles shared, leveraging a topic model [71]. Description similarity focused on the overlap of words within the question descriptions after converting them to a simplified format for comparison. Finally, tag similarity assessed how closely related the

tags assigned to each question were, indicating how well the questions might address similar topics [208].

Once the model is trained in Phase (i), we transition to Phase (ii), where the trained model is applied to detect new duplicate questions by comparing each new question with all past questions on Stack Overflow and computing similarity scores. For measuring similarity, similar to Phase (i), new Similarity scores are calculated between the new question and each historical question using the same four factors (title, description, topics, and tags). After calculating each similarity, a final consolidated similarity score is calculated to rank the historical questions. Questions with higher scores are placed higher in the ranked list, indicating a greater likelihood of being duplicates of the new question. As a result, the model outputs the top-K most similar questions, providing a curated set of historical questions that potentially mirror the new question.

2.2.2 DUPE Methodology

Expanding upon DUPPREDICTOR, Ahasanuzzaman et al. [4] introduced DUPE to address duplicate questions on Stack Overflow. Both methods aim to identify potential duplicates based on title, content, topic, and tag similarity but differ in several aspects. DUPE uses a discriminative classification model [147] for duplicate detection, while DUPPREDICTOR relies on a combined similarity score. DUPE also incorporates more features and investigates the reasons behind duplicate occurrences, aspects not covered by DUPPREDICTOR. DUPE examines question characteristics, including wording and phrasing, to understand why duplicates happen, potentially revealing issues like poor searching or confusing question formulation [4]. By understanding these reasons, DUPE aims to improve the Stack Overflow experience, potentially reducing duplicate questions and enhancing answer searchability.

DUPE operates through three main phases. In the initial phase, the text of the questions undergoes preprocessing. The second phase involves collecting various features, as discussed below, for each question pair (where a pair consists of a question and its

duplicate) to train a binary classification model. This model plays a crucial role in the third phase, where duplicate question detection occurs.

In the initial phase of DUPE, the text undergoes preprocessing, which involves removing stop words, performing stemming, and converting tags into a unified form. On Stack Overflow, tags follow a master-synonym structure. A tag synonym is a tag that shares an identical meaning with another tag or, in some cases, may be a subset of another tag. To enhance the duplicate detection process, DUPE substitutes each instance of a synonym tag with its corresponding master tag. The second phase of DUPE employs a binary classifier to pinpoint duplicate questions. This classifier relies on five key features. Cosine similarity, based on the traditional vector space model (using TF-IDF representations rather than word embeddings), evaluates the similarity between questions by considering the angles between their vector representations. Term overlap directly measures how many identical words the questions share. Entity overlap, measured using the Jaccard coefficient, captures the proportion of shared entities between the questions. An entity, in this context, refers to a word or phrase representing a real-world object or concept (e.g., 'Python', 'Android Studio', '2015'), extracted using Named Entity Recognition. These entities are used to assess semantic similarity between question pairs. Entity type overlap goes beyond just the entities themselves, considering if the types of entities also overlap. Finally, WordNet similarity, leveraging a Java library (WS4J), calculates semantic relatedness between questions using the WordNet knowledge base, identifying deeper conceptual similarities beyond just surface-level keywords.

Ahasanuzzaman et al. [4] analysis in DUPE by drawing insights from their results showed that cosine similarity, which measures the similarity between two vectors by calculating the cosine of the angle between them, is the most valuable feature in this context, while other features provided little to no improvement in the model's performance. With these features, the classifier determines whether a given pair of questions and their corresponding feature values are duplicates and assigns a probabilistic value indicating the level of duplication. Given the potentially large number of question pairs

classified as duplicates, relying solely on a binary classifier is insufficient. Therefore, in the third phase, similar to DUPPREDICTOR, they output the top k probable duplicates by sorting the questions based on their duplication levels to generate a ranked list of results. This ranking is determined by the likelihood of the new question being a duplicate of another in the Stack Overflow repository.

2.2.3 Evaluation Metrics

The promising results of these models were further extended by multiple researchers in a series of models [161] [191] [91]. In all these studies, including DUPPREDICTOR and DUPE, to evaluate the performance of the proposed techniques, the authors used the recall rate. Here, recall rate is important because we do not want to miss any actual duplicates, which is the main goal of our study.

$$recall - rate_k = \frac{N_{dk}}{N_{all}}$$

Here, $recall - rate_k$ refers to the recall of a technique within top-k recommendations. The term N_{all} refers to the total number of duplicates in our test data, and the term N_{dk} refers to the total number of detected duplicates in the top-k recommendations. In the previous works [4] [161] [208], k is set to 5, 10, and 20, respectively, and we also evaluate our results based on these three metrics.

The best results achieved by both models are presented in Table 2.1. In the baseline studies, authors divided the dataset into different question groups such as Java, Python, Ruby, C++, HTML, and Objective-C. As the table clearly shows, DUPE outperformed DUPPREDICTOR in terms of performance.

2.3 Empirical Data

This section details the processes involved in constructing our experimental dataset. We focused on gathering duplicate and non-duplicate question-image pairs to investigate the role of visual information in duplicate question detection. In this section, we discuss the process we applied for data retrieval and creation.

TABLE 2.1: Baseline Model Results for DupPredictor and Dupe

Baseline Model	Question Group	Best Recall Rate	
DupPredictor [208]	Ruby	Top 5 (%)	33.00
		Top 10 (%)	37.00
		Top 20 (%)	39.00
Dupe [4]	Ruby	Top 5 (%)	51.16
		Top 10 (%)	59.56
		Top 20 (%)	66.11

We use this data to evaluate baselines explained in the previous section and perform benchmarks. We also use this data to evaluate whether the use of images would increase the chances of identifying duplicate questions on Stack Overflow and answering our RQs.

2.3.1 Data Gathering Process

For our study, we sourced data from Stack Overflow by querying the Stack Exchange Data dump [165]. The data spans several years since the inception of Stack Overflow in 2008 up to December 31st, 2022. We had two criteria for including images in our set:

Criteria 1- Questions marked as duplicate: Our focus was on identifying duplicate questions within the Stack Overflow platform. When a question is identified as a duplicate, it is closed and linked to a "master question" containing the solution. This redirection ensures users can efficiently access relevant information. To achieve this, we adopted the approach established by Ahasanuzzaman et al. [4]. Their work identified a specific marker used to flag duplicate questions. We leveraged the Stack Exchange Data Dump [165] to extract these questions using targeted queries. To clearly distinguish them within our dataset, we appended the string "[Duplicate]" to the titles of these questions.

Criteria 2- Questions with an image: Given our primary focus on the role of images in question duplication, we specifically tailored our query to filter for questions with an attached image in the body section of the posts. We can easily get only those posts

that contain these images by integrating a condition to look for `<img>` tag in the body section of each post when querying from the Stack Exchange Data Dump [165]. This targeted selection strategy helps us get only those posts with images attached.

By querying the *posts database* in the Stack Exchange Data Dump along with these two criteria, we could extract 42,462 non-master duplicate questions. To form a fair and balanced baseline for comparison, we randomly queried a total of 20,493 non-duplicate questions that were closed and marked as non-duplicates and were not marked as masters of any other question.

Non-master duplicate questions are those identified as duplicates of other questions and, as a result, were closed without extensive discussion. These questions are linked to a master question. Following this, we sought to identify the 'master' questions to which these duplicates were linked. Master questions serve as the original or primary questions that the duplicates reference. To collect these master questions, we queried the `postlinks` and `posts` databases in the Stack Exchange Data Dump, identifying a total of 6,265 master questions [4].

It's important to note that all the questions in our dataset, the duplicates, non-duplicates, and the master questions, have images attached. This unique characteristic of our dataset will allow us to explore the role of images in detecting duplicate questions with greater specificity and accuracy.

2.3.2 Data Set Creation

After extracting each question from the data gathered in Section 2.3.1, we applied a series of preprocessing steps following the process explained in the baseline model DUPE [4]. We collected each question's title, body, tags, and image URLs using the `BeautifulSoup` library. For each question, we removed stop words from the title and body and applied Porter's stemming algorithm to textual features to prevent unwarranted discrepancies [4].

Following DUPE's methodology [4], we substituted each instance of a synonym tag with its corresponding master tag. We then established connections between duplicate

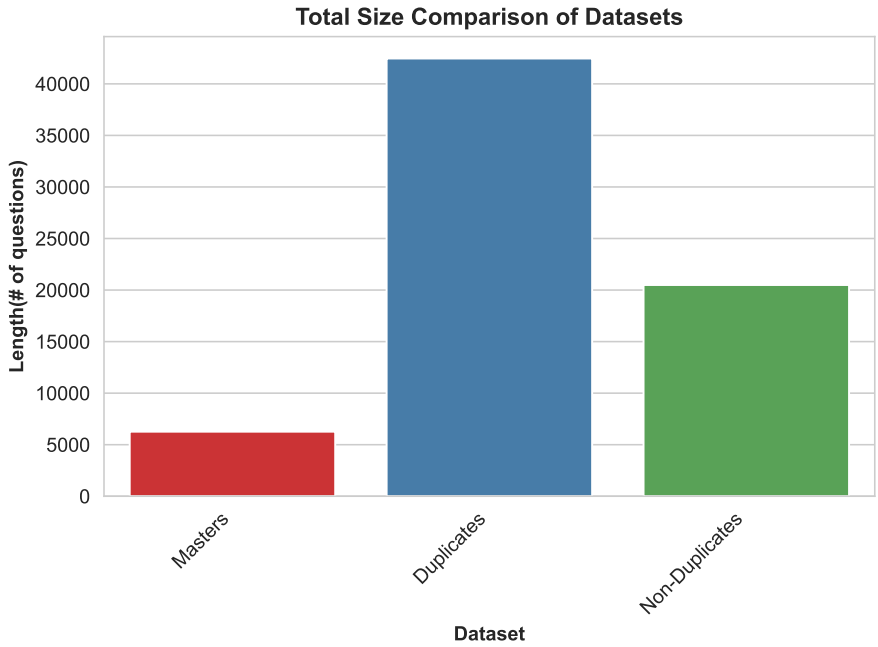


FIGURE 2.2: Overview of the size of the dataset

questions and their master questions, forming pairs. For every duplicate question, we examined the *RelatedPostId* to locate its master question, generating a pair if the master question was present in our data. This process resulted in 6,265 pairs of duplicate questions.

Following a similar process to the previous step, we generate non-duplicate question pairs. This is achieved by randomly associating master questions with 6,265 non-duplicate questions randomly selected from the 20,493 non-duplicate questions gathered in Section 2.3.1, thereby creating an equivalent number of non-duplicate question pairs. We can be sure that they are not duplicates because non-duplicate questions are closed and marked as not duplicates of any other question in stack overflow.

We then filter out those questions for which the image URL was inaccessible. This was necessary to ensure the integrity and usability of our data. Given the resource-intensive nature of image processing tasks, we decided to reduce the complexity of our problem. We achieved this by dividing our dataset in half, which resulted in 3,003 pairs of duplicate and non-duplicate questions, serving as our model's dataset. This selection strategy acknowledges resource constraints and the time required for image processing

while still providing a sufficient data volume for practical model training.

Note: *All the questions in the comprised dataset at the end of Section 2.3 have images attached to them.*

2.4 Proposed Methodology: Image-based Duplicate detection (DUPIMAGE)

We began by replicating DUPE to identify duplicate questions, adopting the methodology from Ahasanuzzaman et al.[4]. This involved applying logistic regression, a probabilistic classifier that generates predictive probabilities for each class. We focused on these probability values to identify the top-k questions and determine whether duplicates exist within this subset. While newer machine learning techniques, including deep learning, might yield better results, our primary objective was to investigate whether images provide additional value in duplicate question detection. Thus, we opted for the simpler logistic regression approach for two main reasons. Firstly, it allowed us to facilitate a direct comparison with the baseline established by Ahasanuzzaman et al. [4]. Secondly, the baseline [4] reported that logistic regression performed better in their approach than other traditional methods they explored. By adhering to this methodology, we could more accurately assess the impact of incorporating image data into the existing framework for duplicate question detection to properly evaluate the value that images add to duplicate question detection. To achieve this, we used the dataset created in Section 2.3.2.

To incorporate images into the detection process, we identified three methods for integrating data: using textual data extracted from the images (**RQ1**), using image captions (**RQ2**), and combining textual content from the pictures with captions generated from the question (**RQ3**). We input this data into our logistic regression model to identify duplicates based on textual content and image information. To extract relevant data from the images, we utilized OCR to extract text present in the images and employed the `Gemini-1.0-pro-vision` model for generating captions. By integrating these

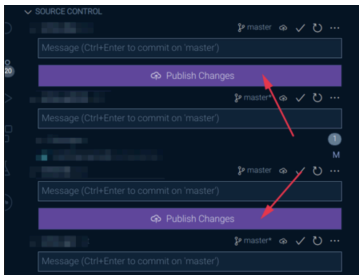


Image Caption:
Click the **Publish Changes** button to publish your changes.

OCR Text:
SOURCE CONTROL I f master Message Ctr+Enter to commit on master C Publish Changes Ps master Message Ctr+Enter to commit on master i M f master Message Ctrl+ Enter to commit on master O Publish Changes is master a Message Ctrl+Enter to commit on master I

(A) Image Captions and OCR for an image in Master Question

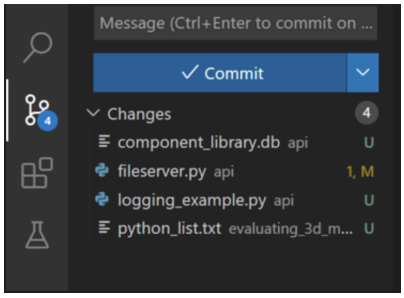


Image Caption:
The image shows a screenshot of a Visual Studio Code window. The window is split into two panes. The left pane shows the source control changes. The right pane shows the files that have been changed.

OCR Text:
(Message (Ctrl+ Enter to commit on . O VCommit 8% V Changes 4 a component library.db api U R fileserver.py api 1. logging_ example.py api U a a python list.txt evaluating 3d m .. U

(B) Image Captions and OCR for an image in Duplicate Question

FIGURE 2.3: Both the images discuss committing changes, and the text and captions generated are very relevant. OCR Text indentation has been removed to make it look cleaner.

image-based features with the original DUPE approach, we aimed to enhance the accuracy and effectiveness of duplicate question detection in software engineering contexts. We avoided directly measuring image similarity because semantic relevance (e.g., error message content, context, task type) is more important than visual resemblance. Thus, we focused on extracting and comparing text (using OCR) and meaningful descriptions (using captions) rather than relying on visual similarity metrics, which risk high false positives in this domain.

2.4.1 RQ1: Mining text from image with Optical Character Recognition (OCR)

Since we are focusing on finding duplicate questions on Stack Overflow, the images we will look at are most likely screenshots, bits of code, diagrams, charts, or user interface mockups (UIs). Because questions with similar problems are likely to have similar text in their images, it is crucial to be able to read the text within these images. This is where OCR proves valuable and helps us identify the text in the images.

OCR is a computer vision technology that primarily aims to convert non-editable textual content embedded in images, scanned documents, or other visual formats into machine-readable text [51, 105, 102]. The process involves these main steps: First, the

image might be converted to grayscale and cleaned up to make the text clearer for the computer to read. Then, the image is divided into smaller chunks, separating individual letters from the background. From each chunk, features like shape or edges are extracted. Finally, these features are compared to a database of known characters (like letters and numbers) using clever algorithms to determine the most likely character in each chunk. By combining these steps, OCR systems can create a function that takes an image region and outputs the most likely recognized character based on what it "sees" in the image. [37]

For reference, look at an image attached to a Stack Overflow question where a user encounters issues with version control in Visual Studio Code (VSCode) in Figure 2.3. The screenshot might depict the changes staged for commit within the VSCode interface. OCR can take screenshots like this and convert the text within the image, such as file names or error messages, into a format the computer can understand. By integrating the extracted text from the image and the written text of the question itself, we can improve our chances of finding duplicates. Textual similarity of images becomes another data point for identifying questions that might address the same issue, even if the phrasing in the written text differs slightly.

To compute textual similarity between images, we follow the same approach as the baseline DUPE model, using TF-IDF vectorization followed by cosine similarity calculation. The OCR-extracted text from each image is first preprocessed (stop word removal and stemming), then converted to TF-IDF vectors, and finally compared using cosine similarity. This approach ensures consistency with the baseline methodology while enabling us to incorporate visual text information into the duplicate detection process. Textual similarity of images becomes another data point for identifying questions that might address the same issue, even if the phrasing in the written text differs slightly.

For our study, we used `Nanonets` [106] to implement OCR in our DUPIMAGE. `Nanonets` is a deep learning-based library with a high-accuracy implementation for the OCR task. The model is based on advanced convolutional and recurrent neural networks. OCR part of our methodology is shown in the process diagram in Figure 2.4 in `DupImage`

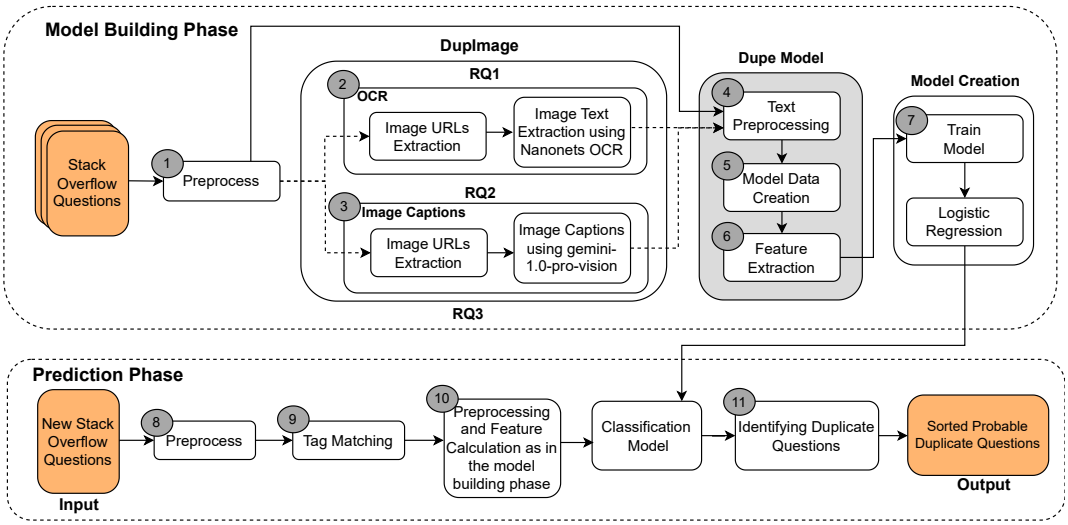


FIGURE 2.4: Overview of the processes used for detecting duplicate questions on Stack Overflow: Text, OCR, and Captions

section. We first extract the Image URL and then use `Nanonets` to extract text from the image.

The computed OCR-based cosine similarity score is used standalone and with the textual features used in DUPE methodology. When used with DUPE, this score is treated as an additional feature and combined with other similarity scores (e.g., title, body, tags, code) from DUPE model. This feature is then included in the input vector for the logistic regression classifier, allowing the model to learn how much weight to assign to the image-based textual content during training.

2.4.2 RQ2: Image Captioning with Gemini-1.0-Pro-Vision Model

In the previous section (Section 2.4.1), we discussed using OCR to read text from images. As evident, it just focuses on extracting the text from the images, and no semantic understanding of the images is captured. To address this issue, we incorporated image captioning as it helps capture the semantic meaning of the images. We utilized a powerful Large Language Model (LLM) from Google called `Gemini-1.0-Pro-Vision` to automatically describe the visual content within images attached to Stack Overflow questions. These descriptions can be very helpful for finding duplicate questions, especially when the image itself might not be very informative or does not contain text for

OCR. To make it more robust, we performed image captioning on each question pair three times and chose the captions that gave us the best similarity scores. This was done to avoid hallucinations by the LLM model, as it can generate different captions for the same image.

To better understand image captioning, consider the images attached to questions about version control in Visual Studio Code (VSCode), as shown in Figure 2.3. Even when screenshots lack explicit branding elements like VSCode Icon, image captioning can effectively identify these IDEs, aiding in image comparison. This technique analyzes the overall layout, presence of buttons or menus, and potential text snippets. By generating descriptions that mention meaningful elements like "source control changes" or "code editors", image captioning provides valuable insights beyond OCR-extracted text. This additional information significantly enhances our ability to identify duplicate questions addressing similar underlying issues.

Compared to Optical Character Recognition (OCR), which extracts text embedded within Stack Overflow question images (code snippets, error messages, etc.), image captioning offers a broader perspective on the visual content. While OCR focuses on the "what" (written text), image captioning can draw insights from the "why" and "how" aspects of the image. It can capture objects, actions, and relationships depicted in the image that might not be explicitly present as text.

Google's `gemini-1.0-pro-vision` model leverages the power of deep learning, specifically convolutional neural networks (CNNs), to analyze images and generate captions that capture their essence [181, 60]. By going beyond basic description, these captions provide a richer understanding of the image's content. Similar to OCR, we first extract the image URL and then pass the URL to `gemini-1.0-pro-vision` model to generate captions as shown in `DupImage` part in the process diagram of Figure 2.4.

Like the OCR-based similarity, the image caption similarity score is used as an individual feature in the classifier. It is combined with the DUPE model's textual similarity scores to form a complete feature vector for each question pair. This approach enables the model to learn whether and how image semantics contribute to duplicate detection

TABLE 2.2: Evaluation of “recall rates" to measure the impact of different Image Comparison Techniques

No.	Configuration	Top 5 (%)	Top 10 (%)	Top 20 (%)
1	Replication Results on Image Dataset (Dupe)	43.43	54.24	61.06
2	Only OCR	21.63	25.12	30.28
3	Only Image Captions	13.98	16.98	24.28
4	OCR with Dupe Text Model	*45.42**	55.57	62.23
5	Image Captions with Dupe Text Model	43.93	54.25	61.23
6	OCR + Image Captions + Dupe Text Model	45.26	*55.64**	*62.37**

beyond traditional textual signals.

2.4.3 RQ3: Combination of Optical Character Recognition and Image Captioning

Building upon the individual strengths of OCR (Section 2.4.1) and image captioning (Section 2.4.2), RQ3 explores how their combined power can enhance duplicate question detection on Stack Overflow. OCR excels at extracting text from images, providing valuable keywords and phrases that can directly relate to the problem at hand (Section 2.4.1). Image captioning, on the other hand, goes beyond just text by analyzing the visual content and generating descriptions that capture the overall context of the image (Section 2.4.2).

In our work, we developed a robust similarity measure for duplicate question detection on Stack Overflow, leveraging both textual and visual information. We combined similarity scores from two sources: 1) Optical Character Recognition (OCR) to extract text from question images (**RQ1**), and 2) image captioning using `gemini-1.0-pro-vision` to generate captions describing the visual content (**RQ2**). We employed a max function to achieve a comprehensive similarity score, selecting the higher score between the image text similarity and the image caption similarity for each question pair. This approach addressed scenarios where images might have imbalanced information distribution, such as graphs with minimal text but high semantic meaning.

We selected the maximum similarity score approach instead of averaging or weighted averaging methods because our similarity measures exhibit asymmetric error patterns.

Through manual examination of question pairs, we observed that both OCR-based text similarity and image caption similarity tend to miss different aspects of image relevance. However, when even one of the two scores was high, the corresponding duplicate question was frequently ranked among the top probable duplicates. This motivated our use of the maximum function, as it increases the likelihood of capturing true duplicates without being penalized by low values from the less informative modality. OCR may miss semantic context while precisely capturing textual elements, whereas image captioning may miss specific technical details while capturing overall visual semantics.

$$S_{\text{combined}}(q_i, q_j) = \max(IT_{\text{sim}}(q_i, q_j), IC_{\text{sim}}(q_i, q_j)) \tag{2.1}$$

where

- $S_{\text{combined}}(q_i, q_j)$ is the combined similarity score between question i and question j.
- $IT_{\text{sim}}(q_i, q_j)$ is the image text similarity score between question i and question j, obtained from OCR in Section 2.4.1.
- $IC_{\text{sim}}(q_i, q_j)$ is the image caption similarity score between question i and question j, obtained from image captioning in Section 2.4.2.

To confirm the effectiveness of our approach (using the max/higher similarity score), we manually analyzed some question pairs. We calculated a value called "delta" for each pair. This delta simply represents the absolute difference between the two similarity scores (image text and image captions). Question pairs with a significant delta indicated a big difference in scores. For manual analysis, we considered a question pair to have a significant difference if the delta was higher than 0.5. This threshold gave us 485 pairs with a 'delta' value greater than 0.5. We then closely examined these question pairs with a large difference in scores. Even though the image text and image caption similarity scores might not have been very close, we found that the questions were duplicates in most cases. This reinforces the importance of using information from both the image text and the visual content of the images. Some images might have very little written

text, but the image might clearly show what's happening. By considering both text and semantics, we can improve our ability to identify duplicates, even when the information is presented differently.

$$\delta(q_i, q_j) = |IT_{\text{sim}}(q_i, q_j) - IC_{\text{sim}}(q_i, q_j)| \tag{2.2}$$

where

- $\delta(q_i, q_j)$ is the delta value representing the absolute difference between image text and image caption similarity scores for question i and question j.

2.5 Evaluation Results

In this section, we present the results of our benchmarking to answer the **Research Questions** in the designed case study and illustrate the experimental results of our approaches compared to the baseline approach of DUPE. We run all our experiments on the *image dataset*, which comprises 3,003 non-master duplicate questions, 3,003 non-duplicate questions, and 3,003 master questions as discussed in Section 2.3. We use this dataset with images in the questions as the chapter's main focus is to identify how important images can prove to be in detecting duplicate questions on Stack Overflow. Moreover, we have not divided our dataset based on the question groups as in the previous approaches [4] [208] [161], as the size of our image dataset is reasonable enough.

We allocate 80% of both the duplicate pairs and non-duplicate pairs datasets, discussed in Section 2.3.2, for training our model, ensuring a balanced representation to prevent any training bias. The remaining 20% of the duplicate-pairs dataset is used for model evaluation. Based on previous methodologies [4, 161], where they used cosine similarity value as a significant feature for detecting duplicate questions, we also incorporate cosine similarity as a feature in our baseline approach, applying it to title-title, title-body, body-body, and code-code comparisons.

Firstly, we integrated the image text obtained through the OCR technique, as outlined in Section 2.4.1, and computed the cosine similarity between vectors of image text pairs as part of our model evaluation. Next, we employed the image captioning method

to determine the similarity score between images based on key visual descriptions, as detailed in Section 2.4.2. Lastly, we looked at the potential of combining these two techniques and assessed their collective impact on detecting duplicate questions on Stack Overflow as discussed in Section 2.4.3.

2.5.1 RQ1: Results of integrating OCR

In our research, to answer **RQ1**, we wanted to see if using images could help find duplicate questions on websites like Stack Overflow. We realized that images in these questions often contain important text that explains the problem. So, we developed a method (explained in Section 2.4.1) to find and analyze this text from the images. Our goal was to find similarities between the text in the images and use that to identify duplicate questions. By treating the text in images as valuable information, we aimed to connect the traditional text-based analysis with the image-heavy world of online Q&A platforms.

Unfortunately, our findings as depicted in Table 2.2 show that extracting text from images for comparison didn't significantly improve our ability to find duplicate questions which is measured by $recall - rate_k$. While using OCR alone performed poorly, even combining it with the existing method of DUPE only led to a small improvement (around 2% on average). These results suggest that focusing on text analysis within images wasn't as effective as we hoped for duplicate question detection.

One potential limitation of using OCR for duplicate question detection is the inherent similarity between images in related questions. Even when questions share similar meaning or intent, the actual text displayed within the images might differ. For instance, questions about troubleshooting a specific software issue could both depict screenshots of the program's interface, yet the specific error messages or menu options displayed might vary. This text discrepancy can affect the effectiveness of OCR in accurately identifying duplicate questions. Additionally, OCR might struggle with poorly lit or blurry images, further reducing its accuracy in capturing relevant textual information. However, Nanonets' OCR is pretty robust in detecting text on dark backgrounds.

2.5.2 RQ2: Results of integrating Image Captions

In our study, for RQ2, we looked at using image captions to understand the semantic meaning of images attached to questions on Stack Overflow. These captions can be very helpful, especially when questions include similar pictures with minimal text in them. We developed a method (explained in Section 2.4.2) to automatically generate captions describing what the images show. Our goal was to find duplicate questions by comparing these captions.

However, the results we obtained, as illustrated in Table 2.2, were not up to the mark. While we anticipated that extracting conceptual information from images through captions would improve duplicate question detection, the results did not demonstrate a significant increase in the $recall - rate_k$. Image captioning on its own, as our results in Table **??** showed, did not perform as well as we had hoped. Even when we combined this approach with the existing method of DUPE, the improvement in recall rate was still relatively small, typically around 1% on average. These findings suggest that focusing on understanding the content described in image captions wasn't as successful as we had hoped for identifying duplicate questions.

While image captioning holds promise for identifying duplicate questions, its effectiveness can be limited by the capabilities of Large Language Models (LLMs) used to generate captions (`gemini-1.0-pro-vision` in our case). LLMs can sometimes create inaccurate captions ("hallucinate") or focus on irrelevant details in images. For example, similar error messages on different operating systems like Linux and Windows might be interpreted differently by the LLM, leading to dissimilar captions and predicting duplicate detection wrong. Recognizing these limitations is crucial for refining image captioning techniques for this specific task.

2.5.3 RQ3: Results of the combination of OCR and Image Captions

In our study, to answer **RQ3**, we also investigated the potential of combining OCR for text-based analysis and Image Captions for image content analysis to enhance duplicate question detection in the Q&A platforms. While images play a significant role in Q&A

platforms, the integration of these approaches, even when combined, did not result in a substantial improvement in the $recall - rate_k$, as illustrated in Table 2.2. While our investigation yielded some improvement in duplicate question detection, the overall improvement was not significant enough to consider this a successful integration.

Our research aimed to enhance the performance of existing state-of-the-art models, such as DUPE [4], for duplicate question detection on Stack Overflow by incorporating information from question images. While we successfully integrated image data, a substantial improvement in overall performance was not achieved. This finding suggests that incorporating visual information, while a promising avenue for future research, may require more sophisticated techniques to unlock its full potential for duplicate question detection.

Although the improvement achieved through the combination of OCR and Image Captioning is marginal, on average, around 2%. The findings suggest that images do provide supplemental information to the text within the question (title, body, and code). However, we categorize this result as negative due to the increased complexity of integrating these image-processing techniques. Generating OCR text and image captions for the entire duplicate questions with images dataset (DUPIMAGE dataset) required approximately 12 hours each on Apple M2 Pro with ten cores. This significant processing time, when compared to the textual based process used in DUPE which runs in minutes, highlights the computational burden associated with these methods.

2.6 Importance of negative results

The call for ESEM 2024 encouraged authors to publish negative results. While we slightly could demonstrate an improvement of 2% when using images compared to the pure analysis of text with methods such as DUPE, we consider this improvement trivial, which does not justify the allocated resources and the complexity involved in image processing.

Discussing negative results in empirical software engineering is crucial for several

reasons [79, 104]. First and foremost, sharing negative results helps to avoid duplication of effort within the community. By clearly demonstrating that integrating image processing for our specific application resulted in only a marginal improvement, other researchers can make more informed decisions about resource allocation. This further is complemented by the comprehensive replication package of not only our DUPIMAGE method and data set but also the replication package we created for DUPE. Furthermore, publishing negative results contributes to a more comprehensive understanding of the problem space. It helps to map out the boundaries of effective techniques and can guide future research toward more promising areas or novel approaches [46, 103]. Additionally, acknowledging and discussing negative results enhances the credibility and authenticity of the research process. It reflects a commitment to transparency and rigor rather than merely pursuing and highlighting positive outcomes.

2.7 Related work

We discuss the problem of identifying duplications in software repositories and the use of image processing in software engineering in this section.

2.7.1 Identifying duplicate entries in Software Engineering

Identifying duplicate bug reports in issue-tracking systems is crucial for software engineers to enhance software teams' efficiency and software developers' productivity. Sun et al. [171] used discriminative models for information retrieval and achieved substantial 17–43% relative improvements in OpenOffice, Firefox, and Eclipse datasets, relying solely on common natural language information. Sometimes, these duplicate bug issues can prove to be useful. Bettenburg et al. [26] conducted a survey and found that while most developers have experienced duplicated bug reports, only a few considered them a serious problem. Meanwhile, a number of studies have developed methods to identify duplicate bug reports [178, 207, 193, 11, 172].

Similarly, with the emergence of crowdsourcing Q&A platforms like Stack Overflow, identifying duplicate questions is a research focus. Studies showed that the majority of duplicated questions are re-asked only once, approximately 15% are posed more than twice, with some reaching up to 558 times, resembling frequently asked questions rather than typical duplicates [45]. However, some argued the positive impact of duplicate questions on developers' understandings [1]. Apart from DUPPREDICTOR [208] and its extended method DUPE [4], as well as the replication of that by Silva et al. [161], which we employed as the baseline in this study, several other studies have explored the application of deep learning for enhanced text analysis to identify duplicate questions. Wang et al. [191] utilized deep learning techniques, such as Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), and Long Short-Term Memory (LSTM), to detect identical questions on Stack Overflow. Additionally, Liao et al. [91] employed the Siamese BiLSTM method to capture semantic and contextual relevance between questions, addressing the duplicate issue. They reported a moderate improvement compared to the previous deep learning model. These questions can also be used for mining requirements[145] and thus detecting duplicates is essential.

2.7.2 Use of Images for Software Engineering tasks

In recent years, the importance of visual content in software engineering has been increasingly recognized. The rise of social coding environments and social networks has led to a surge in the sharing of images among developers [111]. Recent studies on Bugzilla and Stack Overflow over nine years reveal a doubling in image-inclusive posts between 2013 and 2022. These images, often essential for understanding the content, increased engagement from other developers. In 86.9% of cases, comprehension without the image was unlikely, emphasizing the need for visual content consideration in software engineering tool design and developer behavior analysis. [112]. Automated testing for cross-browser compatibility often struggles with false positives. Browser Bite, a tool leveraging image segmentation, differencing, and machine learning, demonstrates significant improvement, achieving an F-score exceeding 90%. This highlights

the potential of image-based machine learning for more accurate software testing [160]. TANGO, a technique for identifying duplicate bug reports from videos using computer vision and text analysis, shows that TANGO can effectively rank duplicates and significantly reduce developer effort compared to manual analysis [41].

A recent study [61] proposed 23 image features for classifying UML class diagrams (UML CDs). It found 19 of these features to be influential predictors, with six classification algorithms achieving a prediction rate of nearly 96% for UML CDs and 91% for non-UML CDs. Moreover, another study [187] reveals that questions with images are more likely to get accepted answers than those without images on platforms like Stack Overflow, demonstrating the importance of images. For instance, Calefato et al. [34] discovered a positive correlation between including code snippets and the success of questions. Jason et. al. introduced WebUI, a large dataset of automatically collected web pages with extracted visual information. The authors explore using web semantics to improve visual UI understanding models, especially for mobile applications with limited labeled data [199].

`GIFdroid`, a lightweight approach using image processing to automatically replay bug reports from screen recordings by generating the execution trace [47]. Existing GUI grouping methods rely on supervised learning or heuristics, while this work proposes a novel unsupervised image-based approach inspired by perceptual grouping principles [200]. However, the influence of images, a commonly used non-textual element in today's society, is still largely unexplored. Therefore, we aim to investigate the potential of using images accompanying questions to detect duplicate queries, a task of significant importance.

2.8 Threats to Validity

Just like any empirical study, ours has some limitations that we need to consider. These limitations can affect how well our findings apply to other situations.

External Validity: The generalizability of our findings depends on the representativeness of our chosen sample. The types of questions we looked at and the time period

we covered could influence how well our conclusions can be applied to other situations. Additionally, online communities are constantly changing. How people use the platform, the types of questions they ask, and even the platform itself can evolve over time, potentially impacting the relevance of our results in the future.

Internal Validity: Threats to internal validity can influence the cause-and-effect relationships observed in our study. Measurement bias can arise from the performance of OCR and image-captioning technologies. The quality and variability of the images within the dataset can affect these technologies' accuracy. The specific implementations we used for OCR and Image Captioning could also introduce bias in our measurements. Another potential source of internal bias is annotation bias. Our study relies on the assumption that identified duplicate questions are accurately labeled. However, the subjective nature of duplicate question identification can introduce bias during the annotation process.

Construct Validity: Construct validity refers to the degree to which our study truly captures the concepts we intended to measure. In this case, a potential threat to construct validity is the difficulty of replicating our results. Variations in hardware configurations and computational resources can hinder the reproducibility of our experiments. For this study, we used a specific hardware setup (Apple M2 Pro with ten cores) and software environment (Python notebooks) to conduct our experiments. If researchers attempt to replicate our findings using different hardware or software, they may encounter discrepancies.

Conclusion Validity: We looked at how well using image data (extracted through OCR and image captioning) helped identify duplicate questions. While there was a slight improvement in accuracy, suggesting that images can provide additional information to the text in question on Q&A platforms, the time and computer power needed for these techniques were significant. Therefore, our conclusion that image features only offer a small benefit while requiring a lot of resources is well-supported by the data we collected and analyzed.

2.9 Conclusion

In the dynamic landscape of developer communities, where Q/A platforms are integral to knowledge sharing, our study responds to the evolving communication dynamics by exploring image-based techniques for duplicate question identification on platforms such as Stack Overflow. Despite the prevalent use of images, contemporary duplicate question detection methodologies predominantly rely on text-based analysis.

We investigated techniques to understand text in images (using OCR) and describe image content (using captions) for duplicate question detection. We also looked into the possible combination of OCR and image captions to make the model more robust and capture more details from the images. These methods yielded a modest improvement (around 1-2%), suggesting images complement the textual content of questions. However, they increased system complexity and processing power demands. Given the minimal improvement versus the significant drawbacks, image features seem to offer limited practical value for duplicate detection tasks, and hence, we present them as negative results. Our approach is easy to use and test by others, which paves the way for future research on better ways to find duplicate questions that consider both text and images on platforms like Stack Overflow.

2.10 Replication Package

For the reproducibility of this study and open science, we provide the community with all the artifacts used in our study. In particular, we provided the source code of our model with documentation and our labeled and unlabelled datasets. The project including all artifacts is available at this google drive link[⋆1]

[1] ⋆ All artifacts related to this study are available at https://drive.google.com/drive/folders/1ocDV2jFUz-mFDmWJebwlNd7saDe9lZVf?usp=sharing

Chapter 3

Inferring and Extracting Relevant Content from Programming Screenshots

Abstract

As developers increasingly rely on screenshots to communicate programming issues on platforms like Stack Overflow, there is a growing need to understand how generative AI models can support this shift from text-based to visual problem descriptions. This study investigates two key capabilities of large language models (LLMs) in this context. First, we assess whether LLMs can infer the intent and content of a programming query directly from a screenshot, without relying on textual input. Second, we evaluate their ability to detect the presence of relevant text within these images and to extract that content accurately. We evaluated three state-of-the-art LLMs—`LLAMA`, `GEMINI`, and `GPT-4o`—using prompt-based techniques such as in-context and few-shot learning, applied to a manually curated dataset of 223 Stack Overflow posts with screenshots. For question inference, `GPT-4o` achieved the strongest results, with over 60% similarity to the original question for 51.75% of images. Developer-rated relevance reached 0.69, and

embedding-based similarity peaked at 0.59, particularly for screenshots with focused, unambiguous content like code or error messages.

For relevance detection and text extraction, we evaluated model performance with and without OCR assistance. `GEMINI` achieved the highest F1 score (0.91) and recall (0.98), performing consistently across both conditions, indicating robust visual understanding. `GPT-4o` benefited significantly from OCR input, reaching BLEU 0.46 and ROUGE-1 0.68, while `LLAMA` lagged behind due to formatting inconsistencies and lower extraction quality.

Our findings highlight both the potential and current limitations of LLMs in supporting image-driven software support. They also point to concrete next steps, including targeted fine-tuning for visual programming tasks and improved prompt strategies for mixed-content inputs.

3.1 Introduction

Recent advancements in foundation models capable of interpreting multimedia content have sparked a compelling question: Are we approaching a future in which developers can upload screenshots of their IDEs—replacing elaborate text-based discussions on Q&A platforms—to seek solutions to development problems? As communication norms across social and technical platforms increasingly embrace rich media, it becomes pertinent to ask whether developers on sites like Stack Overflow might soon rely primarily on screenshots, such as code snippets or full IDE screenshots, with minimal or no accompanying text, to articulate their technical challenges. Motivated by the growing capabilities of foundation models to analyze and interpret visual content, this chapter explores how these developments could transform the nature of interaction on platforms historically grounded in text, enabling richer and potentially more intuitive exchanges in software development communities.

The growing popularity of rich media is reshaping communication norms, including in software development. Much like trends seen on social media, developers increasingly share images—such as screenshots—when fixing bugs, reporting issues, or

seeking help on Q&A sites [113, 186]. This marks a shift from traditional text-based interactions toward more visual, context-rich collaboration. As of April 2024, platforms like YouTube and Instagram are widely used by American adults (83% and 47%, respectively) [143], and global Android users spend over 31 hours per month on TikTok [62], underscoring the dominance of visual media. This shift is also evident in technical spaces—image sharing on Stack Overflow and Bugzilla has doubled over the past decade [111].

Stack Overflow discourages posting code as images, emphasizing text-based code for its searchability, accessibility, and long-term usefulness [167]. However, despite these guidelines, image-based posts—particularly screenshots of code and development environments—are on the rise [111, 69]. Nayebi and Adams [113] found that 69.4% of images on Stack Overflow depict code or terminal outputs, while Wang et al. [186] showed that 26% of images contain valuable technical information that remains unsearchable and underutilized. Although policies favor text for clarity [21, 54, 13], the increasing capability of foundation models to interpret visual content [151, 148, 149, 206] suggests a potential shift in how developers share and retrieve knowledge on technical platforms [189].

Our research is motivated by developers' challenges in clearly articulating technical questions, a process that often involves iterative refinement, adherence to strict platform norms, and concern over community judgment [141]. To reduce this burden, many users complement their posts with screenshots. We envision a solution where foundation models automatically analyze and annotate such screenshots, combining the ease of visual sharing with the benefits of searchable, structured content. This approach could lower the entry barrier for less experienced users while aligning with the goals of Q&A platforms to maintain accessible and high-quality knowledge.

This chapter investigates the potential of foundation models to understand programming-related screenshots, identify faulty areas, and interpret accompanying questions. While these models—and Large Language Models (LLMs)—have shown strong capabilities in

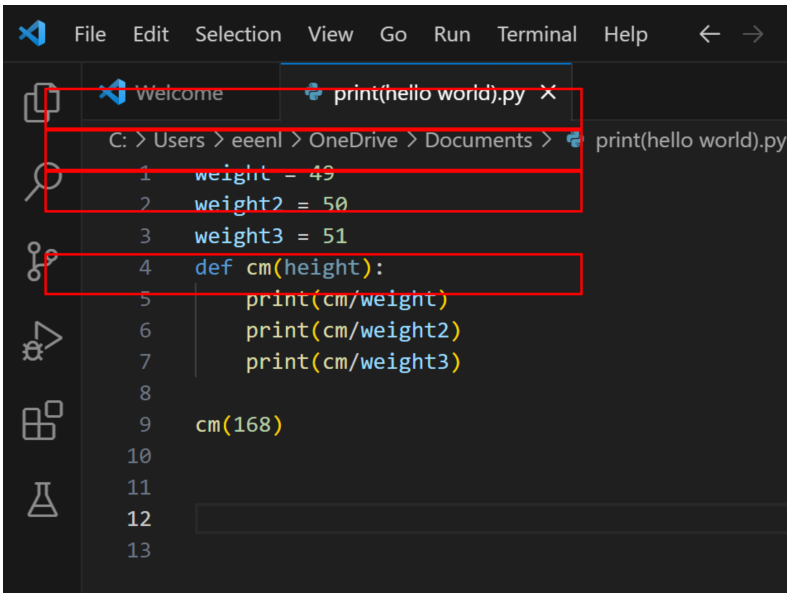


FIGURE 3.1: GPT4o's direct attempt at extracting relevant text or code, showing imprecise annotations

code understanding and generation [198], they still struggle with producing accurate visual annotations. In particular, current models fall short when tasked with highlighting specific issues within code screenshots. For example, as illustrated in Figure 3.1, models like `GPT4o` can provide general visual feedback but often fail to annotate or localize problems within the screenshot precisely.

In particular, we answer the following research questions:

RQ1: How do various LLMs and different prompt engineering techniques perform in inferring questions given code and IDE screenshots?

What and How: We evaluate the performance of three state-of-the-art LLMs (`GPT-4o`, `Gemini-1. 5-pro`, and `Llama-3.2`) using various prompt engineering techniques (in-context learning, chain-of-thought prompting, and few-shot learning) to determine their effectiveness in interpreting screenshots of code and IDEs. We assess each model's ability to generate relevant questions based solely on visual inputs and measure their similarity to original Stack Overflow questions using both embedding-based similarity metrics and expert developer evaluations.

RQ2: To what extent can multimodal LLMs accurately identify whether a screenshot

contains text relevant to a given programming question?

What and How: We examine the ability of multimodal LLMs to determine whether an image includes content pertinent to a related Stack Overflow question. We frame this as a binary classification task using a manually labeled dataset created by experienced programmers. Model performance is evaluated using accuracy, precision, recall, and F1 score—with and without Optical character recognition (OCR) to assess whether extracted text enhances the model's ability to identify relevant visual information.

RQ3: For screenshots representing a programming issue, how effectively can multimodal LLMs retrieve the text relevant to the associated question?

What and How: Using a programmer-annotated dataset as ground truth, we evaluate each model's ability to accurately extract only those text segments from screenshots directly relevant to the corresponding Stack Overflow question. We assess extraction quality using a range of text similarity metrics, including BLEU and ROUGE. To examine the role of preprocessing, we compare model performance when provided with OCR-generated text versus relying solely on their vision capabilities. This extracted text can then easily by highlighted on the screenshot using OCR to generate annotations.

This chapter is an extension of the published work to the Mining Software Repositories (MSR) 2025 conference [6]. Compared to that version, we made significant changes in this chapter:

First, **RQ2** and **RQ3**, along with their associated methodology and results, are novel and presented for the first time.

Second, For **RQ1**, we expanded our dataset by 55.9% by including questions posted between November 2024 and January 2025.

Third, We updated the results of **RQ1** to reflect this extended dataset and conducted a more comprehensive analysis.

Fourth, We release a fully annotated dataset of programming-related screenshots from Stack Overflow, including human-labeled faulty regions and corresponding questions.

Fifth, extended discussion of results and implications, threats to validity and future work based on the new findings.

In this chapter, we begin by introducing the research problem and motivation in Section 3.1. We then provide a comprehensive review of related work in Section 3.2, covering image processing in software teams, image analysis using LLMs, text extraction from images, and relevance detection in technical screenshots. Next, we present our research methodology in Section 3.3, followed by details of our experimental setup in Section 3.4, including data collection, baseline establishment, and implementation specifics. We present our findings across three research questions in Section 3.5, analyzing question inference capabilities, relevance detection performance, and text extraction quality for different LLMs. We then discuss the limitations of our approach and their implications in Section 3.6 and 3.7. Finally, we conclude with a comprehensive discussion of how LLMs can bridge the gap between visual and textual content in developer forums in Section 3.8 and provide information about our replication package in Section 3.9.

3.2 Related Work

We discuss image processing in software teams and projects, as well as the application of LLMs for general image analysis and code extraction from visual content.

3.2.1 Image Analysis for Software Tasks

With advancements in image processing, automatic analysis of images has become a tool in software engineering, particularly for testing purposes. Techniques have been developed to leverage image recognition for detecting user interface inconsistencies and improving the efficiency of software testing processes [36, 196, 31].

In addition to testing, image processing has been explored to facilitate developer communication and increase productivity on social coding platforms. Nayebi [111] investigated the role of visual content on platforms like Stack Overflow and Bugzilla, showing how screenshots enhance communication and support collaborative work among developers. Later, Nayebi and Adams [113] reported the positive impact of visual elements on developer productivity in these settings. Other studies have also characterized visual content in developer environments, such as Jupyter Notebooks [3] and Stack Overflow [86].

In our MSR paper, we examined how LLMs can be used to infer questions from programming screenshots, demonstrating that models like GPT-4o can achieve reasonable similarity to actual questions posted by developers. That work established a foundation for understanding how well current AI models can interpret programming screenshots, but we did not address the extraction of specific relevant text from these screenshots.

Research has further focused on the frequent sharing of code screenshots on social coding platforms, with a particular emphasis on using machine learning models for accurate text retrieval from images [23, 19, 89]. Beltramelli [23] introduced pix2code, a model for generating code from graphical user interface screenshots, demonstrating early potential for translating visual programming elements into textual code. Building on this work, Bao et al. [19] developed PSC2Code, which specifically targets the extraction of code from programming screencasts, addressing challenges such as noise reduction and formatting preservation.

3.2.2 Image Analysis using LLMs

LLMs have shown significant potential in image analysis across diverse fields by combining language and vision capabilities to perform complex multimodal tasks. Recent work integrates LLMs with vision models for applications such as image captioning, object recognition, and detailed description generation, supporting advancements in areas like healthcare, autonomous driving, and digital media [144, 72, 88]. Notably,

models like CLIP [144] align images and text embeddings, enabling the categorization of images based on natural language descriptions. Similarly, multimodal models like BLIP [88] have expanded these capabilities by producing contextually relevant responses to image-based queries.

Prompt engineering techniques have been used to guide LLMs in analyzing images with various approaches to improve accuracy and reasoning [32, 195]. Carefully crafted prompts have shown promise for specialized tasks, such as identifying objects in medical imaging or analyzing visual sentiment [190, 206, 72]. However, the application of these techniques to software engineering tasks, particularly for code and error message extraction from screenshots, remains underexplored [190, 55].

3.2.3 Text Extraction from Images

Traditional approaches to text extraction from images have evolved significantly over the decades. Early morphological techniques focused on separating text regions from complex backgrounds through image segmentation and feature extraction [57, 182]. These methods laid the groundwork for more sophisticated optical character recognition (OCR) systems that combine image preprocessing, character segmentation, and recognition algorithms [**Chaudhuri2017Optical**, 70, 154]. Recent advancements have introduced neural network approaches, particularly convolutional neural networks (CNNs), which have substantially improved the accuracy of text extraction from various image types [20].

In the context of programming screenshots, specialized tools have been developed to address the unique challenges of code extraction. PSC2Code [19] pioneered techniques specifically designed for programming screencasts, focusing on maintaining code structure and syntax during extraction. CodeMotion [77] offered novel approaches for interacting with code in instructional videos but remained limited to text-only outputs. Further research has enhanced developer interactions with programming videos through

more accurate code extraction methods [18]. Recent work has explored combining image super-resolution techniques with large language models to optimize OCR performance specifically for programming videos and screenshots [10]. Despite these advances, the integration of extracted text with precise visual annotations remains a significant challenge for current systems.

3.2.4 Relevance Detection in Technical Images

Determining the relevance of content within technical images represents a significant challenge requiring advanced computational techniques. Recent developments in content-based image retrieval (CBIR) have incorporated deep learning approaches to improve relevance determination [90, 155]. Convolutional neural networks have proven particularly effective for technical domain applications, with specialized architectures like DFIR-Net demonstrating improved performance in identifying relevant image elements [183]. These advances have enabled semi-supervised anomaly detection systems capable of identifying relevant elements with minimal labeled training data [107].

The integration of vision and reasoning has further enhanced image understanding through structured approaches such as Scene Description Graphs [2] and transformer-based architectures that establish relationships between visual elements and their semantic relevance [163]. In medical imaging, relevance feedback techniques have been developed to iteratively improve retrieval performance [192], with multi-modal approaches combining textual, visual, and semantic feedback demonstrating significantly improved accuracy in relevance detection [100, 99]. These developments offer promising directions for application in software engineering contexts, particularly for identifying and extracting relevant code segments, error messages, or UI elements from screenshots.

3.3 Method

3.3.1 Different Prompt Engineering Techniques for RQ1

We applied various prompt engineering techniques to evaluate the performance of different Large Language Models (LLMs) in interpreting screenshot-based queries for developers' technical problem-solving. Specifically, we used in-context learning [44], chain-of-thought prompting [195], and few-shot learning [32] as our primary prompt strategies. In-context learning involves embedding relevant examples within the input prompt, allowing the model to draw from similar problem-solving patterns [44, 32]. Chain-of-thought prompting, which encourages the model to generate step-by-step reasoning, has proven effective for complex queries that require multi-step solutions [195]. Additionally, few-shot learning guides the models with minimal examples to balance adaptability and context comprehension.

We benchmarked three prominent LLMs in this evaluation: `LLaMA-3.2`[179], `Gemini-1.5-pro` [175], and `GPT-4o` [65]. We evaluated each model using various prompt engineering techniques to determine their responsiveness and accuracy in processing visual inputs from the programming environment. This comparative analysis identifies methods that improve interpretation and assesses the strengths and limitations of current generative AI technology in handling visual-based queries related to programming tasks. The findings provide insights into the feasibility of employing generative models as co-pilots [59, 135] for image-centric problem-solving on developer forums.

Based on our initial experiments with question inference (RQ1) (detailed in Section 3.5), we found that in-context learning consistently produced the highest similarity scores and developer-rated relevance across all evaluated LLMs. Given these findings, we opted to use in-context learning exclusively for our text relevance detection (RQ2) and text extraction tasks (RQ3). This decision allowed us to standardize our approach across all research questions while leveraging the most effective prompting technique identified in our preliminary studies.

3.3.2 Dual Extraction Methodology for RQ2 and RQ3

To comprehensively evaluate LLM capabilities in text extraction from screenshots, we implemented a dual extraction approach that tested models both with and without OCR preprocessing. This methodology allows us to assess whether current multimodal LLMs can effectively extract text directly from screenshots using their vision capabilities alone, and whether the addition of OCR-processed text enhances their performance.

For extractions without OCR assistance, we provided the LLMs with the Stack Overflow question metadata (title, body, and tags) alongside the screenshot, requiring the models to rely solely on their built-in vision-language capabilities to identify and extract relevant text. For extractions with OCR assistance, we augmented the prompt with an additional section containing pre-processed text extracted from the screenshot using a commercial OCR service. This dual approach enabled us to isolate the impact of OCR preprocessing on extraction performance across all evaluated models.

To ensure standardized image processing across all experiments, we implemented a consistent image handling pipeline. All screenshots were converted to base64 format for API compatibility, and we verified that each screenshot maintained its original resolution and aspect ratio throughout the process. For the OCR-assisted approach, we employed Google Cloud Vision OCR API, which has demonstrated robust performance in recognizing programming languages and maintaining code structure in previous studies [49].

3.3.3 Evaluation Framework

We developed a multi-faceted evaluation framework to assess question inference (RQ1), relevance detection (RQ2), and text extraction quality (RQ3). This comprehensive framework combines automated metrics with human evaluation to provide a thorough understanding of model capabilities across different aspects of programming screenshot analysis.

Metrics for Question Inference (RQ1)

For evaluating question inference capabilities (RQ1), we implemented a mixed-method approach:

⚙ We employed a vector-based embedding model to compute the *similarity between LLM-generated responses and the original questions*. We used `all-MiniLM-L6-v2`, a highly-rated and widely-used embedding model on HUGGING FACE [64]. We used this model to generate embeddings for both the original questions and LLM responses, and we calculated the similarity between these embeddings using cosine similarity [146].

👥 We used developers' perceptions to assess the practical applicability of our LLM-based approach. First, we examined the *relevance of the posted screenshot and the accompanying question* on Stack Overflow. Then, we measured both the *screenshot's relevance to the LLM-generated question* and the *alignment between LLM-generated responses and the original question* on Stack Overflow. To guide these evaluations, we employed three key questions:

Q1: To what extent does the LLM response relate to the image? (Likert scale 0-10)

Q2: To what extent does the LLM response relate to the original Stack Overflow question? (Likert scale 0-10)

Q3: To what extent is the Stack Overflow question related to the image posted? (Likert scale 0-10)

We randomly selected 50 images for evaluation, with two experienced developers (minimum four years in software development) scoring each screenshot on Q1, Q2, and Q3. We calculated the average score per question for each screenshot and normalized the results to a scale from 0 to 1. We used this survey to facilitate [14]. The Cohen's kappa agreement among the annotators was 0.86, indicating the level of inter-rater reliability. Any discrepancies greater than one point between the two developers were addressed by initially having them discuss and resolve their differences whenever possible. For remaining disagreements, the first author of the paper acted as a moderator to reach a final consensus.

Classification Metrics for Relevance Detection (RQ2)

For evaluating how accurately LLMs determine whether an image contains relevant text (RQ2), we employed standard binary classification metrics [95]. We established a clear framework for categorizing model performance by defining four fundamental outcome types. True Positives (TP) represent cases where the model correctly extracted text when the ground truth contained relevant text. True Negatives (TN) indicate situations where the model correctly identified no relevant text when the ground truth was indeed empty. False Positives (FP) occur when the model extracted text despite the ground truth being empty. Finally, False Negatives (FN) tell about instances where the model failed to extract text when the ground truth contained relevant text.

From these basic measures, we derived a comprehensive set of performance metrics for systematic evaluation. Accuracy ($\frac{TP+TN}{TP+TN+FP+FN}$) measures the overall correctness of the model across all test cases. Precision ($\frac{TP}{TP+FP}$) represents the proportion of positive identifications that were correct, indicating the model's ability to avoid false extractions. Recall ($\frac{TP}{TP+FN}$) quantifies the proportion of actual positives that were identified correctly, reflecting the model's sensitivity to relevant content. The F1 score ($2 \times \frac{Precision \times Recall}{Precision+Recall}$) provides the harmonic mean of precision and recall, offering a balanced assessment of model performance.

To ensure a robust and comprehensive evaluation, we assessed these metrics for each model under two distinct conditions: with and without OCR assistance. This dual approach allowed us to systematically determine whether pre-processed OCR text improved the models' ability to correctly identify the presence or absence of relevant text in programming screenshots. By comparing performance across these conditions, we could isolate the specific impact of OCR preprocessing on relevance detection capabilities.

Similarity Metrics for Extraction Quality (RQ3)

For evaluating text extraction quality (RQ3), we utilized established text similarity metrics to compare extracted text against ground truth annotations. This approach enabled

quantitative assessment of how closely each model's extractions matched the relevant text identified in our manually labeled dataset.

We employed the BLEU score [140] as one of our primary evaluation metrics. BLEU measures n-gram precision between the extracted text and ground truth, offering insight into the lexical accuracy of extractions. This metric is particularly valuable for assessing how well the models preserve the exact terminology and phrasing present in programming code and error messages. Higher BLEU scores indicate greater fidelity to the original text in terms of vocabulary and local phrase structure.

We complemented BLEU with a suite of ROUGE metrics [92] to capture different dimensions of extraction quality. ROUGE-1 (unigram overlap) measures word-level recall, while ROUGE-2 (bigram overlap) captures phrase-level similarities. ROUGE-L (longest common subsequence) evaluates fluency and structure by identifying the longest co-occurring sequence between the extraction and ground truth. ROUGE-L proved particularly valuable for code evaluation as it accounts for word order while allowing for gaps, which is essential when assessing code where line ordering matters but intervening lines might vary in significance [83].

To ensure fair assessment, these metrics were computed only for true positive cases — instances where both the model and ground truth indicated relevant text existed. This methodological decision prevented skewed results from cases where the model correctly identified that no relevant text existed, which would otherwise artificially alter the aggregate similarity scores. This selective analysis provided a more accurate picture of extraction quality when relevant text was correctly identified.

3.4 Experiment

3.4.1 Data and Baseline

Extending the dataset established from our previous work, we expanded the collection from 143 to 223 Stack Overflow posts by including more recent questions from November 2024 through January 2025 following the previous research. This expansion

provided a more diverse and representative sample of programming screenshots for evaluation. Each post in our dataset contains a single screenshot attachment featuring code or IDE screenshots, along with the associated question title, body, and tags.

To enable a more detailed analysis of the relevance and content extraction capabilities of LLMs, we introduced a manual labeling process for the entire dataset. The first two authors of the paper, both having extensive experience in programming and software development, independently examined each image to identify and extract the most relevant text segments that directly corresponded to the question being asked. This "related_text" field could be empty if the screenshot did not contain any text directly relevant to the question.

The initial independent labeling process yielded a Cohen's kappa score of 0.67, indicating substantial but not optimal agreement between annotators. This moderate level of agreement reflected the inherent challenges in determining precisely which text segments were most relevant to a given question, particularly in complex screenshots containing multiple code blocks or error messages.

To improve annotation reliability, the annotators systematically reviewed discrepancies in their labels. For annotations that were similar with only minor differences in wording or scope, they settled on randomly selecting one of the two annotations to ensure fairness in the final dataset. For cases with more significant disagreements, the annotators discussed and reached consensus on the most appropriate text extraction. This reconciliation process increased the final Cohen's kappa score to 0.81, representing strong inter-annotator agreement and establishing a reliable ground truth for our experiments.

The manual labeling process revealed that 83.6% of images in our dataset contained text directly relevant to the associated question, while 16.4% did not contain any specifically relevant text segments. This distribution allowed us to comprehensively evaluate both the relevance detection and extraction capabilities of the models under investigation.

In-Context Prompting Strategy
Task: Generate a realistic Stack Overflow post about the following IDE/code screenshot. **Context:** You are a programmer experienced in various technology stacks, encountering an issue in a project. The screenshot shows the problem without textual content. Generate a post that: 1. Follows Stack Overflow guidelines 2. Includes relevant code/IDE context 3. Clearly states expected vs. actual behavior 4. Matches Stack Overflow format

---

Few-Shot Learning Strategy
Task: As an expert developer and Stack Overflow analyst, generate a question based on this screenshot, following these examples:
Example 1: [Screenshot 1] TITLE: Example Title 1 BODY: Example detailed description 1
Example 2: [Screenshot 2] TITLE: Example Title 2 BODY: Example detailed description 2
Requirements: 1. Clear and concise question 2. Highlight key error messages 3. Match example format

---

Chain-of-Thought Strategy
Task: Analyze the screenshot and create a Stack Overflow question by following these steps:
1. Initial Observation - Visible elements in the screenshot - IDE/tool identification - Error messages/indicators
2. Problem Identification - Main technical issue - Components involved - Issue type (configuration/syntax/runtime)
3. Context Building - Required technical background - Framework/language versions - Potential causes
4. Solution Attempts - Possible fixes tried - Relevant documentation - Troubleshooting steps
5. Question Formulation - Clear title - Technical details - Answerable format

---

Output Format
TITLE: Generated Title BODY: Generated Body

FIGURE 3.2: Prompt used for inferring questions (Made a little short to make it concise)

3.4.2 Implementation details

We employed three state-of-the-art LLMs with multimodal capabilities: `Gemini-1.5-Pro` [175], `Llama- 3.2-90b-vision-preview` [179] accessed via Groq, and `GPT-4o` [65]. `Gemini` and `GPT-4o` are known for their superior multimodal processing capabilities, while `Llama-3.2` represents a prominent open-source alternative.

We standardized the generation parameters for all models to ensure consistency across experiments. We set the temperature (t) to zero to produce deterministic outputs and configured *top_p* to 0.8 and *top_k* to 40 to control the diversity of text generation. The maximum output length was capped at 2,048 tokens to maintain uniformity in response length. For implementation, we used the `Gemini` API directly for interfacing with `Gemini`, the `Groq` client for `Llama`, and the `OpenAI` API for `GPT`. All models processed `base64`-encoded screenshot images we extracted from Stack Overflow posts.

Precise Extraction Task
Task: I am showing you a Stack Overflow question and an image containing code. Your ONLY task is to: 1. Look at the image carefully 2. Identify ONLY the specific parts in the image that directly relate to the error or issue mentioned in the question 3. Extract ONLY those relevant code snippets or error messages from the image 4. Return ONLY the exact text from those relevant portions of the image 5. Maintain original formatting, line breaks, and indentation of the extracted text 6. Do not modify, enhance, or complete the code 7. Do not add comments, explanations, or solutions 8. Do not return irrelevant portions of code from the image
Question Title: {title}
Question Body: {body}
Tags: {tags}
[Optional OCR Text:] {ocr_section}
CRITICAL: Extract ONLY the specific text segments from the image that are directly related to the problem described in the question. If multiple relevant sections exist, separate them with a single line break. If no relevant text is found in the image, respond with "No relevant text found in image."

Output Format
{Relevant Text}

FIGURE 3.3: Prompt used for extracting relevant text from programming screenshots

For question inference (RQ1), we evaluated each model using three prompt types, as detailed in Figure 3.2. For the few-shot learning approach, we selected two examples manually from the most upvoted questions in our dataset [138, 139].

Based on our initial experiments with question inference (RQ1), we found that in-context learning consistently produced the highest similarity scores and developer-rated relevance across all evaluated LLMs. Given these findings, we opted to use in-context learning exclusively for our text relevance detection (RQ2) and text extraction tasks (RQ3).

For text extraction specifically, we employed the prompt shown in Figure 3.3, which directs the model to extract only the relevant text segments from the screenshot. This focused approach enabled more precise evaluation of the models' ability to identify and extract only the specific text that relates directly to the programming issue.

For OCR-assisted experiments in RQ2 and RQ3, we included an additional section in the prompt with pre-processed text from the image using Google Cloud Vision OCR API. This dual approach allowed us to systematically evaluate whether pre-processed OCR text improved the models' ability to correctly identify and extract relevant text.

To handle API rate limits, we added a few seconds delay after few posts based on

the particular models rate limiting requirements. We saved our results regularly, storing intermediate data every twenty iterations in JSON format. This helped us track progress and ensure we didn't lose data if the program stopped unexpectedly. All our data and code are available in our GitHub repository [52] for reproducibility.

3.5 Results

We present our evaluation across three research questions: inferring questions from programming screenshots (RQ1), identifying whether images contain relevant text (RQ2), and extracting the most relevant text from images (RQ3).

3.5.1 RQ1: Question Inference from Programming Screenshots

Embedding-Based Similarity Evaluation

We calculated similarity scores using embeddings from `all-MiniLM-L6-v2`, to evaluate the *semantic similarity between LLM-generated responses and original question* on Stack Overflow (⚙). For a more granular analysis, we separately compared similarity scores for question titles, bodies, and combined inputs to observe potential variations in model performance on shorter versus longer text. We consistently observed higher similarity scores when combining the title and body of questions. Specifically, `GPT-4o` with in-context learning achieved the highest average score of 0.59 for these combined inputs.

TABLE 3.1: Average text similarity of LLMs generated queries with the baseline for different prompts (⚙) (RQ1)

Unit	Prompt Technique	LLAMA	GEMINI	GPT-4o
Question title	In-Context Learning	0.20	0.49	0.47
	Chain-of-Thought	0.21	0.47	0.46
	Few-Shot Learning	0.04	0.36	0.42
Question body	In-Context Learning	0.26	0.52	0.53
	Chain-of-Thought	0.31	0.52	0.51
	Few-Shot Learning	0.05	0.45	0.49
Combined title & body	In-Context Learning	0.31	0.58	*0.59**
	Chain-of-Thought	0.35	0.58	0.57
	Few-Shot Learning	0.07	0.52	0.56

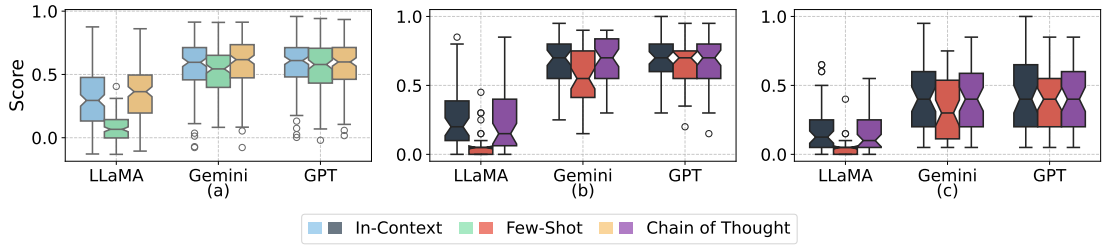


FIGURE 3.4: (a) Embedding-based similarity for generated and original question (⚙) (b) Relevance of generated question to screenshot by developers (👥) and (c) Relevance of generated to the original question by developers (👥) (RQ1)

Additionally, `GPT-4o` using chain-of-thought prompting and `Gemini` with in-context learning and chain-of-thought prompting performed only negligibly lower, with scores of 0.57 and 0.58.

In contrast, `Llama` consistently yielded lower similarity scores across various prompting techniques, underperforming relative to both `GPT-4o` and `Gemini`. Additionally, few-shot learning led to the lowest similarity scores, with averages ranging from 0.07 to 0.56 across models. This suggests that prompting style and the complexity of input texts (titles vs. bodies) play a significant role in the effectiveness of embedding-based similarity measures, yet the degree of similarity between the generated text, while promising, is still unclear. Overall, from Figure 3.4 and Table 3.1, it is evident that both `GPT-4o` and `Gemini-1.5-pro` mostly maintained similarity scores predominantly above 0.4, while `LLAMA-3.2` showed significantly lower and more variable performance with means of μ= 0.31, 0.07, and 0.36.

Developer-Assessed Usefulness

Two developers evaluated the relevance of LLM-generated questions to the screenshots (Q1) and their alignment with the original Stack Overflow questions (Q2). Our results, as shown in Figure 3.4 and summarized in Table 3.2, demonstrate varying performance across different LLMs and prompting techniques.

For the relevance to screenshots (Q1), both `Gemini` and `GPT-4o` performed strongly, achieving scores of $0.67 - 0.68$ with in-context learning and chain-of-thought prompting. This consistent performance indicates their robust capability in interpreting visual

content. `Llama`, in contrast, showed markedly lower relevance scores across all techniques, particularly struggling with few-shot learning, where it achieved only 0.06, suggesting significant limitations in its ability to effectively interpret screenshot context.

For the relevance of LLM-generated questions relative to the original Stack Overflow question (Q2), all models demonstrated a decrease in alignment, particularly in few-shot learning. `GPT-4o` achieved the highest scores in this metric ($0.39 - 0.43$ across all prompt techniques), showing a moderate but noticeable alignment with the original question. `Gemini`'s relevance to the original question was slightly lower than `GPT-4o`'s, scoring around $0.34 - 0.41$. `Llama` again had the lowest performance, particularly in few-shot learning, where its relevance score dropped to 0.04.

TABLE 3.2: Average developer perceived similarity of LLMs generated questions with the baseline (👥) (RQ1)

Unit	Prompt Technique	Llama	Gemini	GPT-4o
Q1: Relevance of LLM question to screenshot	In-Context Learning	0.28	*0.68**	0.67
	Chain-of-Thought	0.26	*0.68**	0.67
	Few-Shot Learning	0.06	0.56	0.65
Q2: Relevance of LLM question vs original	In-Context Learning	0.19	0.41	*0.43**
	Chain-of-Thought	0.17	0.39	0.40
	Few-Shot Learning	0.04	0.34	0.39

To better illustrate how models interpret and generate questions from programming screenshots, we developed an interactive application that showcases input-output examples across multiple scenarios. The tool presents screenshots from Stack Overflow questions and displays corresponding questions generated by GPT-4o, Gemini-1.5, and LLaMA-3.2 using various prompting strategies. The app is publicly available for reproducibility [14].

3.5.2 RQ2: Identifying Images with Relevant Text

For our second research question, we evaluated the models' ability to correctly identify whether a screenshot contains text relevant to the Stack Overflow question. Table 3.5 presents the classification metrics for each model, both with and without OCR assistance.

Gemini achieved the highest overall performance with an F1 score of 0.9116, demonstrating exceptional recall (0.9821) while maintaining good precision (0.8505). Interestingly, Gemini's performance remained identical with and without OCR assistance, suggesting that either its vision capabilities are sufficiently advanced to identify relevant content directly from images, or it simply does not take into account the given OCR content at all, relying solely on the visual input and the question.

Llama showed similarly strong performance with an F1 score of 0.9086, also exhibiting no difference between OCR-assisted and non-OCR approaches. This consistency suggests that either Llama's vision capabilities are sufficiently advanced to identify relevant content directly from images, or it simply does not take into account the given OCR content at all, relying solely on the visual input and the question. However, we observed that Llama frequently added prefixes like "Relevant Text:" or "Code:" to its output despite explicit instructions not to do so, indicating some limitations in following precise formatting instructions.

In contrast, GPT-4o demonstrated the highest precision (0.8974) when provided with OCR assistance, but its overall performance was lower than the other models, with F1 scores of 0.8642 (with OCR) and 0.8349 (without OCR). The notable difference between GPT-4o's OCR-assisted and non-OCR performance suggests that it benefits significantly from the additional context provided by pre-processed text, particularly for improving recall.

Examining the confusion matrix components in Tables 3.3 and 3.4 reveals concerning patterns that qualify these seemingly impressive performance metrics. While Gemini and Llama achieved high F1 scores, they demonstrated a strong bias toward generating output regardless of relevance. Gemini produced only 4 false negatives compared to 32 false positives, while Llama generated just 10 false negatives versus 26 false positives. Gemini and Llama appear to err on the side of inclusion — labeling most images as relevant and thereby maintaining very low false negative rates. However, this tendency does not mean they extract relevant content from every image accurately as shown in Section 3.5.3. In practical applications, this could result in users being overwhelmed

TABLE 3.3: Confusion Matrix for LLM Models With OCR (RQ2)

Model	True Positive	False Negative	False Positive	True Negative
Gemini	183	4	32	4
Groq	176	10	26	11
GPT	155	31	18	19

TABLE 3.4: Confusion Matrix for LLM Models Without OCR (RQ2)

Model	True Positive	False Negative	False Positive	True Negative
Gemini	183	4	32	4
Groq	176	10	26	11
GPT	149	38	21	15

with irrelevant information.

`GPT-4o`, despite its lower overall F1 score, exhibits a more balanced approach to the task with 31 false negatives and 18 false positives (with OCR). This behavior suggests it applies more conservative criteria when labeling content as relevant. Such conservatism may be advantageous in scenarios where minimizing irrelevant results (high precision) is more critical than capturing all possible relevant content (high recall). The difference in behavior indicates that while `Gemini` and `Llama` optimize for high recall at the cost of precision, `GPT-4o` prioritizes a more measured assessment of relevance.

3.5.3 RQ3: Extracting Relevant Text from Programming Screenshots

For our third research question, we evaluated the ability of LLMs to extract the most relevant text from programming screenshots. We computed similarity metrics between the extracted text and the ground truth annotations for all true positive cases—instances

TABLE 3.5: Classification metrics for identifying images with relevant text (RQ2)

Model	OCR	Accuracy	Precision	Recall	F1 Score
GEMINI	With	0.84	0.85	*0.98**	*0.91**
	Without	0.84	0.85	*0.98**	*0.91**
LLAMA	With	0.84	0.87	0.95	0.91
	Without	0.84	0.87	0.95	0.91
GPT-4o	With	0.78	*0.90**	0.83	0.86
	Without	0.74	0.88	0.80	0.83

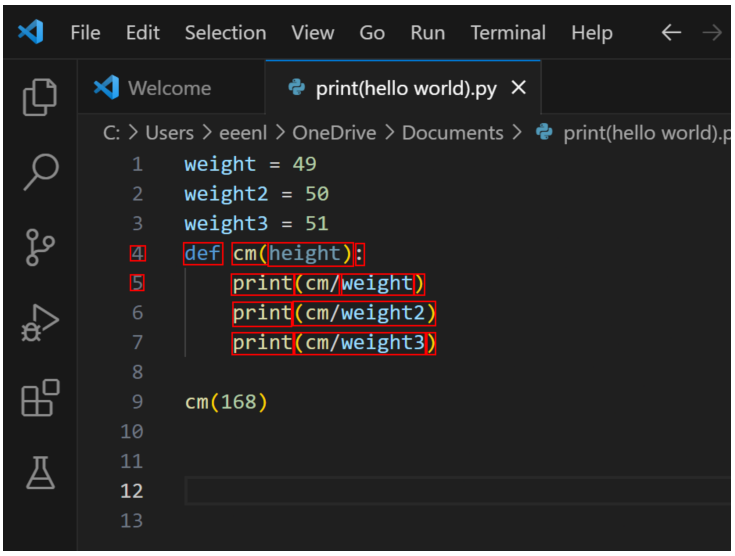


FIGURE 3.5: Example of precise relevant code highlighting that identifies the exact part where the question is targeting

where both the model and ground truth indicated relevant text existed. Table 3.6 presents these results for each model, both with and without OCR assistance.

`GPT-4o` demonstrated superior extraction quality once it correctly identified relevant text, despite its lower performance in relevance detection (RQ2). It achieved the highest scores across all similarity metrics when assisted by OCR, with a BLEU score of 0.4636, ROUGE-1 of 0.6820, ROUGE-2 of 0.6412, and ROUGE-L of 0.6749. This suggests that `GPT-4o` excels in precise extraction when it has both visual and textual information available.

`Gemini` performed comparably well in extraction quality, with scores only slightly lower than `GPT-4o` across all metrics. Similar to its performance in relevance detection,

TABLE 3.6: Text similarity metrics for extracted relevant text (RQ3)

Model	OCR	BLEU	ROUGE-1	ROUGE-2	ROUGE-L
GEMINI	With	0.45	0.64	0.59	0.63
	Without	0.45	0.64	0.59	0.63
LLAMA	With	0.13	0.25	0.19	0.24
	Without	0.13	0.25	0.19	0.24
GPT-4o	With	*0.46**	*0.68**	*0.64**	*0.67**
	Without	0.46	0.68	0.63	0.67

Gemini showed minimal difference between OCR-assisted and non-OCR approaches, reinforcing the robustness of its vision-language capabilities.

Llama, despite its strong performance in relevance detection, exhibited significantly lower extraction quality compared to the other models. With BLEU scores of 0.1291 and ROUGE scores ranging from 0.1923 to 0.2542, Llama struggled to precisely extract the specific relevant text segments, often including extraneous information or missing crucial details. Additionally, even with explicit instructions, Llama frequently added formatting prefixes like "Relevant Text:" or "Code:" to its extractions, requiring post-processing to remove these artifacts. This inconsistency in following output format instructions highlights a significant limitation in Llama's practical application for text extraction tasks.

An interesting observation across all models is that the OCR assistance had minimal impact on extraction quality for Gemini and Llama, while it provided a noticeable benefit for GPT-4o. This pattern mirrors our findings from RQ2, suggesting consistent behavior across different aspects of the image processing pipeline. The minimal improvement from OCR for Gemini and Llama suggests that these models have developed strong internal representations for processing text within screenshots, whereas GPT-4o appears to benefit from the additional structured text input. An example of how this relevant text can be highlighted on the image is shown in Figure 3.5

3.6 Threats to Validity

3.6.1 Internal Validity

Our experimental design introduces several potential threats to internal validity. First, using recent Stack Overflow data minimizes the likelihood of LLMs being explicitly trained on our specific examples, though this cannot be definitively verified [179, 175]. This potential lack of specific data exposure could partially explain observed performance variations.

We identified a moderate correlation (average of 0.53) between the relevance of posted screenshots to original questions and the similarity between LLM-generated questions and original questions. This suggests that inconsistent image-question alignment may influence the evaluation results, potentially affecting the causal relationship we're investigating.

Despite explicit formatting instructions, we observed notable inconsistencies in outputs, particularly with `Llama`, which frequently added unauthorized prefixes to responses. This necessitated post-processing to standardize outputs, highlighting limitations in following precise output format instructions [195, 44] and potentially introducing experimental bias.

3.6.2 External Validity

Our dataset size ($n = 223$), while expanded from previous work, still constrains the study's scope, potentially limiting generalizability across diverse programming contexts [190, 55].

Manual analysis of the best and worst-performing images revealed that optimal results occurred with screenshots containing both error messages and code with explicit errors. High-performing cases often included relevant background information and user annotations highlighting specific issues [19, 89]. Conversely, poor performance occurred with screenshots displaying isolated code snippets or error messages without IDE context, or with multiple complex elements causing model confusion [3, 86]. This suggests findings may not generalize to all types of programming screenshots.

The study's focus on specific LLMs (`GPT-4o`, `Gemini`, and `Llama`) means results may not generalize to other multimodal models or future iterations of these models.

3.6.3 Construct Validity

For question inference (RQ1), embedding similarity effectively captures semantic meaning rather than exact word matches [75, 27, 146]. For relevance detection (RQ2) and text

extraction (RQ3), we employed classification metrics and text similarity measures appropriate for these tasks [95, 140, 92]. However, these metrics may not fully capture nuances of code text extraction, particularly for programming languages with significant syntactic importance [99, 2].

A primary threat to construct validity lies in how we operationalize "relevance" for binary classification. Relevance of text in programming screenshots is inherently subjective and context-dependent, but our ground truth uses binary labels derived from annotator judgment. This simplification may fail to capture nuanced developer needs or alternative valid interpretations of what constitutes "relevant" content. Furthermore, our evaluation relies on proxy metrics (precision, recall, F1), which may not fully represent a model's deeper semantic understanding or practical usefulness.

3.6.4 Conclusion Validity

The relevance scores for screenshots to original questions (average relevance score of 0.69, as shown in Figure 3.6) were based on human evaluation, which introduces potential subjectivity and inter-rater reliability concerns.

While we identified a moderate correlation between screenshot relevance to original questions and LLM-generated question similarity to original questions, correlation does not necessarily imply causation, and other factors may influence these relationships.

3.7 Discussion

3.7.1 Relevance of Posted Images to Original Questions and its impact on LLMs performance

In addition to evaluating the relevance of LLM-generated questions to both the screenshot and the original Stack Overflow question, we assessed the alignment between the posted Stack Overflow question and the associated image (Q3). Figure 3.6 illustrates this distribution, showing an average relevance score of 0.69 while indicating that some

screenshots have low relevance to the posted Stack Overflow question, suggesting inconsistencies in the alignment between visual content and the accompanying text-based query.

We further examined whether the relevance of the posted screenshot to the original question (Q3) correlates with the similarity between the LLM-generated question and the original question (Q2). Suppose the posted screenshot is only loosely related to the original question. In that case, it follows that the LLM-generated query—based solely on the screenshot—might also diverge from the original question. We identified a moderate correlation between these two factors (average of 0.53), suggesting that while the LLM-generated question does partially reflect the original question's intent when the screenshot is relevant, this alignment is not consistently strong. This finding highlights a moderate performance and potential limitation in the LLM's ability to interpret questions raised in programming screenshots independently.

3.7.2 Impact of OCR Preprocessing

Our dual evaluation approach revealed `GPT-4o` showed notable improvements with OCR preprocessing, while `Gemini` and `Llama` demonstrated minimal differences. This suggests these latter models have developed robust internal vision-language capabilities [8, 20]. These findings have important implications for system design decisions—particularly whether to incorporate computationally expensive OCR preprocessing [49, 37].

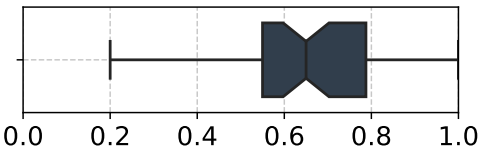


FIGURE 3.6: Image relevance to questions on Stack Overflow (👥)

3.7.3 Model Output Inconsistencies

Despite explicit formatting instructions, we observed notable inconsistencies in outputs, particularly with `Llama`, which frequently added unauthorized prefixes to responses. This necessitated post-processing to standardize outputs, highlighting limitations in following precise output format instructions [195, 44]. Such inconsistencies present challenges for implementing these models in production systems where consistent formatting is crucial [135, 59].

3.7.4 Practical Implications for Developer Forums

Our findings suggest that while current multimodal LLMs show promise for bridging visual and textual content on developer forums, they are not yet ready for fully autonomous deployment [69, 141]. Rather than replacing text-based communication, visual analysis tools could serve as complementary features, automatically generating text-based versions of screenshot content for validation by users before posting [21, 54, 13].

3.8 Can LLMs Bridge the Gap Between Visual and Textual Content in Developer Forums?

In this study, we investigated how effectively large language models can process programming screenshots for technical problem-solving on platforms like Stack Overflow. Our initial exploration examined whether LLMs can infer questions from screenshots alone, with our evaluation demonstrating that state-of-the-art models, particularly `Gemini` and `GPT-4o`, show promising capabilities. These models achieved embedding-based similarity scores up to 0.59 and perceived relevance scores from developers reaching 0.69. Our best-performing model, `GPT-4o`, generated questions with over 60% similarity to the baseline for 51.75% of the tested images, with in-context learning consistently yielding the highest relevance scores across models.

Building on these findings, our expanded research moved beyond question inference to address the crucial tasks of identifying and extracting relevant content from screenshots. Our evaluation of relevance detection capabilities revealed that `Gemini` achieved exceptional performance with an F1 score of 0.9116 and a recall rate of 0.9821, demonstrating its robust ability to determine whether screenshots contain text relevant to a given question. Interestingly, both `Gemini` and `Llama` showed consistent performance regardless of OCR assistance, suggesting their vision-language integration has reached a level where external text processing provides minimal benefits. In contrast, `GPT-4o` demonstrated notable improvements with OCR assistance, highlighting differences in how these models process visual textual content.

For the critical task of extracting specific relevant text from screenshots, `GPT-4o` demonstrated superior capabilities when correctly identifying relevant sections, achieving the highest scores across all similarity metrics (BLEU: 0.4636, ROUGE-1: 0.6820) when assisted by OCR. This extraction capability is particularly valuable for automatically converting screenshot content into searchable, indexable text while preserving the contextual benefits that developers seek when sharing screenshots.

We observed consistent patterns in how content characteristics affected model performance across all three research questions. Screenshots containing clear, isolated content with explicit error messages consistently led to better question inference, more accurate relevance detection, and higher-quality text extraction. The challenges posed by complex, multi-element screenshots affected all evaluated tasks, suggesting common limitations in current multimodal processing capabilities.

Our findings demonstrate that while current LLMs show promising abilities to bridge the gap between developers' tendency to share screenshots and platforms' need for searchable text content, several challenges remain. The inconsistent output formatting observed with `Llama`, the varying impact of OCR preprocessing across models, and performance differences with complex screenshots all indicate areas where improvements are needed.

Looking forward, we envision a future where visual-based debugging copilot tools

could transform how developers seek and receive assistance, automatically extracted and highlighed relevant content from screenshots while generating searchable text representations. With continued advances in multimodal understanding and targeted fine-tuning for programming tasks, LLMs could effectively serve as intermediaries that maintain the convenience of screenshot sharing while addressing the technical limitations that platform guidelines aim to prevent.

3.9 Replication Package

For the reproducibility of this study and open science, we provide the community with all the artifacts used in our study. In particular, we provided the source code of our model with documentation and our complete dataset. The project including all artifacts is available at this gtihub link[⋆1] [53].

[1] ⋆ All artifacts related to this study are available at https://github.com/Research-Purpose/image-citation/tree/main

Chapter 4

COntext aware Recognition from Text and codE (CORTEx) for Visual Information Mining

Abstract

Developers frequently share screenshots when posting questions on Q&A platforms, yet the responses they receive rarely incorporate visual cues, forcing readers to interpret details from text alone. Inspired by the growing trend of visual communication and advancements in image processing, our study explores how effectively images can be perceived and interpreted by state-of-the-art methods and Large Language Models (LLMs). To this end, we introduce *CORTEx*, a novel method designed to identify programming context and extract both code and text from images posted on Q&A platforms. CORTEx works by segmenting the image into various sections and identifying text within specific areas, significantly improving accuracy. Using Stack Overflow as our case study, we compare *CORTEx's* performance against Google Vision OCR, LLMs like ChatGPT-4o and Gemini 1.5 Pro, and state-of-the-art methods such as PSC2code and CodeMotion in interpreting code-related images. Our results show that *CORTEx* outperforms existing methods and LLMs, with an average increase of 15% in BLEU and ROUGE scores. This

improvement demonstrates *CORTEx's* superior ability to accurately extract and interpret information from code-related images. To complement our quantitative findings, we conducted a qualitative study to evaluate developers' perceptions of the segmented image layouts produced by *CORTEx*, which showed that the developers preferred *CORTEx* over state-of-the-art methods by an average of 70% overall. We also assessed how developers viewed the format and structure of *CORTEx's* output compared to various LLMs and other state-of-the-art methods in which LLMs did outperform *CORTEx* but they often missed important text from the images, which again proves the importance of a method like *CORTEx*. By addressing these questions, our research aims to enhance the clarity, accessibility, and usefulness of text extracted from code-related images. These images often contain a wealth of knowledge and can provide invaluable insights for questions posted on Q&A platforms, particularly in areas such as debugging and issue identification.

4.1 Introduction

The performance of current image processing techniques and automated text extractors, also known as Optical Character Recognition (OCR) systems, for reading programming screenshots remains unclear. Programming screenshots, which often involve IDE issues or contain only code, have a distinct structure that differentiates them from generic object images. These screenshots typically feature clear borders and contextual information, where each section conveys specific details such as syntax errors, runtime exceptions, compiler warnings, execution outputs, or debugging insights. This structured nature presents unique challenges for OCR and image processing methods, as they must accurately capture the textual content, spatial relationships, and semantic meaning embedded within the screenshot. This is while the studies have shown an increasing trend of sharing images on social coding platforms for bug triaging and problem solving [111, 112].

The use of image-based communication in programming discussions has more than doubled between 2013 and 2022, with developers increasingly relying on screenshots

to illustrate complex technical issues [111]. This trend signifies a broader shift in communication practices, where visual elements play a crucial role in articulating programming challenges, debugging processes, and demonstrating solutions. Studies indicate that approximately 86.9% of posts containing images on platforms like Stack Overflow require visual context for full comprehension [112].

However, a *critical gap* exists in current development workflows, as valuable information embedded within images often remains underutilized or inaccessible to automated tools designed for programmatic analysis. This challenge is further exacerbated by the diverse nature of programming screenshots, which may contain not only code snippets but also error messages, console outputs, and IDE interface elements—each requiring distinct processing techniques for accurate information extraction.

We introduce CORTEx (COntext aware Recognition from Text and codE), a novel approach specifically designed for processing code-related images on Q&A platforms. CORTEx represents a significant advancement over existing methods by employing a context-aware processing pipeline that combines sophisticated image segmentation techniques with advanced OCR capabilities. Unlike previous approaches that treat code extraction as a purely textual problem, CORTEx incorporates layout analysis and structural understanding to maintain the semantic integrity of the extracted content.

But first, let us explain why we need CORTEx. We identified that current Large Language Models (LLMs) when asked to extract complete text from the images sometimes hallucinate and generate extra code or miss some part of the images. Also, some methods like CodeMotion[77] and PSC2Code[19] focus on screencasts from YouTube videos and do not solely work with individual images even though they do work with images and thus, we created our own method, *CORTEx* which is designed to work with individual images posted on Q&A platforms like Stack Overflow.

4.2 Background and Motivation

4.2.1 OCR Tools for Text Extraction from Programming Screenshots

The performance of current image processing techniques and automated text extractors, also known as Optical Character Recognition (OCR) systems, for reading programming screenshots remains unclear. Programming screenshots, which often involve IDE issues or contain only code, have a distinct structure that differentiates them from generic object images. These screenshots typically feature clear borders and contextual information, where each section conveys specific details such as syntax errors, runtime exceptions, compiler warnings, execution outputs, or debugging insights. This structured nature presents unique challenges for OCR and image processing methods, as they must accurately capture the textual content, spatial relationships, and semantic meaning embedded within the screenshot. This is while the studies have shown an increasing trend of sharing images on social coding platforms for bug triaging and problem solving [111, 112].

Deep learning-based Optical Character Recognition (OCR) techniques have traditionally been trained on datasets like MNIST, which comprises 28×28 pixel grayscale images of handwritten digits ranging from 0 to 9. While these models excel in recognizing handwritten numerals, their performance on screenshots containing typed programming code is less certain. Empirical studies have demonstrated that general-purpose OCR engines often struggle to accurately transcribe source code from images, highlighting the necessity for specialized OCR solutions tailored to programming languages' unique syntax and structure [97].

Malkadi et al. [97] conducted a comprehensive evaluation of six OCR engines—`Google Drive` OCR [**Memon2020Handwritten**], `ABBYY FineReader` [17], `GOCR` [42], `Tesseract` [162], `OCRAD` [**Memon2020Handwritten**], and `Cuneiform`—to assess their performance in processing programming screenshots. Their findings revealed substantial performance variations among these OCR systems. Notably, `Google Drive` OCR and `ABBYY FineReader` exhibited the highest accuracy, whereas `Tesseract`, despite its widespread

adoption, did not rank among the top-performing engines. The unique formatting and syntax of programming languages, combined with the presence of distinct contextual panels in various Integrated Development Environments (IDEs), further complicate OCR-based text extraction. Critical elements such as indentation, special characters, and language-specific keywords necessitate a high degree of precision to ensure accurate recognition. Any misinterpretation of these components can result in syntactic errors, loss of structural integrity, or unintended modifications, ultimately rendering the extracted code non-functional or introducing defects that affect its execution.

Therefore, improving OCR performance for programming screenshots requires specialized techniques that address the unique structural and syntactical characteristics of source code [136]. One of the most critical aspects of code extraction from images is preserving the original layout and contextual information that developers rely on for code comprehension. Programming screenshots contain rich contextual cues that extend beyond the raw text content, including indentation patterns, spatial relationships between code blocks, line numbers, file structures, and IDE interface elements. These visual elements are not merely aesthetic; they carry semantic meaning that is essential for understanding code structure, identifying problematic areas, and maintaining the logical flow of programming constructs.

4.2.2 Screencast-Based Code Extraction Tools

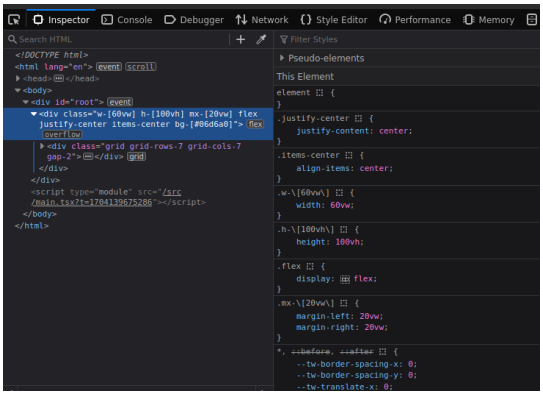
An emerging research direction focuses on extracting code and other relevant content from programming video tutorials. `CodeMotion` [77] and then `PSC2Code` [19], have been developed to extract code from programming screencasts. Both `CodeMotion` and `PSC2Code` leverage temporal analysis by comparing multiple frames to infer code structure and execution flow. However, static screenshots lack a sequential reference for comparison, necessitating their independent interpretation. This distinction introduces additional complexities in automating the extraction and comprehension of programming screenshots.

CodeMotion is a computer vision algorithm developed to enhance the interactivity of programming tutorial videos. It processes raw screencast videos to identify regions containing source code, applies OCR to convert the visual code into editable text, and tracks code edits over time to group related changes into continuous intervals. The *input* to CodeMotion is a programming tutorial video, and its *output* includes extracted source code snippets and their corresponding edit histories, enabling dynamic and interactive learning experiences.

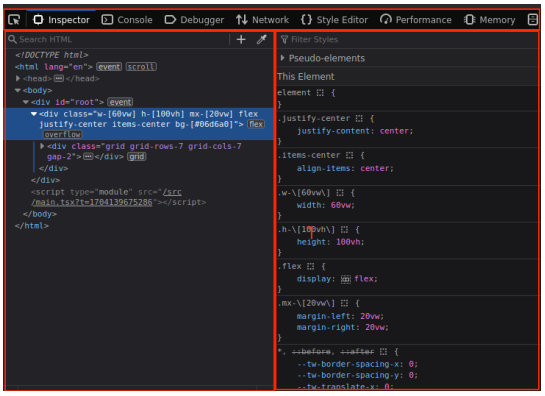
Building upon CodeMotion, PS2Code is a tool designed to further enhance the learning experience by integrating the extracted code directly into the video interface. PS2Code takes the output from CodeMotion—the segmented code snippets and their edit intervals—and synchronizes them with the video playback. This allows learners to interact with the code in real-time, providing features such as inline code editing, code-based navigation, and in-video coding exercises. By leveraging CodeMotion's capabilities, PS2Code transforms passive video tutorials into active learning platforms, facilitating a more engaging and hands-on approach to mastering programming concepts.

However, both CodeMotion and PSC2Code face significant challenges when applied to static screenshots. Their algorithms are optimized for dynamic video content, where temporal information aids in accurately detecting and extracting code regions. When used on single, static images, these tools often misidentify or fail to detect code areas correctly. This can result in fragmented, merged, or missed code segments, leading to outputs that are unclear and unstructured. Consequently, the extracted code may be incomplete or jumbled, making it difficult to discern whether it originates from the code panel, debug panel, or other sections of the integrated development environment (IDE). As such, the code retrieved by these tools also lacks programming context.

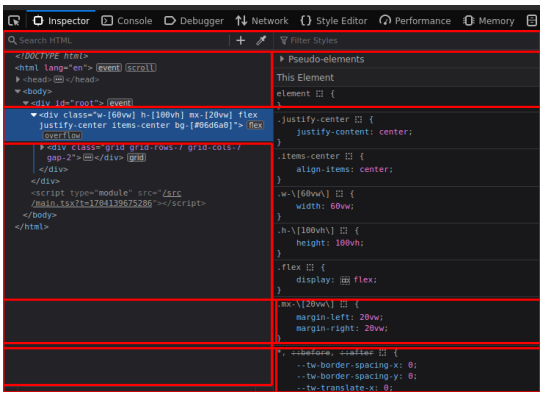
In Figure 4.1, we present a comparative analysis of CodeMotion [77] and PSC2Code [19] applied to a static image sourced from Stack Overflow. The red outlines in Figure 4.1-(c) and Figure 4.1-(d) highlight the regions automatically detected by PSC2Code and CodeMotion, identifying 10 and 25 areas, respectively. In contrast, when experienced



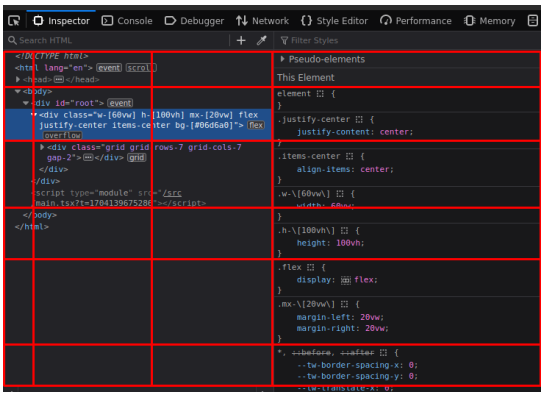
(A) Input Image



(B) Human Sectioned Image



(C) PSC2Code Sectioned Image



(D) CodeMotion Sectioned Image

FIGURE 4.1: Layout generation on a single (a) Input Image from Stack Overflow by (b) Human, (c) PSC2Code [19], and (d) CodeMotion [77]

programmers manually annotated the same screenshot, they identified only three primary regions, as depicted in Figure 4.1-(b). This discrepancy shows a significant limitation in both tools: *their tendency to over-segment static images, leading to fragmented and imprecise code extraction.* Consequently, the output often becomes ambiguous, making it challenging to discern whether the extracted code originates from the code panel, debug panel, or other sections of the IDE.

4.2.3 Use of LLMs in Visual Code Understanding

Developers understand code in context [76], making the preservation of spatial relationships and visual hierarchies crucial for effective code extraction tools. Traditional OCR approaches often fail to maintain these relationships, treating text extraction as a purely

sequential process that loses the two-dimensional structure inherent in programming interfaces. This limitation becomes particularly problematic when dealing with complex IDE screenshots that contain multiple panels, each serving different purposes such as code editing, debugging output, file navigation, and error reporting.

The challenge is further compounded by the diverse nature of programming environments and coding practices. Different IDEs present information in varying layouts, font sizes, and color schemes, while programming languages have distinct syntactic requirements for indentation and spacing. Additionally, developers frequently include annotations, highlights, or selections in their screenshots to draw attention to specific problematic areas, creating additional visual context that must be preserved and interpreted correctly.

Current Large Language Models (LLMs), while powerful in many text processing tasks, exhibit significant limitations when applied to code extraction from images. These models often hallucinate or generate additional code that was not present in the original image, a behavior stemming from their extensive training on code repositories that leads them to “fill in" perceived gaps based on common patterns [50, 169]. Furthermore, LLMs frequently fail to capture all parts of the code present in images, resulting in incomplete extraction that can alter the meaning or functionality of the extracted code [180]. The inconsistency in LLM outputs, where the same input image can produce different results across multiple runs, further undermines their reliability for precise code extraction tasks [98].

LLMs often face challenges in accurately preserving the structural elements and extracting the complete text from code images. While they may capture the semantic content of the code, crucial visual components are frequently lost or misrepresented. A significant issue is the omission of line numbers, which are commonly present in code screenshots and serve as critical reference points, especially in contexts like Stack Overflow discussions. Additionally, LLMs may fail to capture or accurately represent the file structure or Integrated Development Environment (IDE) interface elements visible

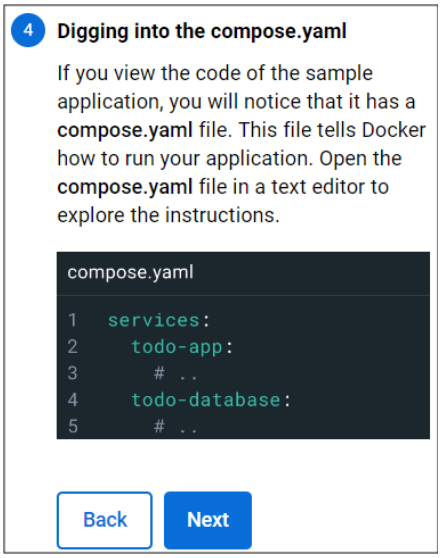


FIGURE 4.2: Example of text omission by `GPT-4o` in image-to-text extraction in 1 out of 3 tries. The left image displays the original code screenshot, while the right shows `GPT-4o`'s extracted output, demonstrating the model's failure to capture crucial textual content from the top portion of the image.

in the image. Furthermore, when code snippets are accompanied by other textual information (such as annotations or additional info by the developer or other IDE panels), LLMs often extract only the code, neglecting the surrounding context. These limitations highlight the need for specialized tools that can effectively interpret and extract both the code and its contextual elements from static images. These omissions are particularly problematic because users frequently refer to specific line numbers or file locations or choose to add annotations of screenshots when describing their coding issues [29].

Another concerning aspect is the variability in output produced by LLMs. We observed that these models can generate different outputs for the same input image across multiple runs [98, 7, 16]. In Figure 4.2, we illustrate the inconsistency of Large Language Models (LLMs) in processing code screenshots. Despite being provided with the same prompt, the LLM omitted content from the screenshot in one out of three attempts, focusing solely on the code snippet and disregarding surrounding text. This variability highlights a significant limitation of LLMs: their tendency to inconsistently capture all textual elements within an image, even when explicitly instructed to do so.

These limitations of LLMs in code extraction from images underscore the need for specialized approaches like *CORTEx*. By focusing specifically on the task of code extraction and layout preservation, *CORTEx* aims to address these issues, offering a more accurate, consistent, and explainable solution for processing code-related images on Q&A platforms like Stack Overflow. The development of *CORTEx* is driven by the need to overcome these challenges and provide a reliable tool for developers and researchers working with code embedded in visual formats.

A critical advantage of *CORTEx* lies in its comprehensive approach to image analysis, ensuring that all parts of the image are taken into account. Unlike `PSC2Code` and `CodeMotion`, which sometimes overlook certain areas of complex screenshots, *CORTEx* implements a thorough scanning and segmentation process. This meticulous approach is particularly valuable when dealing with screenshots from IDEs or console outputs, which often contain multiple regions of interest such as code blocks, file structures, and error messages. By carefully analyzing the entire image, *CORTEx* minimizes the risk of missing crucial information that users might reference in their questions. This comprehensive coverage is especially important on platforms like Stack Overflow, where users frequently share screenshots containing not just code, but also surrounding context like line numbers, file paths, or console outputs. *CORTEx's* ability to capture and preserve all these elements provides a more complete and accurate representation of the user's problem, facilitating more effective problem-solving and communication within the developer community.

4.3 Related Work

4.3.1 Utilization of images in Software Engineering Platforms

The importance of visual content in software engineering has grown significantly, with studies showing a doubling of image-inclusive posts on platforms like Stack Overflow between 2013 and 2022 [111]. Images are often crucial for understanding, with 86.9% of cases unlikely to be comprehended without them [112]. This trend extends to software

testing, where image-based techniques like Browser Bite for cross-browser compatibility testing [160] and TANGO for identifying duplicate bug reports [41] demonstrate improved accuracy and efficiency. Ahmed et. al. [5] show the incorporation of images in the textual-based methodology of Ahsanuzzaman et. al.[4].

A study identified 19 influential image features for classifying UML class diagrams, achieving high prediction rates [61]. Questions with images on platforms like Stack Overflow are more likely to receive accepted answers [187], with code snippets positively correlating with question success [34]. Additionally, the WebUI dataset introduces extracted visual information from web pages to improve UI understanding models, particularly beneficial for mobile applications with limited labeled data [199]. `GIFdroid` uses screen recordings to automatically generate execution traces for bug reports [47], while a novel unsupervised approach applies perceptual grouping principles to GUI element clustering [200]. These developments highlight the increasing relevance of visual content and image-based approaches across various aspects of software engineering.

4.3.2 Code and Text Retrieval from Images

Recent research has made significant progress in mining and analyzing source code from images and videos, addressing the challenge of accessing code embedded in visual media. Some recent work utilized deep learning, specifically convolutional neural networks (CNNs), to identify typeset and handwritten source code in video frames with high accuracy [136, 137]. This demonstrated the potential of deep architectures to learn and differentiate lexical features of programming languages from images or screenshots. Subsequent studies introduced curated datasets of programming tutorial frames [24] and explored transfer learning to overcome limited data availability in software engineering contexts [25, 80].

Building on this foundation, researchers developed more comprehensive approaches to code extraction from videos. These methods combined CNN-based image classification, edge detection, clustering-based image segmentation, and optical character recognition (OCR) to filter non-code frames, identify code regions, and extract the final code

[19]. Comparative studies of OCR engines for source code transcription revealed variations in accuracy among different tools and highlighted the impact of font choices on recognition quality [97]. Advanced CNN-based methods were developed to precisely locate code areas within video frames, achieving high accuracy in code detection and improving subsequent OCR results [9, 8].

In recent years, document understanding and recognition have been active areas of research, driven by the need to automate the processing and analysis of business documents [49]. This area developed from rule-based approaches to machine learning and, more recently, deep learning techniques. include the introduction of pre-trained language models like BERT [181]. In the meanwhile, LayoutLM was proposed to jointly model text and layout information in a single framework [202]. By incorporating both textual and visual features, LayoutLM-series models achieved state-of-the-art results in understanding scanned documents like forms, receipts, and other business documents, highlighting the importance of layout in the images [202, 201, 63].

4.3.3 Multi-modal Models and Large Language Models in Document Understanding

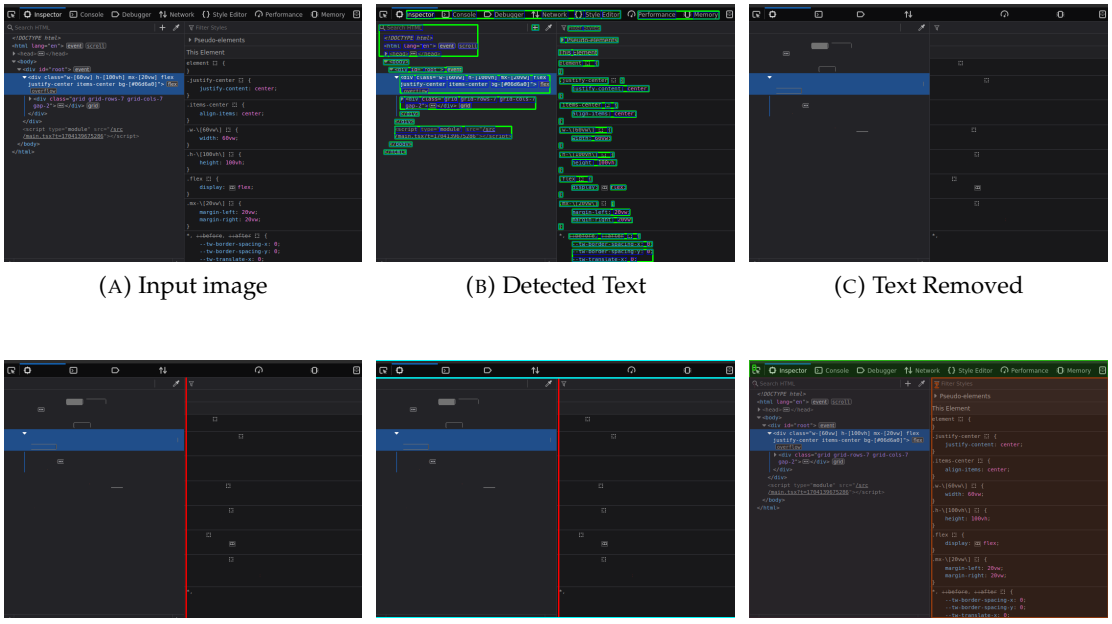
Pre-training base models on large datasets and fine-tuning with specific datasets are commonly used to solve image and text understanding problems [170, 89]. Multi-modal learning, which combines textual, visual, and layout information, has shown a great opportunity in document understanding. Models like LayoutLM v2 and v3 leverage transformers to process multi-modal inputs, achieving improved performance. [202, 201, 63]. Optical character recognition (OCR) engines play a crucial role in transcribing text from images, especially extracting source code from programming screencasts [49].

In recent years, multimodal document understanding has become a significant area of research, leveraging both textual and visual information for better comprehension. LayoutLMv2 by Xu et al. [201] introduced a multi-modal pre-training model specifically designed for visually rich documents by integrating both text and layout information. This approach was expanded further in DocLLM by Wang et al. [188], which

incorporated layout awareness in generative language models to enhance document comprehension. Similarly, Pix2Struct by Lee et al. [87] proposed pretraining through screenshot parsing, focusing on visual language understanding without relying heavily on optical character recognition (OCR). Kim et al. [78] in their OCR-free Document Understanding Transformer eliminated OCR dependencies, advocating for a more streamlined document understanding pipeline. Further advancements include DocFormer by Appalaraju et al. [15], which leveraged an end-to-end transformer architecture for document processing, and GeoLayoutLM by Luo et al. [94], which emphasized geometric pre-training for extracting visual information from documents. Lastly, UniDoc by Gu et al. [56] introduced a unified pretraining framework that harmonizes textual and layout information for document understanding tasks, pushing the boundaries of cross-modal pretraining. These works collectively highlight the advancements in integrating visual and textual modalities for enhanced document understanding.

4.4 CORTEx Methodology

Developers understand code in context [76], hence we needed a method that segments the image into various sections which makes it easier to isolate various parts of the image and pinpoint the parts the user is talking about in their question in the images and that could hopefully be used to provide the users with visual solutions which is the ultimate goal of *CORTEx*. But, before that, we should be able to segment the image and extract correct and accurate text and code with their relative location to be able to properly structure and format the extracted text. To this end, *CORTEx* works in 5 main phases: Detecting text, Removing those text, Detecting vertical lines and Horizontal lines, Retrieving layout structure, and finally putting the text back by associating them with Sections. These various stages are clearly shown in Figure 4.3 and the overview of the process and methodologies are shown in Figure 4.4.



(A) Input image (B) Detected Text (C) Text Removed

(D) Vertical Lines Detected (E) Horizontal Lines Detected (F) Sections Generated with Text

FIGURE 4.3: Image processing steps for OCR and layout analysis in *CORTEx*

4.4.1 Pipeline Overview

CORTEx was created to address several limitations in existing tools like `PSC2Code` [19] and `CodeMotion` [77], particularly when dealing with standalone code images rather than video screencasts. While `PSC2Code` and `CodeMotion` are good at processing code from video frames, they lack certain features crucial for analyzing individual images, which are common in platforms like Stack Overflow. *CORTEx's* development was driven by the need for a more comprehensive, layout-aware approach to text extraction and analysis from single and complex images as seen on Q&A platforms like Stack Overflow.

Our approach builds upon recent developments in deep learning and computer vision while addressing the specific requirements of programming content. By leveraging advanced neural network architectures and custom-designed image processing algorithms, *CORTEx* achieves superior accuracy in both text extraction and structural preservation. The system employs a multi-stage processing pipeline that includes intelligent

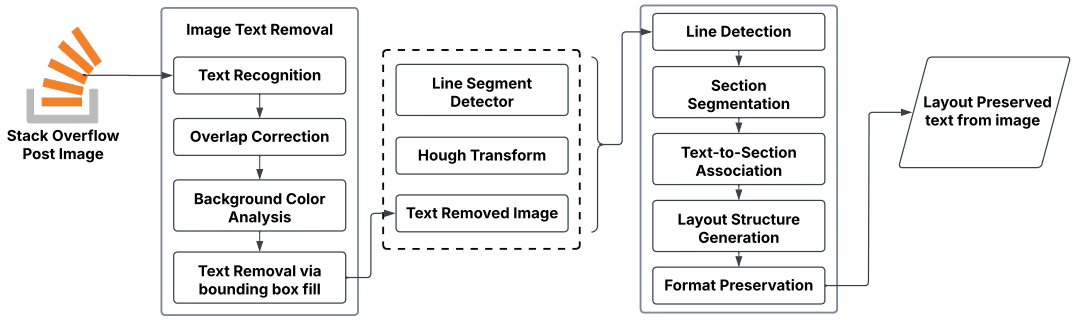


FIGURE 4.4: Overview of the processes used for extracting layout preserved text in *CORTEx*

region detection, context-aware text extraction, and structure-preserving formatting, ensuring that the extracted code maintains its original formatting and relationships.

The *CORTEx* methodology consists of five distinct phases, each designed to address specific challenges in code extraction from programming screenshots:

1. **Text Detection:** Using advanced OCR capabilities to identify and locate all textual elements within the image, providing precise bounding box coordinates for each detected text segment.

2. **Text Removal:** Systematically removing detected text from the image while preserving the underlying structure, creating a clean foundation for subsequent line detection and segmentation processes.

3. **Line Detection:** Employing sophisticated computer vision techniques to detect both vertical and horizontal structural lines that define the layout boundaries and organizational structure of the programming interface.

4. **Layout Structure Retrieval:** Analyzing the detected lines to construct a comprehensive understanding of the image's sectional organization, identifying distinct regions that correspond to different functional areas of the programming environment.

5. **Text-to-Section Association:** Intelligently mapping the originally detected text elements back to their corresponding sections while preserving spatial relationships, indentation patterns, and structural formatting.

This systematic approach ensures that *CORTEx* not only extracts text accurately but also maintains the critical contextual and structural information that is essential for code comprehension and debugging purposes. The methodology is specifically optimized for the unique characteristics of programming screenshots, addressing the limitations observed in general-purpose OCR systems and existing code extraction tools.

4.4.2 Image Segmentation and Region Detection

Our approach to processing Stack Overflow images leverages Google Vision's OCR library on Vertex AI, a choice driven by its superior accuracy and comprehensive feature set. This powerful API doesn't just recognize text; it provides a rich set of annotations that include full-text detection organized into paragraphs and words, each accompanied by precise bounding boxes. These bounding boxes are crucial as they allow us to maintain the spatial relationships between text elements, which is essential for preserving code structure and formatting.

The high accuracy of this text recognition forms the foundation of our subsequent processing steps. In the context of code snippets and technical discussions on Stack Overflow, even small errors in character recognition can lead to significant misinterpretations. Therefore, starting with a reliable OCR solution is paramount to the success of our entire pipeline.

Our layout structure retrieval begins with a sophisticated color analysis to determine the predominant background color for text removal. This step is crucial for isolating the text and structural elements from the background, especially in cases where the image might have a non-uniform or slightly textured background.

We identify the most common color C by minimizing the following equation:

$$\arg\min_{C} \sum_{i=1}^{n} \min(\|C - c_i\|_2, \tau) \tag{4.1}$$

where c_i are sampled colors and τ is a threshold. This approach allows us to handle slight variations in background color that might occur due to image compression or lighting inconsistencies.

The text removal process is a critical preparatory step for our structural analysis. It involves calculating and expanding bounding boxes around detected words, sampling colors from points around these boxes, determining the most common background color (accounting for slight RGB variations), and finally filling each expanded bounding box with this identified color. This process effectively "erases" the text from the image, leaving us with a clean slate for detecting structural elements like lines and sections.

For line detection, we employ a dual approach using both the Hough Transform [67] and the Line Segment Detector (LSD) [184]. This combination allows us to capture both long, continuous lines and shorter line segments, providing a comprehensive understanding of the image's structure.

The Hough Transform is particularly effective for detecting long, continuous lines after Gabor filtering [101] and Canny edge detection [35]. It excels at identifying global structures in the image, such as borders between major sections or long separators in tables. The Line Segment Detector, on the other hand, is adept at identifying shorter line segments. This is crucial for detecting elements like indentation lines in code blocks or separators between closely spaced elements.

4.4.3 Text Extraction and Layout Reconstruction

The extraction process begins by meticulously parsing the API annotations to obtain not just the text content, but also the precise bounding coordinates for each recognized element. This spatial information is critical for maintaining the original layout and structure of the code or text in the image.

We then enhance this extracted text through a series of sophisticated post-processing steps. First, we merge overlapping words based on their bounding box area. This step is crucial for handling cases where the OCR might have incorrectly split a single word into multiple parts, which is common with certain fonts or when dealing with variable names in code.

Next, we unify proximate paragraphs using a threshold-based approach. This helps in reconstructing logical blocks of text or code that might have been separated due to

formatting or image quality issues. The threshold is carefully tuned to avoid merging unrelated sections while still capturing intended groupings.

Perhaps one of the most innovative steps in our process is the word proximity analysis. Here, we merge nearby words if one is not found in an English dictionary. This step is particularly useful in reconstructing mathematical expressions and code snippets, where variable names, operators, and syntax elements might be misinterpreted as separate words by the OCR.

After these merging operations, we recalculate the bounding boxes to ensure accurate spatial representation. This step is essential as it updates our spatial understanding of the text elements after the various merging operations, maintaining the integrity of the layout for subsequent processing steps.

The merging process can be mathematically represented as an overlap function between two bounding boxes A and B:

$$\text{Overlap}(A, B) = \frac{\text{Area}(A \cap B)}{\min(\text{Area}(A), \text{Area}(B))} \tag{4.2}$$

This equation allows us to quantify the degree of overlap between text elements, providing a robust basis for deciding when to merge them. By using the minimum area in the denominator, we ensure that even a small word fully contained within a larger one will be correctly identified as overlapping.

We directly detect vertical lines and infer horizontal lines based on the vertical line endpoints. This process includes several steps: First, we sort the detected lines by x-coordinates to establish a clear left-to-right ordering. Then, by merging nearby vertical lines based on proximity thresholds, which helps in consolidating slightly broken or interrupted lines. Next, we filter out short vertical lines to reduce noise and focus on significant structural elements. Using this, we create horizontal lines at the top and bottom endpoints of each vertical line, which helps in defining the boundaries of sections. Lastly, we extend these horizontal lines to the nearest vertical lines or image edges, completing the structural grid of the image.

This comprehensive approach to line detection allows us to create a robust structural representation of the image, capturing both major divisions and finer details of the layout.

We then create sections by identifying line intersections, followed by filtering and merging for meaningful segmentation. This step transforms our understanding of the image from a collection of lines to a set of logical sections, each potentially containing a distinct part of the code or text.

4.4.4 Structure-Preserving Formatting

In this crucial phase, we assign text elements to sections based on their coordinates, effectively mapping the OCR results onto our structural understanding of the image. Within each section, we meticulously sort and format words to maintain the original layout, preserving the logical flow and structure of the code or text.

Our process involves several key steps, each designed to capture a different aspect of the text's layout:

1. **Word Sorting:** We sort words using a composite key that considers both the estimated line number and the x-coordinate. The line number is calculated by dividing the y-coordinate by the median line height, which allows us to handle variations in line spacing. This approach ensures that words are ordered correctly both vertically and horizontally within each section.

2. **Line Break Detection:** We insert a new line when the vertical gap between words exceeds 0.75 times the median line height and an extra one when it exceeds 2.5 times. This adaptive approach allows us to accurately detect line breaks even in documents with varying line spacing and represent larger gaps when present.

3. **Indentation Preservation:** We calculate the horizontal offset of each line's first word from the section's left edge. This step is crucial for preserving the structure of code snippets, where indentation often carries semantic meaning.

4. **Tab Insertion:** We estimate the average character width and define a tab as 15 times this width. We then convert the initial indentation to tabs, with a maximum of 8 tabs per line. When enabled, we also insert up to 4 relative tabs within lines based on horizontal distances between words. This approach allows us to maintain the visual structure of the code while providing a standardized representation.

This meticulous approach helps us maintain the structural integrity of code snippets and other formatted text, preserving not just the content but also the visual cues that aid in understanding the code's structure and flow.

To accommodate different use cases and preferences, we generate two different versions of the formatted text:

- A full version preserves all characters, which maintains the highest fidelity to the original text.
- An ASCII-only version to mitigate potential OCR errors, which can be useful for processing or analysis tools that expect pure ASCII input.

Our system produces a comprehensive set of outputs designed to facilitate further analysis, verification, and use of the extracted information. These outputs include:

1. Sectioned Images: These visually represent the system's understanding of the image structure, displaying text bounding boxes, detected lines, and section segmentation. These annotated images serve as a valuable tool for verifying the accuracy of our structural analysis and can be used for debugging or refining our algorithms.

2. Text files: We generate multiple text files, including both the raw OCR output and several formatted versions of the extracted text. The raw OCR output serves as a baseline, allowing users to see the unprocessed results of the text recognition. The formatted versions, including those with preserved layout and indentation, provide ready-to-use representations of the extracted code or text.

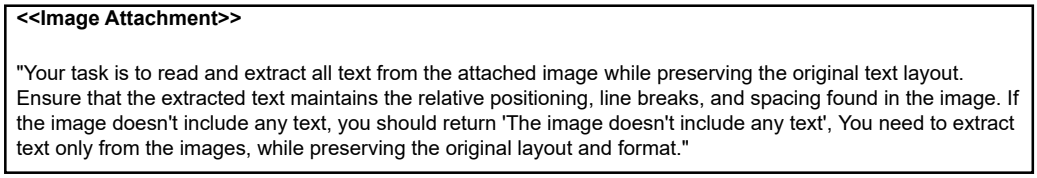


FIGURE 4.5: The Prompt Prefix. The «Image Attachment» part shows the image for which the text is being extracted.

These diverse outputs provide a rich set of data for further analysis and verification, allowing users to choose the most appropriate format for their specific needs, whether that's further processing, manual review, or direct use in development environments.

4.5 Evaluation Design

4.5.1 Research Questions

To evaluate the effectiveness of *CORTEx*, we conducted a comprehensive comparative analysis against state-of-the-art models and leading Large Language Models (LLMs). We replicated the text extraction methodologies employed by existing state-of-the-art approaches and developed a detailed prompt structure, as illustrated in Figure 4.5. This prompt was then utilized with both `GPT-4o` and `Gemini-1.5-pro` models. Our evaluation framework combines both qualitative and quantitative approaches to provide a comprehensive assessment of *CORTEx's* performance across multiple dimensions.

Specifically, we put forth the following Research Questions (RQs) for investigation:

Qualitative Research Questions

RQ1: To what extent do developers prefer layout or section segmentation generated by *CORTEx* versus state-of-the-art methods?

What and How: One of the main methodologies in *CORTEx* is segmenting images into various sections. To investigate how well *CORTEx* performs against other state-of-the-art models like `PSC2Code` and `CodeMotion`, we conducted a qualitative survey with 22 developers. We assessed the performance of *CORTEx* by providing these

developers with images processed by *CORTEx*, `PSC2Code`, and `CodeMotion` without specifying which model produced each image, ensuring an unbiased evaluation.

RQ2: How do developers perceive the format and structure of text extracted using *CORTEx* compared to the state-of-the-art methods and Large Language Models (LLMs)?

What and How: The format and structure of the text extracted from images through *CORTEx* requires evaluation, as Large Language Models (LLMs) and state-of-the-art methods like `PSC2Code` and `CodeMotion` can also extract text from images. However, the format and structure of the extracted text play a crucial role because they directly impact the usability and interpretability of the output. Well-structured text that maintains the original layout and organization of the image content can significantly enhance downstream tasks such as code understanding, documentation generation, and user interface analysis. Thus, we conducted a qualitative survey with the same developers to assess the performance of *CORTEx* against these models, focusing not only on the accuracy of text extraction but also on the preservation of the original format and structure.

Quantitative Research Question

RQ3: To what extent is *CORTEx* accurate in extracting text when compared to state-of-the-art methods and LLMs?

What and How: We wanted to look at how accurate is *CORTEx* in extracting correct text and code from the screenshots when compared with state-of-the-art models and LLMs. For this purpose, we manually extracted the code from all the images and then calculated BLEU, ROUGE-1, ROUGE-2, and ROUGE-L scores with all the models i.e. *CORTEx*, `PSC2Code`, `CodeMotion`, `Google Vision API`, `GPT-4o`, and `Gemini-1.5-pro`. BLEU and ROUGE scores tell us how good is the machine-generated text as compared to the original text and this is explained in more detail in Section 4.6.2. To assess the quality of extracted text across all approaches, we employed standard Natural Language Processing metrics—BLEU and ROUGE scores—comparing the outputs against a manually verified ground truth dataset. This evaluation framework

enabled us to quantitatively measure both the accuracy and contextual relevance of the text extracted by each method.

4.5.2 Dataset and Benchmarks

For our study, we sourced data from Stack Overflow by querying the Stack Exchange Data dump [165]. The data spans a 1-year period from June 1st, 2023, to June 31st, 2024. We established two primary criteria for including images in our dataset:

Criteria 1: Questions marked as closed: We focused on identifying questions within the Stack Overflow platform that had definitive resolutions. On the platform, when a question is resolved, it is marked as closed. We utilized this status to extract questions that had reached some form of conclusion, enabling us to highlight and evaluate our methodology effectively on substantive programming discussions.

Criteria 2: Questions with an image: Given our primary focus on the role of images in questions, we tailored our query to filter for questions with attached images in the body section of the posts. We implemented this by including a condition in our query to search for the `<img>` tag in the body section of each post when querying the Stack Exchange Data Dump [165]. This targeted selection strategy ensured we obtained only those posts containing relevant images.

By applying these criteria to our query of the *posts database* in the Stack Exchange Data Dump, we initially extracted 1,162 distinct questions from Stack Overflow. It's worth noting that some questions contained multiple images, resulting in a total image count exceeding the number of questions.

We then manually filtered these images to include only questions containing images that fall into the categories of "Code" or "Run Time Error" as described by Nayebi et al. [111]. Following this manual filtering process, we verified that the questions were still accessible, as some questions might have been removed or become inaccessible over time. After this comprehensive filtering process, our final dataset comprised 169 questions, each containing images of "Code" or "Run Time Error" and remaining accessible

on the platform. These 169 questions contained a total of 225 images, which serve as the foundation for our research.

Ground Truth Generation and Inter-rater Reliability

For the evaluation purposes and to enable comprehensive quantitative evaluation of our approach, we established a robust ground truth dataset through a careful manual extraction process. Two experienced developers, each with more than 4 years of professional software development experience, independently extracted text from these 225 images while maintaining the format and structure to the best of their ability. Both developers were familiar with various programming languages and integrated development environments, ensuring they could accurately interpret the content and context of the programming screenshots.

The manual extraction process involved preserving crucial elements such as indentation patterns, line breaks, spacing, special characters, and the spatial relationships between different text elements within each image. The developers were instructed to maintain the original structure as closely as possible, including line numbers when present, file paths, error messages, and any annotations or highlights that appeared in the screenshots.

To ensure inter-rater reliability and consistency in our ground truth dataset, the two developers worked independently on the same set of images and then convened to discuss any discrepancies in their extractions. Through a consensus-building process, they resolved differences by carefully examining the original images and agreeing on the most accurate representation of the text content and structure. This collaborative approach helped eliminate individual biases and ensured that our ground truth dataset accurately reflected the content present in the programming screenshots.

The resulting manually extracted text serves as our gold standard for comparing each model's performance, including *CORTEx*, state-of-the-art models like `PSC2Code` [19] and `CodeMotion` [77], and Large Language Models (LLMs) like `GPT-4o` and `Gemini-1.5-pro`.

This rigorous ground truth generation process ensures that our evaluation metrics provide meaningful and reliable assessments of each method's capabilities in accurately extracting and preserving the structure of code-related content from images.

It is important to emphasize that all questions in our dataset contain images that represent substantial programming content, enabling us to thoroughly explore the role of visual information on Q&A platforms like Stack Overflow. This unique characteristic of our dataset provides valuable insights into the interplay between visual and textual information in problem-solving contexts, which aligns with the ultimate goal of *CORTEx* to enhance developer support through improved image understanding and processing.

4.5.3 Evaluation Metrics and Baselines

Our evaluation framework employs a comprehensive set of metrics to assess both the qualitative user satisfaction and quantitative accuracy of *CORTEx* compared to existing approaches. We utilize developer surveys for subjective evaluation and established Natural Language Processing metrics for objective assessment.

Qualitative Evaluation Framework

For qualitative evaluation, we conducted comprehensive surveys with 22 developers to assess user preferences and perceptions. The survey participants were selected based on their experience with software development and familiarity with Q&A platforms like Stack Overflow. Table 4.1 presents the detailed demographics and usage patterns of our survey participants.

Among the 22 participants surveyed, the majority (42.9%) have 3-4 years of experience in the Software Development or Computer Science field. 23.8% have over 4 years of experience, while 14.3% have 1-2 years, 9.5% have 2-3 years, and 4.8% each have 0-1 year of experience or preferred not to answer. Regarding the use of Q&A platforms such as Stack Overflow, most participants (52.4%) use them often, while 33.3% use them sometimes. 9.5% always use these platforms, and only 4.8% never use them. When encountering questions with attached images, 57.1% of participants often rely on the

TABLE 4.1: Questions asked and Results of demographics from 22 developers

ID	Question	Response
Q1	How many years of experience do you have in the Software Development or Computer Science field?	0-1 year: 4.8% 1-2 years: 14.3% 2-3 years: 9.5% 3-4 years: 42.9% 4+ years: 23.8% Prefer not to answer: 4.8%
Q2	How often do you use Q/A platforms such as Stack Overflow for programming issues?	Never: 4.8% Rarely: 0.0% Sometimes: 33.3% Often: 52.4% Always: 9.5%
Q3	When you encounter a question with an attached image, how often do you rely on the image for the context of the problem?	Never: 4.8% Rarely: 4.8% Sometimes: 28.6% Often: 57.1% Always: 4.8%
Q4	How likely are you to use a feature that highlights relevant parts of images to make understanding and solving questions easier? (1-Not at all, 5-Definitely)	1: 0.0% 2: 4.8% 3: 9.5% 4: 47.6% 5: 38.1%
Q5	How often do you include annotations (e.g., arrows, colored boxes) in your screenshots to highlight specific areas of interest? (1-Never, 5-Always)	1: 4.8% 2: 38.1% 3: 9.5% 4: 23.8% 5: 23.8%

image for context, 28.6% sometimes rely on images, and 4.8% each always, rarely, or never rely on images for context.

The likelihood of using a feature that highlights relevant parts of images is generally high among the participants. The majority rated their likelihood as either 4 or 5 on a 1-5 scale, with 5 being the most common response. As for including annotations in screenshots, there's a varied distribution. The most common frequency is 2 on a 1-5 scale, followed closely by 4 and 5. This suggests that while some participants frequently include annotations, others do so less often.

The qualitative evaluation focuses on three key dimensions:

1. **Layout Creation and Section Segmentation:** Evaluating how well different methods segment images into meaningful sections that aid in code comprehension.

2. **Structure Preservation:** Assessing the ability of each method to maintain the original formatting, indentation, and spatial relationships present in the source images.

3. **Debugging and Information Retention:** Measuring how effectively the extracted text supports debugging tasks and retains critical information from the original screenshots.

Quantitative Metrics

For quantitative evaluation, we employ BLEU (Bilingual Evaluation Understudy) [140] and ROUGE (Recall-Oriented Understudy for Gisting Evaluation) [92] metrics, which are widely used for evaluating the quality of machine-generated text in various natural language processing tasks, including machine translation, text summarization, and in our case, code and text extraction from screenshots.

BLEU, originally designed for machine translation tasks, measures the precision of n-grams in the generated text compared to one or more reference texts. It scores from 0 to 1 (or 0 to 100 as a percentage), with higher scores indicating better performance. BLEU calculates the precision of n-grams in the machine-generated translation by comparing them to the reference translations and incorporates a brevity penalty to account for translations that are shorter than the reference translations [150].

The formula for BLEU score is as follows:

$$BLEU = BP \times \exp(\sum_{n=1}^{N} w_n \log p_n) \tag{4.3}$$

Where BP is the brevity penalty, w_n are weights (typically uniform), and p_n is the n-gram precision. The brevity penalty is calculated as:

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases} \tag{4.4}$$

Where c is the length of the candidate translation and r is the effective reference length.

ROUGE, on the other hand, was developed primarily for automatic summarization tasks. It focuses on recall and has several variants like ROUGE-1, ROUGE-2, and ROUGE-L, also scoring from 0 to 1. While BLEU emphasizes precision, ROUGE considers how much of the reference text is captured in the generated output. ROUGE-1, for instance, measures the overlap of 1-grams between the candidate text and the reference text [93].

The formula for ROUGE-N score is:

$$ROUGE\text{-}N = \frac{\sum_{S \in \{RefSummaries\}} \sum_{gram_n \in S} Count_{match}(gram_n)}{\sum_{S \in \{RefSummaries\}} \sum_{gram_n \in S} Count(gram_n)} \tag{4.5}$$

Where $Count_{match}(gram_n)$ is the maximum number of n-grams co-occurring in a candidate summary and a set of reference summaries, and $Count(gram_n)$ is the number of n-grams in the reference summaries.

Both BLEU and ROUGE have their strengths and limitations. BLEU is particularly effective for evaluating machine translation quality, as it focuses on the precision of word sequences. ROUGE, with its emphasis on recall, is more suitable for summarization tasks where capturing the main ideas of the reference text is crucial. However, both metrics rely heavily on exact word matches and may not fully capture semantic meaning or fluency. In practice, we use multiple metrics to get a comprehensive view of model performance, presenting average BLEU, ROUGE-1, ROUGE-2, and ROUGE-L scores for each model, providing a multi-faceted assessment of their performance across different metrics.

Baseline Methods

We compare *CORTEx* against several established baselines representing different approaches to text extraction from images:

`PSC2Code` **[19]:** A state-of-the-art method originally designed for extracting code from programming screencasts, adapted for static image analysis.

`CodeMotion` [77]: A computer vision algorithm that identifies and extracts source code regions from programming tutorial videos, applied to individual screenshots.

`Google Vision API`: A commercial OCR service that provides general-purpose text extraction capabilities with bounding box information.

`GPT-4o`: OpenAI's multimodal large language model capable of processing both text and images, representing the current state-of-the-art in general-purpose vision-language understanding.

`Gemini-1.5-pro`: Google's advanced multimodal language model that can analyze and extract information from images, providing another perspective on LLM-based approaches.

These baselines represent a comprehensive range of approaches, from specialized code extraction tools to general-purpose OCR systems and advanced multimodal language models. This diverse set of comparisons allows us to position *CORTEx* within the broader landscape of text extraction technologies and demonstrate its specific advantages for programming-related image analysis.

The evaluation methodology ensures fair comparison by applying consistent preprocessing and evaluation procedures across all methods, using the same dataset and ground truth references for quantitative metrics, and presenting outputs to survey participants in a randomized, anonymized manner to eliminate bias in qualitative assessments.

4.6 Results

In this section, we present a comprehensive analysis of our evaluation of *CORTEx* in comparison to state-of-the-art models and large language models for extracting and preserving code structure from images. Our evaluation addresses three key research questions: developer preference for layout and section segmentation (**RQ1**), developer perception of format and structure preservation (**RQ2**), and accuracy of text extraction

(**RQ3**). To assess *CORTEx's* performance, we employed a mixed-methods approach, combining qualitative surveys involving 22 developers with quantitative metrics including BLEU and ROUGE scores. This multifaceted evaluation strategy allows us to gauge both user perception and objective performance measures, providing a holistic view of *CORTEx's* capabilities.

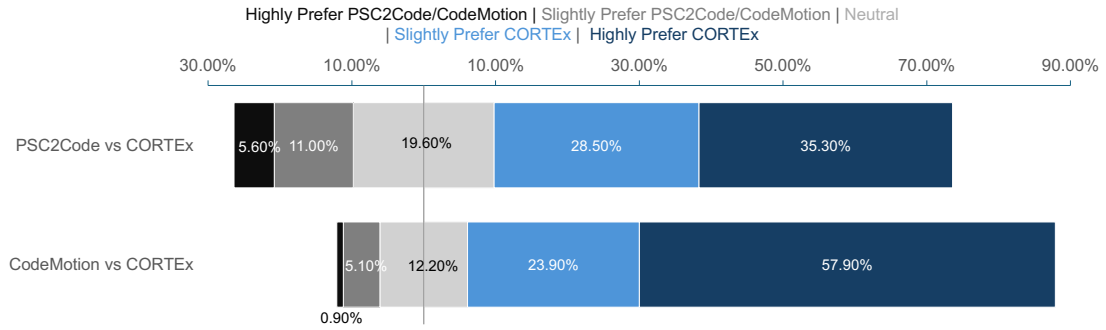
We benchmarked *CORTEx* against established models such as `CodeMotion`, `PSC2Code`, and `Google Vision API`, as well as advanced LLMs like `GPT-4o` and `Gemini-1.5-pro`. The following subsections detail our findings for each research question, offering insights into *CORTEx's* capabilities and its position within the current landscape of code extraction technologies. Our results demonstrate *CORTEx's* competitive edge in preserving code structure and readability, while also highlighting areas where it competes effectively with cutting-edge language models. Through this evaluation, we aim to provide a clear understanding of *CORTEx's* strengths and its potential impact on code extraction and representation tasks, paving the way for improved tools and methodologies in software development and maintenance.

4.6.1 Qualitative Results

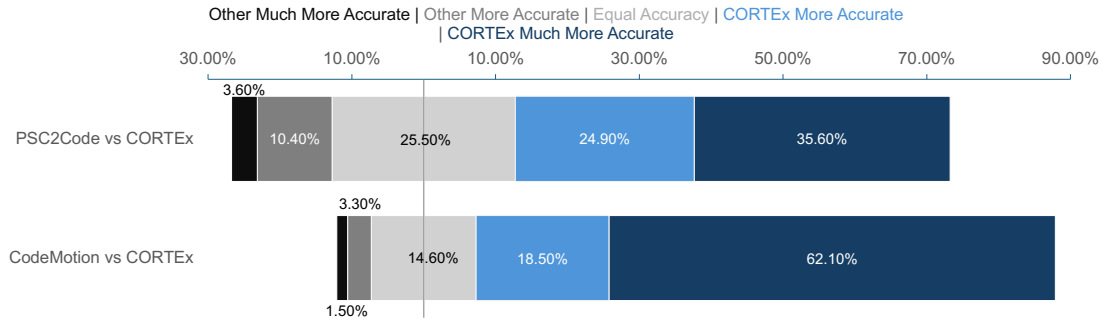
RQ1: Effectiveness of Layout Creation

We conducted a comprehensive survey to evaluate the quality of layout generation across different models, with a particular focus on comparing *CORTEx* to state-of-the-art solutions like `CodeMotion` and `PSC2Code`. The results of this survey reveal a strong preference among developers for the layouts generated by *CORTEx*. Figure 4.6 illustrates the comparison between *CORTEx* and these state-of-the-art models, demonstrating a consistent pattern of user preference across three key dimensions: layout creation, accurately defined sections, and easier understanding of layout for debugging purposes.

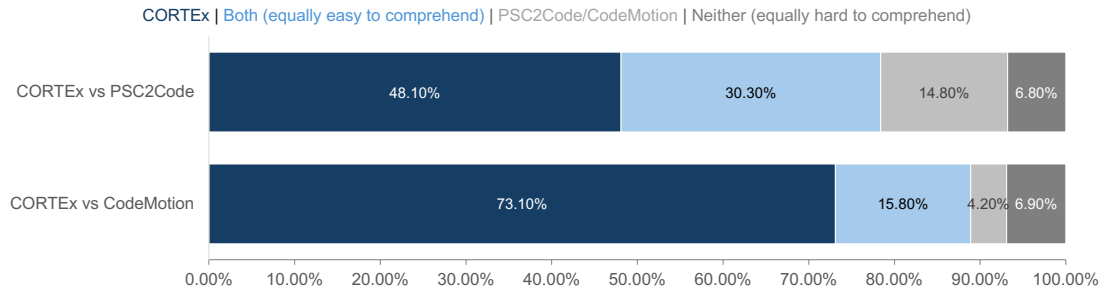
In terms of layout creation, *CORTEx* demonstrates clear superiority over its competitors. When compared to `CodeMotion`, an overwhelming 81.80% of users prefer *CORTEx*,



(A) Layout Creation preference of users for *CORTEx* vs PSC2Code/CodeMotion



(B) Accurately defined sections preference of users for *CORTEx* vs PSC2Code/CodeMotion



(C) Debugging preference of users for *CORTEx* vs PSC2Code/CodeMotion

FIGURE 4.6: Comparison of *CORTEx* vs `CodeMotion` and `PSC2Code` in terms of user preference, accuracy, and ease of comprehension.

with 57.90% expressing a strong preference and 23.90% a slight preference. In stark contrast, only 0.90% highly prefer `CodeMotion`, with 5.10% slightly favoring it. The remaining 12.20% maintain a neutral stance. The comparison with `PSC2Code`, while less pronounced, still heavily favors *CORTEx*: 63.80% of users prefer *CORTEx* (35.30% highly and 28.50% slightly), compared to only 16.60% preferring `PSC2Code` (5.60% highly and 11.00% slightly). Notably, a higher proportion (19.60%) remain neutral in this comparison, suggesting that while *CORTEx* leads, `PSC2Code` may offer some competitive features. These results underscore *CORTEx's* significant advantage in layout creation functionality and user satisfaction. The result is shown as a bar chart in Figure 4.6a.

Regarding accurately defined sections, *CORTEx* shows better results. In the comparison with `CodeMotion`, a substantial 80.60% of users view *CORTEx* as more accurate, with 62.10% considering it much more accurate and 18.50% seeing it as more accurate. `CodeMotion` falls far behind, with only 1.50% of users finding it much more accurate and 3.30% viewing it as more accurate. A small fraction (14.60%) of developers remained neutral. When compared to `PSC2Code`, *CORTEx* still leads convincingly, with 60.50% of users favoring its accuracy (35.60% much more accurate, 24.90% more accurate). `PSC2Code` fares slightly better than `CodeMotion` in this aspect, with 3.60% finding it much more accurate and 10.40% more accurate. However, a larger portion (25.50%) remained neutral, indicating that `PSC2Code` may have some strengths in this area. These results highlight *CORTEx's* perceived superior accuracy in defining sections, which is crucial for effective code organization and readability as evident from Figure 4.6b.

The debugging preference results further solidify *CORTEx's* advantage across different aspects of development. When compared to `CodeMotion`, an impressive 73.10% of users find *CORTEx* easier to comprehend and use for debugging tasks. This is in sharp contrast to the mere 6.90% who favor `CodeMotion` (4.20% finding it equally easy to comprehend, and 2.70% preferring it outright). The comparison with `PSC2Code`, while still favoring *CORTEx*, shows a more nuanced picture: 48.10% prefer *CORTEx* for debugging, while only 6.80% favor `PSC2Code`. Interestingly, a significant 30.30% found both *CORTEx* and `PSC2Code` equally easy to comprehend in this aspect. This suggests that

TABLE 4.2: Percentage of user responses on whether changes are needed to improve section creation in different methods, along with top suggestions for improvements (Responses are written combining similar text as it was an open-ended question)

Method	**Changes Needed**		**Top 4 Common Problems pointed out to improve layout**
	No	**Yes**	
CORTEx	*63.10%**	*36.90%**	1. Doesn't provide additional value by sectioning 2. Sectioning serves zero purpose in some areas of the image 3. Fewer sections in some parts of the image, like code area 4. Merge some sections together to avoid unnecessary sections
`PSC2Code`	40.36%	59.64%	1. Sectioning serves zero purpose in some areas of the image 2. Sections do not cover the entire image 3. Inaccurate Sections 4. Way too many sections in some parts of the image
`CodeMotion`	26.87%	73.13%	1. Way too many sections 2. Sections do not cover the entire image 3. Too many lines make it seem busy 4. Sections do not make sense at all

while *CORTEx* maintains a clear lead, `PSC2Code` offers competitive features for debugging that resonate with a substantial portion of users. The remaining 14.80% found neither tool particularly easy to comprehend for debugging, indicating room for improvement in both systems. Overall, these results underscore *CORTEx's* strong performance across layout creation, section accuracy, and debugging ease, positioning it as a preferred tool for a majority of users in various aspects of development work as shown in Figure 4.6c.

To further improve the *CORTEx* model, we asked developers for suggestions to improve each model without bias. As referenced in Table 4.2, the user feedback on section creation methods reveals varying levels of satisfaction and areas for improvement across *CORTEx*, `PSC2Code`, and `CodeMotion`. *CORTEx* emerges as the most positively received method, with 63.10% of users indicating that no changes are needed, while 36.90% suggest improvements. This contrasts sharply with `PSC2Code` and `CodeMotion`, where the majority of users (59.64% and 73.13% respectively) believe changes are necessary. These statistics underscore *CORTEx's* superior performance in meeting user expectations for section creation, although there is still room for enhancement.

The top suggestions for improvement provide insight into the specific challenges users face with each method. For *CORTEx*, the primary concerns revolve around the value and necessity of sectioning in certain areas, with users noting that sectioning sometimes "doesn't provide additional value" or "serves zero purpose in some areas of the image." `PSC2Code` users primarily struggle with coverage and accuracy issues, citing that "sections do not cover the entire image" and there are "inaccurate sections." `CodeMotion` faces the most critical feedback, with users reporting "way too many sections," "sections do not cover the entire image," and that "sections do not make sense at all." These findings highlight the need for balanced and purposeful sectioning which is somewhat achieved by *CORTEx* but there is still room for improvement which we leave for future work.

RQ2: Results of Format and Structure comparison

The comprehensive analysis of Structure Preservation Preference and various performance metrics across different models compared to *CORTEx* provides significant insights into the tool's positioning and strengths within the field of code understanding and manipulation. Figure 4.7 illustrates the Structure Preservation Preference by users for different models compared against *CORTEx*, revealing a clear advantage for *CORTEx* over most competitors. In comparisons with `CodeMotion` and `PSC2Code`, *CORTEx* demonstrates overwhelming user preference. An impressive 77.01% of users prefer *CORTEx* over `CodeMotion` (with 57.01% highly preferring and 20.00% slightly preferring it), while only 2.87% favor `CodeMotion`. Similarly, when compared to `PSC2Code`, 78.06% of users express a preference for *CORTEx* (60.97% highly preferring and 17.10% slightly preferring), with merely 6.77% leaning towards `PSC2Code`. These statistics emphatically underscore *CORTEx's* superior ability to maintain code structure, a crucial aspect for developers working on large and complex codebases. The stark contrast in user preference highlights *CORTEx's* effectiveness in preserving the structural integrity of code, which is vital for code readability, maintainability, and overall development efficiency.

Interestingly, when *CORTEx* is compared to more advanced language models like

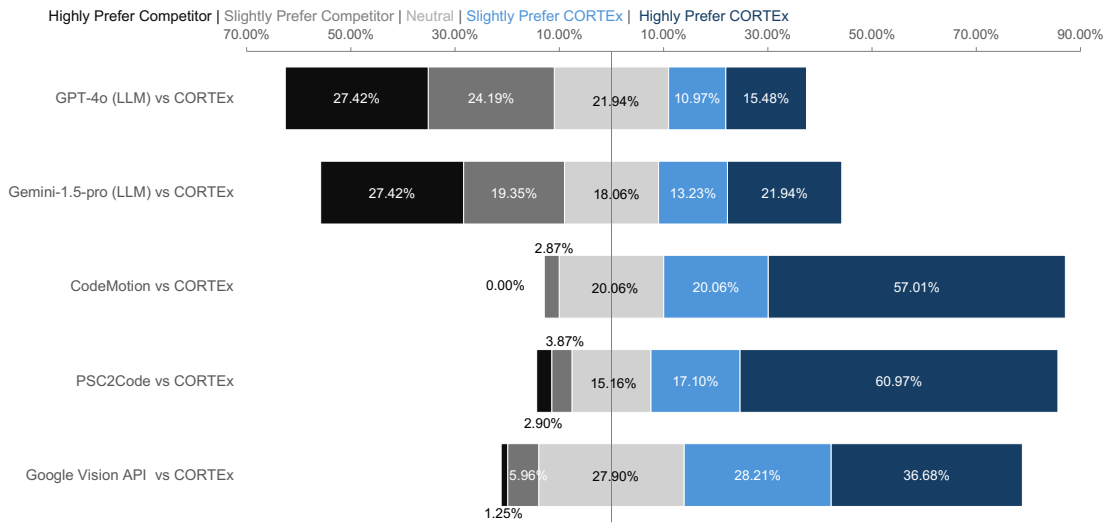


FIGURE 4.7: Structure Preservation Preference by users for different models compared against *CORTEx*

`GPT-4o` and `Gemini-1.5-pro`, the preference distribution becomes more balanced, though *CORTEx* still maintains a notable edge because sometimes LLM hallucinate and generate extra text or omit text which is not a problem with *CORTEx*. In the comparison with `GPT-4o`, 26.34% of users prefer *CORTEx* (15.48% highly and 10.87% slightly), while 51.61% favor `GPT-4o`. The results are similar when compared to `Gemini-1.5-pro`, where 34.84% of users prefer *CORTEx* (21.94% highly and 13.23% slightly), while 45.77% lean towards `Gemini-1.5-pro`. This more evenly distributed preference suggests that while large language models are strong contenders in the field of code analysis and manipulation, *CORTEx* holds its ground remarkably well in structure preservation. This competitive performance against advanced LLMs is likely attributable to *CORTEx's* specialized focus on code understanding and structure preservation. This is important as sometimes the LLMs are not very good at retaining the complete information from the image as they omit some parts of the image as shown in Figure 4.2 and also evident from Table 4.3. The comparison with `Google Vision API` presents an even more favorable case for *CORTEx*, with 64.89% of users preferring it (36.68% highly and 28.21% slightly) compared to only 6.77% favoring `Google Vision API`. This substantial difference might be attributed to *CORTEx's* tailored approach to code structure analysis compared to a more generalized vision API, highlighting the importance of domain-specific tools in

TABLE 4.3: Comparison of Metrics Across Various Models Against *CORTEx* based on the survey

Competitor Model	Metric	*CORTEx*	Competitor	Both	Neither
`Google Vision API`	Debugging	*54.55%**	5.64%	22.57%	17.24%
	Information Retention	40.75%	4.08%	*47.02%**	8.15%
	Readability	*62.38%**	5.33%	14.73%	17.55%
`PSC2Code`	Debugging	*80.00%**	4.52%	1.94%	13.55%
	Information Retention	*87.42%**	3.23%	4.19%	5.16%
	Readability	*78.71%**	3.87%	3.23%	14.19%
`CodeMotion`	Debugging	*75.16%**	4.14%	5.73%	14.97%
	Information Retention	*81.53%**	2.87%	8.92%	6.69%
	Readability	*75.48%**	2.55%	7.64%	14.33%
`Gemini-1.5-pro`	Debugging	*36.77%**	29.03%	24.19%	10.00%
	Information Retention	*50.32%**	10.97%	33.23%	5.48%
	Readability	31.61%	*37.10%**	26.13%	5.16%
`GPT-4o`	Debugging	30.00%	*35.16%**	27.74%	7.10%
	Information Retention	*47.42%**	13.87%	35.81%	2.90%
	Readability	22.58%	*43.55%**	30.00%	3.87%

specialized tasks like code structure preservation.

Table 4.3 provides a detailed comparison of metrics across various models against *CORTEx*, offering deeper insights into specific performance areas that contribute to overall user preference. In the critical area of debugging using the text extracted from various models, *CORTEx* consistently outperforms most competitors, with particularly high scores against `PSC2Code` (80.00%) and `CodeMotion` (75.16%). These impressive figures suggest that *CORTEx* provides superior tools and insights for identifying and resolving code issues, a crucial ability for efficient software development. However, it's worth noting that *CORTEx* faces stiffer competition from `GPT-4o` in this domain, where it slightly trails (30.00% vs. 35.16%). This close competition with an advanced LLM underscores the high level of performance *CORTEx* achieves in debugging tasks.

In terms of information retention, *CORTEx* shows exceptional performance, particularly against `PSC2Code` (87.42%) and `CodeMotion` (81.53%). It maintains this advantage even against advanced LLMs, scoring 47.42% against `GPT-4o's` 13.87%. These results highlight *CORTEx's* ability to effectively capture and preserve critical information within code structures, an essential feature for maintaining code integrity and facilitating long-term project maintenance.

Regarding readability, another crucial aspect of code quality, *CORTEx* again demonstrates strong performance against most competitors, notably scoring 78.71% against `PSC2Code` and 75.48% against `CodeMotion`. However, it faces more substantial competition from `GPT-4o` in this metric, with `GPT-4o` scoring slightly higher (43.55% vs. 22.58%). This suggests that while *CORTEx* excels in making code more readable compared to specialized code tools, advanced language models like `GPT-4o` may offer some advantages in this particular area, possibly due to their broader language understanding capabilities.

4.6.2 Quantitative Results

RQ3: Accuracy of different models in text extraction

To evaluate the accuracy of different models in text extraction from code images, we compared our *CORTEx* model against several state-of-the-art systems and commercial APIs. BLEU (Bilingual Evaluation Understudy) [140] and ROUGE (Recall-Oriented Understudy for Gisting Evaluation) [92] are two widely used metrics for evaluating the quality of machine-generated text in various natural language processing tasks, including machine translation, text summarization, and in this case, code and text extraction from screenshots.

BLEU, originally designed for machine translation tasks, measures the precision of n-grams in the generated text compared to one or more reference texts. It scores from 0 to 1 (or 0 to 100 as a percentage), with higher scores indicating better performance. BLEU calculates the precision of n-grams in the machine-generated translation by comparing them to the reference translations and incorporates a brevity penalty to account for translations that are shorter than the reference translations [150].

ROUGE, on the other hand, was developed primarily for automatic summarization tasks. It focuses on recall and has several variants like ROUGE-1, ROUGE-2, and ROUGE-L, also scoring from 0 to 1. While BLEU emphasizes precision, ROUGE considers how much of the reference text is captured in the generated output. ROUGE-1, for

TABLE 4.4: Average BLEU and ROUGE scores for different models

No.	Model	BLEU (%)	ROUGE-1 (%)	ROUGE-2 (%)	ROUGE-L (%)
1	`CodeMotion`	25.98	49.02	27.89	47.11
2	`PSC2Code`	12.57	28.65	14.12	27.05
3	*CORTEx* (Ours)	*61.08**	*78.29**	54.82	*74.33**
4	`Google Vision API`	47.43	62.73	37.07	59.43
5	`Gemini-1.5-pro`	45.25	62.50	49.21	62.03
6	`GPT-4o`	53.09	71.63	*57.40**	70.81

instance, measures the overlap of 1-grams between the candidate text and the reference text [93].

Both BLEU and ROUGE have their strengths and limitations. BLEU is particularly effective for evaluating machine translation quality, as it focuses on the precision of word sequences. ROUGE, with its emphasis on recall, is more suitable for summarization tasks where capturing the main ideas of the reference text is crucial. However, both metrics rely heavily on exact word matches and may not fully capture semantic meaning or fluency.

In practice, researchers often use multiple metrics to get a comprehensive view of model performance. For instance, evaluations might present average BLEU, ROUGE-1, ROUGE-2, and ROUGE-L scores for each model, providing a multi-faceted assessment of their performance across different metrics. This approach helps to balance out the limitations of individual metrics and provides a more robust evaluation of the generated text's quality.

Our *CORTEx* model demonstrates superior performance in most metrics, achieving the highest scores in BLEU (61.08%), ROUGE-1 (78.29%), and ROUGE-L (74.33%). These results indicate that *CORTEx* produces text that is both more similar to the reference text (as measured by BLEU) and captures a higher proportion of the reference content (as shown by ROUGE-1 and ROUGE-L). Notably, `GPT-4o` achieved the highest ROUGE-2 score (57.40%), suggesting it performs particularly well in capturing bi-gram overlaps, which often correlate with preserving code structure and syntax.

Among the commercial solutions, `Google Vision API` and `Gemini-1.5-pro` show

competitive performance, with scores generally higher than `CodeMotion` and `PSC2Code` but lower than *CORTEx* and `GPT-4o`. This highlights the effectiveness of large-scale, general-purpose models in handling specialized tasks like code extraction from images. However, the significant performance gap between these general models and our specialized *CORTEx* model underscores the value of task-specific training and architectures in achieving state-of-the-art results for text extraction tasks.

The results demonstrate that *CORTEx* significantly outperforms existing methods and state-of-the-art LLMs in both quantitative measures, with an average improvement of around 15% in BLEU and ROUGE scores compared to current approaches, indicating superior accuracy in code extraction. This improvement demonstrates *CORTEx's* superior ability to accurately extract and interpret information from code-related images, validating our approach's effectiveness in addressing the specific challenges of programming screenshot analysis.

4.7 Discussion

Based on the comprehensive evaluation presented in this study, *CORTEx* demonstrates significant advancements in code extraction and layout preservation from images, particularly for Q&A platforms like Stack Overflow. The results consistently show *CORTEx's* superior performance across multiple dimensions when compared to state-of-the-art methods and advanced LLMs.

4.7.1 Implications for Developer Support Tools

The findings reveal important implications for future developer support tools and Q&A platform enhancement. *CORTEx's* superior performance in layout creation and section segmentation addresses fundamental challenges in visual communication within software development communities. The overwhelming developer preference for *CORTEx* (over 70% preference rates across various metrics) suggests significant value in specialized tools that understand programming content characteristics.

With image-based communication in programming discussions more than doubling between 2013 and 2022 [111], *CORTEx's* ability to accurately process and structure visual communications can significantly enhance knowledge transfer efficiency and problem-solving within developer communities. The quantitative results showing 15% improvement in BLEU and ROUGE scores translate to tangible benefits: fewer errors in automated processes, reduced manual correction efforts, and improved reliability in downstream applications.

CORTEx's competitive performance against advanced LLMs demonstrates that specialized tools can offer unique advantages over general-purpose models. While LLMs excel in language understanding, *CORTEx's* focus on structure preservation and comprehensive information retention makes it particularly suitable for applications where fidelity to original content is critical.

4.7.2 Strengths and Limitations of CORTEx

CORTEx exhibits several key strengths that distinguish it from existing approaches. First, its comprehensive approach to image analysis ensures complete coverage of complex screenshots, minimizing the risk of missing crucial information. Unlike `PSC2Code` and `CodeMotion`, which sometimes overlook certain areas, *CORTEx* implements thorough scanning and segmentation processes particularly valuable for IDE screenshots containing multiple regions of interest.

Second, *CORTEx's* sophisticated structure preservation capabilities maintain indentation patterns, spatial relationships, and visual hierarchies crucial for code comprehension. The methodology's five-phase approach (text detection, removal, line detection, layout retrieval, and text-to-section association) ensures both accurate extraction and structural integrity maintenance.

Third, the context-aware processing pipeline addresses specific challenges of programming screenshots, incorporating layout analysis and structural understanding to maintain semantic integrity. This specialized focus yields superior performance in debugging, information retention, and readability compared to general-purpose solutions.

However, *CORTEx* also has limitations that warrant consideration. User feedback indicates that sectioning sometimes doesn't provide additional value in certain image areas, with 36.90% of users suggesting improvements. The system occasionally creates unnecessary sections or fails to merge related sections appropriately.

Additionally, *CORTEx's* reliance on Google Vision API for initial OCR introduces dependency on external services, and the current implementation focuses specifically on Stack Overflow-style programming screenshots, potentially limiting generalizability to other visual programming contexts.

4.7.3 Future Directions for Visual Q&A

The success of *CORTEx* opens several promising research directions for visual Q&A systems and developer support tools. Combining *CORTEx* with LLM capabilities represents a particularly promising direction that could leverage the strengths of both approaches. While *CORTEx* excels in structure preservation and comprehensive information extraction, LLMs offer superior language understanding and generation capabilities. A hybrid approach could maintain structural fidelity while improving readability and semantic understanding, potentially addressing the readability limitations observed in our evaluation where `GPT-4o` outperformed *CORTEx* in readability metrics.

Integration with existing Q&A platforms could provide real-time visual analysis capabilities, automatically highlighting relevant code sections and generating structured summaries of visual content to enhance question clarity and answer relevance. This integration would transform how developers interact with visual programming content on platforms like Stack Overflow, making it more accessible and actionable for both question askers and answerers, ultimately improving the efficiency of knowledge transfer within developer communities.

Expanding beyond static images to handle dynamic content like GIFs, video tutorials, and interactive code demonstrations represents a natural evolution of this research. This could bridge the gap between *CORTEx's* static image processing capabilities and the temporal analysis features of tools like `CodeMotion`, creating more comprehensive

solutions for multimedia programming content that reflects the evolving nature of developer communication and learning resources.

4.8 Threats to Validity

Like any empirical study, our evaluation of *CORTEx* has limitations that need to be considered. These limitations can affect the applicability and generalizability of our findings.

External Validity: The generalizability of our findings depends on the representativeness of our chosen sample. Our dataset of 225 images from 169 Stack Overflow questions, while carefully curated, may not fully represent the diversity of code-related images across all Q&A platforms or programming contexts. The types of questions we analyzed and the specific time period we covered (June 2023 to June 2024) could influence how well our conclusions apply to other situations or time frames. Additionally, online coding communities and their practices are constantly evolving, which could impact the long-term relevance of our results.

Internal Validity: Threats to internal validity can influence the cause-and-effect relationships observed in our study. A primary concern is measurement bias arising from the performance of our OCR and image processing technologies. The quality and variability of the images within our dataset can affect the accuracy of text extraction and layout analysis. The specific implementations we used for OCR (Google Vision API) and our custom image processing techniques could introduce bias in our measurements. Furthermore, the subjective nature of evaluating layout quality and code structure preservation in our developer surveys could introduce annotation bias.

Construct Validity: Construct validity refers to the degree to which our study truly captures the concepts we intended to measure. A potential threat is the difficulty in fully replicating our results due to the complexity of our *CORTEx* system. Variations in hardware configurations, computational resources, and even slight differences in implementation could hinder the reproducibility of our experiments. We have attempted

to mitigate this by providing detailed descriptions of our methodology and making our code available, but replication challenges may still exist.

Conclusion Validity: Our conclusions about *CORTEx's* performance relative to other models and LLMs are based on specific metrics (BLEU, ROUGE scores) and qualitative developer feedback. While these measures provide valuable insights, they may not capture all aspects of code extraction and layout preservation quality. The statistical significance of our results, particularly in comparisons with state-of-the-art models, should be carefully considered. Additionally, the practical significance of the improvements offered by *CORTEx*, especially in terms of computational resources required versus performance gains, warrants careful interpretation.

To address these threats, future work could involve expanding the dataset to include a wider range of programming languages, image types, and Q&A platforms. Additionally, conducting more extensive user studies with a larger and more diverse group of developers could further validate our findings on layout preference and code structure preservation.

4.9 Conclusion

This study introduced *CORTEx*, a novel method for context-aware recognition and extraction of text and code from images posted on Q&A platforms. Through comprehensive evaluation comparing *CORTEx* against state-of-the-art methods and advanced LLMs, we demonstrated significant improvements in code extraction accuracy and layout preservation.

CORTEx consistently outperformed existing approaches across multiple dimensions. Qualitatively, developers showed strong preference for *CORTEx's* layout creation and section segmentation, with over 70% preferring it over `PSC2Code` and `CodeMotion`. Quantitatively, *CORTEx* achieved superior performance in most metrics, with 15% average improvement in BLEU and ROUGE scores, demonstrating better text extraction accuracy and content preservation.

While *CORTEx* faced competition from advanced LLMs like `GPT-4o` in some aspects, it maintained competitive performance and often excelled in critical areas such as information retention and structure preservation. This achievement is particularly noteworthy given the specialized nature of *CORTEx* compared to general-purpose LLMs, highlighting the value of domain-specific solutions for programming-related tasks.

The study addresses the growing trend of image-based communication in software development, where visual content has more than doubled on platforms like Stack Overflow. *CORTEx's* ability to accurately extract and preserve code structure from images enhances the accessibility and utility of visual information in coding contexts, potentially improving problem-solving efficiency and knowledge sharing within developer communities.

CORTEx represents a significant step forward in bridging the gap between visual and textual information in software development contexts. As image-based communication continues to grow, tools like *CORTEx* are positioned to play a crucial role in improving developer support and enhancing the effectiveness of Q&A platforms for programming-related discussions.

4.10 Replication Package

For the reproducibility of this study and open science, we provide the community with all the artifacts used in our study. In particular, we provided the source code of our model with documentation and our complete dataset. The project including all artifacts is available at this gtihub link[⋆1] [52].

[1] ⋆ All artifacts related to this study are available at `https://github.com/Research-Purpose/CORTEx`

Chapter 5

Conclusion and Future Work

The integration of visual elements into programming The integration of visual elements into programming practices represents a shift in how developers communicate technical problems and solutions. Screenshots of code and development environments have become increasingly common on platforms like Stack Overflow, with studies showing a doubling of image-inclusive posts between 2013 and 2022 [111, 112, 113]. This creates a tension in technical knowledge sharing: developers prefer visual communication for its immediacy and context, yet these screenshots are largely overlooked in automated systems. This research explores how image processing techniques and emerging AI technologies can bridge the gap between visual inputs and textual representations, potentially transforming how developers seek and receive assistance while preserving the convenience of sharing images [113].

5.1 Conclusion

This manuscript-based thesis has explored how images can help the developer community and users on Q&A platforms like Stack Overflow through three interconnected studies aligned with our research objectives. We investigated whether images can enhance automated duplicate post detection on Stack Overflow, evaluated the effectiveness of programming screenshots for question inference and highlighting problematic code segments in screenshots, and Developed CORTEx as a novel approach to extract structured text from programming screenshots

In Chapter 3, we extended our research from our previous work [6] to evaluate how effectively multimodal LLMs identify and extract relevant content from screenshots. `Gemini` achieved an F1 score of 0.91 for relevance detection, while `GPT-4o` excelled at text extraction with BLEU scores of 0.46 and ROUGE-1 scores of 0.68 when assisted by OCR.

In Chapter 4, we developed *CORTEx*, a method specifically designed to extract structured text from programming screenshots. CORTEx outperformed existing tools like PSC2Code and CodeMotion by 15.00% on BLEU and ROUGE scores, with 70.00% of developers preferring its layout preservation capabilities.

These contributions collectively advance our understanding of how visual content can be effectively leveraged in programming contexts. By enabling more accurate extraction and interpretation of code from screenshots, our work helps bridge the gap between developers' preference for visual communication and platforms' requirements for searchable, accessible text content. As programming continues to evolve as a visually rich discipline, the methods and insights presented in this thesis provide a foundation for more intuitive and effective knowledge sharing in software engineering communities.

5.2 Future Work

The findings of this thesis open several exciting avenues for future research in visual programming assistance. One particularly promising direction is developing complete image-to-solution pipelines, where developers could share screenshots and receive tailored code fixes or explanations without writing any text and even completely formulated Stack Overflow post with only the image. This could dramatically lower the barrier to seeking help, especially for novice programmers who struggle to articulate their problems. Additionally, investigating how images might reduce bias in other software engineering automation tasks—beyond duplicate detection—could reveal new applications where visual context provides crucial information that text alone misses. For instance, in accessibility testing, many issues like poor color contrast or tiny buttons are

obvious when you see them but hard to describe in text. Visual analysis could catch these problems that would otherwise slip through automated checks.

Visual content opens up more exciting possibilities for how developers could interact with programming tools in the future. Imagine being able to upload a screenshot of a confusing error and getting back not just a text explanation, but also a visual walkthrough showing exactly where the problem is and how to fix it. Future systems could automatically generate step-by-step visual guides that walk developers through solutions. This could be especially valuable for beginners who often know something's wrong but can't quite put their finger on what it is. Additionally, these tools could adapt to different skill levels—showing more detailed explanations for newcomers while providing concise visual cues for experienced developers. The goal would be making programming help as intuitive as pointing at something and saying "this doesn't work" rather than trying to describe complex technical problems in words.

Bibliography

[1] Durham Abric et al. "Can duplicate questions on stack overflow benefit the software development community?" In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, pp. 230–234.

[2] Somak Aditya et al. "Image Understanding using vision and reasoning through Scene Description Graph". In: *Comput. Vis. Image Underst.* 173 (2018), pp. 33–45. DOI: 10.1016/j.cviu.2017.12.004.

[3] Vishakha Agrawal, Yong-Han Lin, and Jinghui Cheng. "Understanding the characteristics of visual contents in open source issue discussions: a case study of jupyter notebook". In: *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering*. 2022, pp. 249–254.

[4] Muhammad Ahasanuzzaman et al. "Mining Duplicate Questions in Stack Overflow". In: *Proceedings of the 13th International Conference on Mining Software Repositories*. MSR '16. Austin, Texas: Association for Computing Machinery, 2016, 402–412. ISBN: 9781450341868. DOI: 10.1145/2901739.2901770. URL: https://doi.org/10.1145/2901739.2901770.

[5] Faiz Ahmed, Suprakash Datta, and Maleknaz Nayebi. "Negative Results of Image Processing for Identifying Duplicate Questions on Stack Overflow". In: *arXiv preprint arXiv:2407.05523* (2024).

[6] Faiz Ahmed et al. "Inferring questions from programming screenshots". In: *arXiv preprint arXiv:2504.18912* (2025).

[7] Jihyun Janice Ahn et al. “Direct-Inverse Prompting: Analyzing LLMs’ Discriminative Capacity in Self-Improving Generation”. In: *ArXiv* abs/2407.11017 (2024). DOI: 10.48550/arXiv.2407.11017.

[8] Mohammad Alahmadi et al. “Accurately predicting the location of code fragments in programming video tutorials using deep learning”. In: *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*. 2018, pp. 2–11.

[9] Mohammad Alahmadi et al. “Code localization in programming screencasts”. In: *Empirical Software Engineering* 25 (2020), pp. 1536–1572.

[10] Mohammad D. Alahmadi and Moayad Alshangiti. “Optimizing OCR Performance for Programming Videos: The Role of Image Super-Resolution and Large Language Models”. In: *Mathematics* (2024). DOI: 10.3390/math12071036.

[11] Anahita Alipour, Abram Hindle, and Eleni Stroulia. “A contextual approach towards more accurate duplicate bug report detection”. In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE. 2013, pp. 183–192.

[12] Leif Andersen, Mike Ballantyne, and M. Felleisen. “Adding interactive visual syntax to textual code”. In: *Proceedings of the ACM on Programming Languages* 4 (2020), pp. 1 –28. DOI: 10.1145/3428290.

[13] apaul. *Discourage screenshots of code and/or errors*. Accessed: 2024-11-05. 2015. URL: https://meta.stackoverflow.com/questions/303812/discourage-screenshots-of-code-and-or-errors.

[14] Streamlit App. *Question Inferring Tool*. https://question-inferring.streamlit.app/. Accessed: 2024-11-05. 2024.

[15] Srikar Appalaraju et al. “Docformer: End-to-end transformer for document understanding”. In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2021, pp. 993–1003.

[16] Berk Atil et al. “LLM Stability: A detailed analysis with some surprises”. In: *ArXiv* abs/2408.04667 (2024). DOI: 10.48550/arXiv.2408.04667.

[17] A. R. Bagirzade et al. “OCR/ICR Text Recognition Using ABBYY FineReader as an Example Text”. In: (2021).

[18] Lingfeng Bao et al. “Enhancing developer interactions with programming screencasts through accurate code extraction”. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2020). DOI: 10.1145/3368089.3417925.

[19] Lingfeng Bao et al. “psc2code: Denoising code extraction from programming screencasts”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29.3 (2020), pp. 1–38.

[20] Ojas Kumar Barawal and Dr. Yojna Arora. “Text Extraction from Image”. In: *International Journal of Innovative Research in Engineering & Management* (2022). DOI: 10.55524/ijirem.2022.9.3.12.

[21] batFINGER. *Policy on posting code / error message etc as images*. Accessed: 2024-11-05. 2021. URL: https://blender.meta.stackexchange.com/questions/2788/policy-on-posting-code-error-message-etc-as-images.

[22] Brett A. Becker et al. “Effective compiler error message enhancement for novice programming students”. In: *Computer Science Education* 26 (2016), pp. 148 –175. DOI: 10.1080/08993408.2016.1225464.

[23] Tony Beltramelli. “pix2code: Generating code from a graphical user interface screenshot”. In: *Proceedings of the ACM SIGCHI symposium on engineering interactive computing systems*. 2018, pp. 1–6.

[24] Adrienne Bergh et al. “A curated set of labeled code tutorial images for deep learning”. In: *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE. 2020, pp. 1276–1280.

[25] Natalie Best, Jordan Ott, and Erik J Linstead. “Exploring the efficacy of transfer learning in mining image-based software artifacts”. In: *Journal of Big Data* 7 (2020), pp. 1–10.

[26] Nicolas Bettenburg et al. “Duplicate bug reports considered harmful... really?” In: *2008 IEEE International Conference on Software Maintenance*. IEEE. 2008, pp. 337–345.

[27] Meghana Moorthy Bhat et al. “Investigating Answerability of LLMs for Long-Form Question Answering”. In: *arXiv preprint arXiv:2309.08210* (2023).

[28] Tingting Bi et al. “Accessibility in software practice: A practitioner’s perspective”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31.4 (2022), pp. 1–26.

[29] Zhen Bi et al. “Codekgc: Code language model for generative knowledge graph construction”. In: *ACM Transactions on Asian and Low-Resource Language Information Processing* 23.3 (2024), pp. 1–16.

[30] Neda Hajiakhoond Bidoki et al. “Modeling social coding dynamics with sampled historical data”. In: *Online Social Networks and Media* 16 (2020), p. 100070.

[31] Emil Borjesson and Robert Feldt. “Automated system testing using visual gui testing tools: A comparative study in industry”. In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE. 2012, pp. 350–359.

[32] Tom B. Brown et al. “Language Models are Few-Shot Learners”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 1877–1901. URL: https://arxiv.org/abs/2005.14165.

[33] Yuzhe Cai et al. *Low-code LLM: Graphical User Interface over Large Language Models*. 2024. arXiv: 2304.08103 [cs.CL]. URL: https://arxiv.org/abs/2304.08103.

[34] Fabio Calefato, Filippo Lanubile, and Nicole Novielli. “How to ask for technical help? Evidence-based guidelines for writing questions on Stack Overflow”. In: *Information and software technology* 94 (2018), pp. 186–207.

[35] John Canny. “A Computational Approach to Edge Detection”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8.6 (1986), pp. 679–698. DOI: 10.1109/TPAMI.1986.4767851.

[36] Tsung-Hsiang Chang, Tom Yeh, and Robert C Miller. "GUI testing using computer vision". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2010, pp. 1535–1544.

[37] A. Chaudhuri et al. "Optical Character Recognition Systems". In: (2017), pp. 9–41. DOI: 10.1007/978-3-319-50252-6_2.

[38] Zheyi Chen et al. "Evolution and Prospects of Foundation Models: From Large Language Models to Large Multimodal Models." In: *Computers, Materials & Continua* 80.2 (2024).

[39] Jaemin Cho et al. "X-lxmert: Paint, caption and answer questions with multimodal transformers". In: *arXiv preprint arXiv:2009.11278* (2020).

[40] Noptanit Chotisarn et al. "A systematic literature review of modern software visualization". In: *Journal of Visualization* 23 (2020), pp. 539 –558. DOI: 10.1007/s12650-020-00647-w.

[41] Nathan Cooper et al. "It takes two to tango: Combining visual and textual information for detecting duplicate video-based bug reports". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 957–969.

[42] Shivani Dhiman and A Singh. "Tesseract vs gocr a comparative study". In: *International Journal of Recent Technology and Engineering* 2.4 (2013), p. 80.

[43] Gabriele Di Rosa, Andrea Mocci, and Marco D'Ambros. "Visualizing interaction data inside & outside the ide to characterize developer productivity". In: *2020 Working Conference on Software Visualization (VISSOFT)*. IEEE. 2020, pp. 38–48.

[44] Qingxiu Dong et al. "A survey on in-context learning". In: *arXiv preprint arXiv:2301.00234* (2022).

[45] Mathias Ellmann. "Same-same but different: on understanding duplicates in stack overflow". In: *Informatik Spektrum* 42.4 (2019), pp. 266–286.

[46] Daniele Fanelli. "Negative results are disappearing from most disciplines and countries". In: *Scientometrics* 90.3 (2012), pp. 891–904.

[47] Sidong Feng and Chunyang Chen. "Gifdroid: Automated replay of visual bug reports for android apps". In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 1045–1057.

[48] Lennart Fiebig et al. "Effective GUI Generation: Leveraging Large Language Models for Automated GUI Prototyping". In: (2025).

[49] Yasuhisa Fujii. "Optical character recognition research at google". In: *2018 IEEE 7th Global Conference on Consumer Electronics (GCCE)*. IEEE. 2018, pp. 265–266.

[50] James Gallagher and Piotr Skalski. "GPT-4 with Vision: Complete Guide and Evaluation". In: (2023). URL: https://blog.roboflow.com/gpt-4-vision/.

[51] M Geetha et al. "Implementation of text recognition and text extraction on formatted bills using deep learning". In: *Int J Contrl Automat* 13.2 (2020), pp. 646–651.

[52] Github. *Github Repository*. https://github.com/Research-Purpose/CORTEx. Accessed: 2025-02-05. 2024.

[53] Github. *Github Repository*. https://github.com/Research-Purpose/image-citation/tree/main. Accessed: 2024-11-05. 2024.

[54] Cody Gray. *Why should I not upload images of code/data/errors?* Accessed: 2024-11-05. 2015. URL: https://meta.stackoverflow.com/questions/285551/why-should-i-not-upload-images-of-code-data-errors/285557#285557.

[55] Jindong Gu et al. "A systematic survey of prompt engineering on vision-language foundation models". In: *arXiv preprint arXiv:2307.12980* (2023).

[56] Jiuxiang Gu et al. "Unidoc: Unified pretraining framework for document understanding". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 39–50.

[57] Y. Hasan and Lina Karam. "Morphological text extraction from images". In: *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society* 9 11 (2000), pp. 1978–83. DOI: 10.1109/83.877220.

[58] Yalda Hashemi, Maleknaz Nayebi, and Giuliano Antoniol. "Documentation of machine learning software". In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2020, pp. 666–667.

[59] Ahmed E Hassan et al. "Rethinking Software Engineering in the Foundation Model Era: From Task-Driven AI Copilots to Goal-Driven AI Pair Programmers". In: *arXiv preprint arXiv:2404.10225* (2024).

[60] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[61] Truong Ho-Quang et al. "Automatic Classification of UML Class Diagrams from Images". In: *2014 21st Asia-Pacific Software Engineering Conference*. Vol. 1. 2014, pp. 399–406. DOI: 10.1109/APSEC.2014.65.

[62] Hootsuite. *53 Social Media Statistics to Inform Your 2024 Social Strategy*. Accessed: 2024-11-05. 2024. URL: https://blog.hootsuite.com/social-media-statistics-for-social-media-managers/.

[63] Yupan Huang et al. "Layoutlmv3: Pre-training for document ai with unified text and image masking". In: *Proceedings of the 30th ACM International Conference on Multimedia*. 2022, pp. 4083–4091.

[64] Hugging Face. *model card*. https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2. Taken from the official website of Hugging Face. 2024.

[65] Aaron Hurst et al. "Gpt-4o system card". In: *arXiv preprint arXiv:2410.21276* (2024).

[66] Maeve Hutchinson et al. "LLM-Assisted Visual Analytics: Opportunities and Challenges". In: *arXiv preprint arXiv:2409.02691* (2024).

[67] John Illingworth and Josef Kittler. "A survey of the Hough transform". In: *Computer vision, graphics, and image processing* 44.1 (1988), pp. 87–116.

[68] Industrywired. *Instagram vs TikTok: Which Platform is Dominating in 2025*. https://industrywired.com/social-media-platform/instagram-vs-tiktok-which-platform-is-dominating-in-2025-8707912. Accessed: May 12, 2025. 2025.

[69] Md Farhan Ishmam et al. “From image to language: A critical analysis of visual question answering (vqa) approaches, challenges, and opportunities”. In: *Information Fusion* (2024), p. 102270.

[70] Noman Islam, Zeeshan Islam, and Nazia Noor. “A Survey on Optical Character Recognition System”. In: *ArXiv* abs/1710.05703 (2017).

[71] Hamed Jelodar et al. “Latent Dirichlet allocation (LDA) and topic modeling: models, applications, a survey”. In: *Multimedia Tools and Applications* 78 (2019), pp. 15169–15211.

[72] Chao Jia et al. “Scaling up visual and vision-language representation learning with noisy text supervision”. In: *International conference on machine learning*. PMLR. 2021, pp. 4904–4916.

[73] Shaikh Jeeshan Kabeer et al. “Predicting the vector impact of change-an industrial case study at brightsquid”. In: *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE. 2017, pp. 131–140.

[74] Samia Kabir et al. “Is stack overflow obsolete? an empirical study of the characteristics of chatgpt answers to stack overflow questions”. In: *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. 2024, pp. 1–17.

[75] Ehsan Kamalloo et al. “Evaluating open-domain question answering in the era of large language models”. In: *arXiv preprint arXiv:2305.06984* (2023).

[76] Mik Kersten and Gail C Murphy. “Using task context to improve programmer productivity”. In: *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. 2006, pp. 1–11.

[77] Kandarp Khandwala and Philip J Guo. “Codemotion: expanding the design space of learner interactions with computer programming tutorial videos”. In: *Proceedings of the Fifth Annual ACM Conference on Learning at Scale*. 2018, pp. 1–10.

[78] Geewook Kim et al. “Ocr-free document understanding transformer”. In: *European Conference on Computer Vision*. Springer. 2022, pp. 498–517.

[79] Barbara A Kitchenham et al. “Preliminary guidelines for empirical research in software engineering”. In: *IEEE Transactions on software engineering* 28.8 (2002), pp. 721–734.

[80] Elife Ozturk Kiyak et al. “Comparison of image-based and text-based source code classification using deep learning”. In: *SN Computer Science* 1.5 (2020), p. 266.

[81] Umme Ayman Koana et al. “Examining ownership models in software teams”. In: *Empirical Software Engineering* 29.6 (2024), pp. 1–43.

[82] Umme Ayman Koana et al. “Ownership in the hands of accountability at bright-squid: A case study and a developer survey”. In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2023, pp. 2008–2019.

[83] Charles Koutcheme et al. “Evaluating Distance Measures for Program Repair”. In: *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1* (2023). DOI: 10.1145/3568813.3600130.

[84] M. Kuhail et al. “Characterizing Visual Programming Approaches for End-User Developers: A Systematic Review”. In: *IEEE Access* 9 (2021), pp. 14181–14202. DOI: 10.1109/ACCESS.2021.3051043.

[85] Adrian Kuhn, David Erni, and Oscar Nierstrasz. “Embedding spatial software visualization in the IDE: an exploratory study”. In: *Proceedings of the 5th international symposium on Software visualization*. 2010, pp. 113–122.

[86] Hiroki Kuramoto et al. “Understanding the characteristics and the role of visual issue reports”. In: *Empirical Software Engineering* 29.4 (2024), p. 89.

[87] Kenton Lee et al. “Pix2struct: Screenshot parsing as pretraining for visual language understanding”. In: *International Conference on Machine Learning*. PMLR. 2023, pp. 18893–18912.

[88] Junnan Li et al. “Blip: Bootstrapping language-image pre-training for unified vision-language understanding and generation”. In: *International conference on machine learning*. PMLR. 2022, pp. 12888–12900.

[89] Liunian Harold Li et al. “Visualbert: A simple and performant baseline for vision and language”. In: *arXiv preprint arXiv:1908.03557* (2019).

[90] Xiaoqing Li, Jiansheng Yang, and Jinwen Ma. “Recent developments of content-based image retrieval (CBIR)”. In: *Neurocomputing* 452 (2021), pp. 675–689. DOI: 10.1016/J.NEUCOM.2020.07.139.

[91] Zhifang Liao et al. “Detecting duplicate questions in stack overflow via semantic and relevance approaches”. In: *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE. 2021, pp. 111–119.

[92] Chin-Yew Lin. “Rouge: A package for automatic evaluation of summaries”. In: *Text summarization branches out*. 2004, pp. 74–81.

[93] Chin-Yew Lin and FJ Och. “Looking for a few good metrics: ROUGE and its evaluation”. In: *Ntcir workshop*. 2004.

[94] Chuwei Luo et al. “Geolayoutlm: Geometric pre-training for visual information extraction”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2023, pp. 7092–7101.

[95] Amalia Luque et al. “The impact of class imbalance in classification performance metrics based on the binary confusion matrix”. In: *Pattern Recognit.* 91 (2019), pp. 216–231. DOI: 10.1016/J.PATCOG.2019.02.023.

[96] Walid Maalej, Maleknaz Nayebi, and Guenther Ruhe. “Data-driven requirements engineering-an update”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE. 2019, pp. 289–290.

[97] Abdulkarim Malkadi, Mohammad Alahmadi, and Sonia Haiduc. “A study on the accuracy of ocr engines for source code transcription from programming screencasts”. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. 2020, pp. 65–75.

[98] Potsawee Manakul, Adian Liusie, and Mark JF Gales. "Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models". In: *arXiv preprint arXiv:2303.08896* (2023).

[99] Dimitrios Markonis, Roger Schaer, and H. Müller. "Evaluating multimodal relevance feedback techniques for medical image retrieval". In: *Information Retrieval Journal* 19 (2015), pp. 100 –112. DOI: 10.1007/s10791-015-9260-4.

[100] Dimitrios Markonis, Roger Schaer, and H. Müller. "Multi-modal Relevance Feedback for Medical Image Retrieval". In: (2014), pp. 20–23.

[101] R. Mehrotra, Kamesh Namuduri, and N. Ranganathan. "Gabor filter-based edge detection". In: *Pattern Recognition* 25 (Dec. 1992), pp. 1479–1494. DOI: 10.1016/0031-3203(92)90121-X.

[102] Jamshed Memon, Maira Sami, and Rizwan Ahmed Khan. "Handwritten Optical Character Recognition (OCR): A Comprehensive Systematic Literature Review (SLR)". In: *IEEE Access* 8 (2020), pp. 142642–142668. DOI: 10.1109/ACCESS.2020.3012542.

[103] Tim Menzies et al. "Negative results for software effort estimation". In: *Empirical Software Engineering* 22 (2017), pp. 2658–2683.

[104] Ana Mlinarić, Martina Horvat, and Vesna Šupak Smolčić. "Dealing with the positive publication bias: Why you should really publish your negative results". In: *Biochemia medica* 27.3 (2017), pp. 447–452.

[105] Shunji Mori, Hirobumi Nishida, and Hiromitsu Yamada. *Optical character recognition*. John Wiley & Sons, Inc., 1999.

[106] Nanonets. *Nanonets-OCRvsTesseract*. https://nanonets.com/blog/ocr-with-tesseract/. Taken from the official website of Nanonets. 2023.

[107] Paolo Napoletano, Flavio Piccoli, and R. Schettini. "Semi-supervised anomaly detection for visual quality inspection". In: *Expert Syst. Appl.* 183 (2021), p. 115275. DOI: 10.1016/J.ESWA.2021.115275.

[108] Humza Naveed et al. “A comprehensive overview of large language models”. In: *arXiv preprint arXiv:2307.06435* (2023).

[109] Maleknaz Nayebi. “Analytical Release Management for Mobile Apps”. PhD thesis. PhD thesis, University of Calgary, 2018.

[110] Maleknaz Nayebi. “Data driven requirements engineering: Implications for the community”. In: *2018 IEEE 26th International Requirements Engineering Conference (RE)*. IEEE. 2018, pp. 439–441.

[111] Maleknaz Nayebi. “Eye of the mind: Image processing for social coding”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*. 2020, pp. 49–52.

[112] Maleknaz Nayebi and Bram Adams. “Image-based communication on social coding platforms”. In: *Journal of Software: Evolution and Process* (2023), e2609.

[113] Maleknaz Nayebi and Bram Adams. “Image-based communication on social coding platforms”. In: *Journal of Software: Evolution and Process* 36.5 (2024), e2609.

[114] Maleknaz Nayebi, Bram Adams, and Guenther Ruhe. “Release Practices for Mobile Apps–What do Users and Developers Think?” In: *2016 ieee 23rd international conference on software analysis, evolution, and reengineering (saner)*. Vol. 1. IEEE. 2016, pp. 552–562.

[115] Maleknaz Nayebi, Henry Cho, and Guenther Ruhe. “App store mining is not enough for app improvement”. In: *Empirical Software Engineering* 23 (2018), pp. 2764–2794.

[116] Maleknaz Nayebi, Homayoon Farahi, and Guenther Ruhe. “Which version should be released to app store?” In: *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE. 2017, pp. 324–333.

[117] Maleknaz Nayebi, Homayoon Farrahi, and Guenther Ruhe. “Analysis of marketed versus not-marketed mobile app releases”. In: *Proceedings of the 4th International Workshop on Release Engineering*. 2016, pp. 1–4.

[118] Maleknaz Nayebi and Guenther Ruhe. "An open innovation approach in support of product release decisions". In: *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*. 2014, pp. 64–71.

[119] Maleknaz Nayebi and Guenther Ruhe. "Analytical open innovation for value-optimized service portfolio planning". In: *Software Business. Towards Continuous Value Delivery: 5th International Conference, ICSOB 2014, Paphos, Cyprus, June 16-18, 2014. Proceedings 5*. Springer. 2014, pp. 273–288.

[120] Maleknaz Nayebi and Guenther Ruhe. "Analytical product release planning". In: *The art and science of analyzing software data*. Elsevier, 2015, pp. 555–589.

[121] Maleknaz Nayebi and Guenther Ruhe. "Asymmetric release planning: Compromising satisfaction against dissatisfaction". In: *IEEE Transactions on Software Engineering* 45.9 (2018), pp. 839–857.

[122] Maleknaz Nayebi and Guenther Ruhe. "Optimized functionality for super mobile apps". In: *2017 IEEE 25th international requirements engineering conference (RE)*. IEEE. 2017, pp. 388–393.

[123] Maleknaz Nayebi and Guenther Ruhe. *Trade-off service portfolio planning–a case study on mining the android app market*. Tech. rep. PeerJ PrePrints, 2015.

[124] Maleknaz Nayebi, Guenther Ruhe, and Thomas Zimmermann. "Mining treatment-outcome constructs from sequential software engineering data". In: *IEEE Transactions on Software Engineering* 47.2 (2019), pp. 393–411.

[125] Maleknaz Nayebi et al. "A longitudinal study of identifying and paying down architecture debt". In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE. 2019, pp. 171–180.

[126] Maleknaz Nayebi et al. "Analytics for Software Project Management–Where are We and Where do We Go?" In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. IEEE. 2015, pp. 18–21.

[127] Maleknaz Nayebi et al. “Anatomy of functionality deletion: an exploratory study on mobile apps”. In: *Proceedings of the 15th International Conference on Mining Software Repositories*. 2018, pp. 243–253.

[128] Maleknaz Nayebi et al. “App store mining is not enough”. In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE. 2017, pp. 152–154.

[129] Maleknaz Nayebi et al. “Crowdsourced exploration of mobile app features: A case study of the fort mcmurray wildfire”. In: *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Society Track (ICSE-SEIS)*. IEEE. 2017, pp. 57–66.

[130] Maleknaz Nayebi et al. “ESSMArT way to manage customer requests”. In: *Empirical Software Engineering* 24 (2019), pp. 3755–3789.

[131] Maleknaz Nayebi et al. “Hybrid labels are the new measure!” In: *IEEE Software* 35.1 (2017), pp. 54–57.

[132] Maleknaz Nayebi et al. “More insight from being more focused: analysis of clustered market apps”. In: *Proceedings of the International Workshop on App Market Analytics*. 2016, pp. 30–36.

[133] Maleknaz Nayebi et al. “Recommending and release planning of user-driven functionality deletion for mobile apps”. In: *Requirements Engineering* 29.4 (2024), pp. 459–480.

[134] Maleknaz Nayebi et al. “User driven functionality deletion for mobile apps”. In: *2023 IEEE 31st International Requirements Engineering Conference (RE)*. IEEE. 2023, pp. 6–16.

[135] Anh Nguyen-Duc et al. “Generative Artificial Intelligence for Software Engineering–A Research Agenda”. In: *arXiv preprint arXiv:2310.18648* (2023).

[136] Jordan Ott et al. “A deep learning approach to identifying source code in images and video”. In: *Proceedings of the 15th International Conference on Mining Software Repositories*. 2018, pp. 376–386.

[137] Jordan Ott et al. “Learning lexical features of programming languages from imagery using convolutional neural networks”. In: *Proceedings of the 26th conference on program comprehension*. 2018, pp. 336–339.

[138] Stack Overflow. *Few-Shot Example 1*. https://stackoverflow.com/questions/79044080/is-getenv-s-not-part-of-cstdlib/. Accessed: 2024-11-05. 2024.

[139] Stack Overflow. *Few-Shot Example 2*. https://stackoverflow.com/questions/79084406/trying-to-stack-2-columns-into-one-excel. Accessed: 2024-11-05. 2024.

[140] Kishore Papineni et al. “Bleu: a method for automatic evaluation of machine translation”. In: *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 2002, pp. 311–318.

[141] Lisa Park. *Research update: Improving the question-asking experience*. https://stackoverflow.blog/2019/09/26/research-update-improving-the-question-asking-experience/. Accessed: 2024-12-15. 2019.

[142] Federica Pepe et al. “How do Papers Make into Machine Learning Frameworks: A Preliminary Study on TensorFlow”. In: *33rd IEEE/ACM International Conference on Program Comprehension (ICPC 2025)*. 2025.

[143] Pew Research Center. *Americans' Social Media Use*. Accessed: 2024-11-05. 2024. URL: https://www.pewresearch.org/internet/2024/01/31/americans-social-media-use/.

[144] Alec Radford et al. “Learning transferable visual models from natural language supervision”. In: *International conference on machine learning*. PMLR. 2021, pp. 8748–8763.

[145] Shadikur Rahman, Faiz Ahmed, and Maleknaz Nayebi. “Mining Reddit Data to Elicit Students' Requirements During COVID-19 Pandemic”. In: *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*. 2023, pp. 76–84. DOI: 10.1109/REW57809.2023.00021.

[146] Faisal Rahutomo, Teruaki Kitasuka, Masayoshi Aritsugi, et al. “Semantic cosine similarity”. In: *The 7th international student conference on advanced science and technology ICAST*. Vol. 4. 1. University of Seoul South Korea. 2012, p. 1.

[147] Rajat Raina et al. “Classification with hybrid generative/discriminative models”. In: *Advances in neural information processing systems* 16 (2003).

[148] Aditya Ramesh et al. “Zero-Shot Text-to-Image Generation”. In: *arXiv preprint arXiv:2102.12092* (2021). URL: https://arxiv.org/abs/2102.12092.

[149] Ali Razavi, Aaron van den Oord, and Oriol Vinyals. “Generating Diverse High-Fidelity Images with VQ-VAE-2”. In: *Advances in Neural Information Processing Systems*. Vol. 32. 2019. URL: https://arxiv.org/abs/1906.00446.

[150] Ehud Reiter. “A structured review of the validity of BLEU”. In: *Computational Linguistics* 44.3 (2018), pp. 393–401.

[151] Robin Rombach et al. “High-Resolution Image Synthesis with Latent Diffusion Models”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 10684–10695. DOI: 10.1109/CVPR52688.2022.01042. URL: https://openaccess.thecvf.com/content/CVPR2022/html/Rombach_High-Resolution_Image_Synthesis_With_Latent_Diffusion_Models_CVPR_2022_paper.html.

[152] Guenther Ruhe, Maleknaz Nayebi, and Christof Ebert. “The vision: Requirements engineering in society”. In: *2017 IEEE 25th International Requirements Engineering Conference (RE)*. IEEE. 2017, pp. 478–479.

[153] Günther Ruhe and Maleknaz Nayebi. “What counts is decisions, not numbers—toward an analytics design sheet”. In: *Perspectives on Data Science for Software Engineering*. Elsevier, 2016, pp. 111–114.

[154] N. Sarika, N. Sirisala, and M. S. Velpuru. “CNN based Optical Character Recognition and Applications”. In: *2021 6th International Conference on Inventive Computation Technologies (ICICT)* (2021), pp. 666–672. DOI: 10.1109/ICICT50816.2021.9358735.

[155] R. Rani Saritha, V. Paul, and P. Ganesh Kumar. “Content based image retrieval using deep learning process”. In: *Cluster Computing* 22 (2018), pp. 4187 –4200. DOI: 10.1007/s10586-018-1731-0.

[156] Sk Golam Saroar and Maleknaz Nayebi. “Developers’ perception of GitHub Actions: A survey analysis”. In: *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*. 2023, pp. 121–130.

[157] SK Golam Saroar et al. “GitHub Marketplace: Driving Automation and Fostering Innovation in Software Development”. In: *2025 IEEE 32nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Journal First. 2025.

[158] SK Golam Saroar et al. “GitHub marketplace for automation and innovation in software production”. In: *Information and Software Technology* 175 (2024), p. 107522.

[159] E. Schoop, Forrest Huang, and Bjoern Hartmann. “UMLAUT: Debugging Deep Learning Programs using Program Structure and Model Behavior”. In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (2021). DOI: 10.1145/3411764.3445538.

[160] Nataliia Semenenko, Marlon Dumas, and Tõnis Saar. “Browserbite: Accurate Cross-Browser Testing via Machine Learning over Image Features”. In: *2013 IEEE International Conference on Software Maintenance*. 2013, pp. 528–531. DOI: 10.1109/ICSM.2013.88.

[161] Rodrigo FG Silva, Klérisson Paixão, and Marcelo de Almeida Maia. “Duplicate question detection in stack overflow: A reproducibility study”. In: *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*. IEEE. 2018, pp. 572–581.

[162] Raymond W. Smith. “An Overview of the Tesseract OCR Engine”. In: *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)* 2 (2007), pp. 629–633. DOI: 10.1109/ICDAR.2007.56.

[163] R. Sortino, S. Palazzo, and C. Spampinato. "Transformer-based Image Generation from Scene Graphs". In: *Comput. Vis. Image Underst.* 233 (2023), p. 103721. DOI: 10.48550/arXiv.2303.04634.

[164] Stack Exchange. *All Sites – Stack Exchange*. https://stackexchange.com. Archived from the original on 22 November 2019. Retrieved 26 March 2023. 2019.

[165] Stack Exchange. *All Sites – Stack Exchange*. https://data.stackexchange.com/stackoverflow/queries. Taken from the official website of Stack Exchange. 2023.

[166] Stack Overflow. *Duplicate-Detection*. https://stackoverflowteams.help/en/articles/8688772-close-and-reopen-questions. Taken from the official website of Stack Overflow. 2024.

[167] Stack Overflow. *Policy Guidelines*. https://meta.stackoverflow.com/questions/285551/why-should-i-not-upload-images-of-code-data-errors. Taken from the official website of Stack Overflow. 2024.

[168] Stack Overflow. *Policy Guidelines*. https://meta.stackoverflow.com/questions/303812/discourage-screenshots-of-code-and-or-errors. Taken from the official website of Stack Overflow. 2024.

[169] steveny. *How does the image OCR actually work in GPT-4?* 2024. URL: https://community.openai.com/t/how-does-the-image-ocr-actually-work-in-gpt-4/785890.

[170] Weijie Su et al. "Vl-bert: Pre-training of generic visual-linguistic representations". In: *arXiv preprint arXiv:1908.08530* (2019).

[171] Chengnian Sun et al. "A discriminative model approach for accurate duplicate bug report retrieval". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 2010, pp. 45–54.

[172] Ashish Sureka and Pankaj Jalote. "Detecting duplicate bug report using character n-gram-based features". In: *2010 Asia Pacific software engineering conference*. IEEE. 2010, pp. 366–374.

[173] Xuchen Tan. “Leveraging the Power of Images: Image Recommendation to Enhance Issue Reports”. In: (2024).

[174] Xuchen Tan et al. “ImageR: Enhancing Bug Report Clarity by Screenshots”. In: *arXiv preprint arXiv:2505.01925* (2025).

[175] Gemini Team et al. “Gemini: a family of highly capable multimodal models”. In: *arXiv preprint arXiv:2312.11805* (2023).

[176] The Small Business Blog. *TikTok vs Instagram - Users & Stats Compared for 2025*. https://thesmallbusinessblog.com/tiktok-vs-instagram/. Accessed: May 12, 2025. 2025.

[177] Sharuka Promodya Thirimanne et al. “One Documentation Does Not Fit All: Case Study of TensorFlow Documentation”. In: *arXiv preprint arXiv:2505.01939* (2025).

[178] Yuan Tian, Chengnian Sun, and David Lo. “Improved duplicate bug report identification”. In: *2012 16th European conference on software maintenance and reengineering*. IEEE. 2012, pp. 385–390.

[179] Hugo Touvron et al. “Llama: Open and Efficient Foundation Language Models”. In: *arXiv preprint arXiv:2302.13971* (2023). URL: https://arxiv.org/abs/2302.13971.

[180] Shubham Ugare et al. “Improving llm code generation with grammar augmentation”. In: *arXiv preprint arXiv:2403.01632* (2024).

[181] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).

[182] O. Vignesh, H. Mangalam, and S. Gayathri. “FPGA architecture for text extraction from images”. In: *Cluster Computing* (2018), pp. 1–10. DOI: 10.1007/s10586-017-1567-z.

[183] V. Vijaya et al. “DFIR-Net: Convolutional Neural Networks with Deep Learning for Content-based Image Retrieval”. In: *2023 IEEE Fifth International Conference on Advances in Electronics, Computers and Communications (ICAECC)* (2023), pp. 1–4. DOI: 10.1109/ICAECC59324.2023.10560230.

[184] Rafael Grompone Von Gioi et al. “LSD: A line segment detector”. In: *Image Processing On Line* 2 (2012), pp. 35–55.

[185] Dingbang Wang et al. “An Empirical Study on Leveraging Images in Automated Bug Report Reproduction”. In: *arXiv preprint arXiv:2502.15099* (2025).

[186] Dong Wang et al. “Understanding the role of images on stack overflow”. In: *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE. 2023, pp. 377–388.

[187] Dong Wang et al. “Understanding the Role of Images on Stack Overflow”. In: *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. 2023, pp. 377–388. DOI: 10.1109/MSR59073.2023.00059.

[188] Dongsheng Wang et al. “DocLLM: A layout-aware generative language model for multimodal document understanding”. In: *arXiv preprint arXiv:2401.00908* (2023).

[189] Jianfeng Wang et al. “Git: A generative image-to-text transformer for vision and language”. In: *arXiv preprint arXiv:2205.14100* (2022).

[190] Jiaqi Wang et al. “Review of large vision models and visual prompt engineering”. In: *Meta-Radiology* (2023), p. 100047.

[191] Liting Wang, Li Zhang, and Jing Jiang. “Duplicate question detection with deep learning in stack overflow”. In: *IEEE Access* 8 (2020), pp. 25964–25975.

[192] Rui Wang et al. “Medical Image Retrieval Method Based on Relevance Feedback”. In: (2012), pp. 650–662. DOI: 10.1007/978-3-642-35527-1_54.

[193] Xiaoyin Wang et al. “An approach to detecting duplicate bug reports using natural language and execution information”. In: *Proceedings of the 30th international conference on Software engineering*. 2008, pp. 461–470.

[194] Yidan Wang. "Enhancing software engineering with visual design and multimedia: An empirical and theoretical exploration". In: *Applied and Computational Engineering* (2024). DOI: 10.54254/2755-2721/74/20240478.

[195] Jason Wei et al. "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models". In: *arXiv preprint arXiv:2201.11903* (2022). URL: https://arxiv.org/abs/2201.11903.

[196] Thomas D White, Gordon Fraser, and Guy J Brown. "Improving random GUI testing with image-based widget detection". In: *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*. 2019, pp. 307–317.

[197] Peter Willett. "The Porter stemming algorithm: then and now". In: *Program* 40.3 (2006), pp. 219–223.

[198] M. Wong et al. "Natural Language Generation and Understanding of Big Code for AI-Assisted Programming: A Review". In: *Entropy* 25 (2023). DOI: 10.3390/e25060888.

[199] Jason Wu et al. "Webui: A dataset for enhancing visual ui understanding with web semantics". In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 2023, pp. 1–14.

[200] Mulong Xie et al. "Psychologically-inspired, unsupervised inference of perceptual groups of GUI widgets from GUI images". In: *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering*. 2022, pp. 332–343.

[201] Yang Xu et al. "Layoutlmv2: Multi-modal pre-training for visually-rich document understanding". In: *arXiv preprint arXiv:2012.14740* (2020).

[202] Yiheng Xu et al. "Layoutlm: Pre-training of text and layout for document image understanding". In: *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. 2020, pp. 1192–1200.

[203] Wenhua Yang and Chaochao Shen. "Understanding the Role of Stack Overflow in Supporting Software Development Tasks: A Research Perspective". In: *International Journal of Software Engineering and Knowledge Engineering* 33.07 (2023), pp. 1119–1148.

[204] Annie TT Ying. "Mining challenge 2015: Comparing and combining different information sources on the stack overflow data set". In: *The 12th working conference on mining software repositories*. 2015.

[205] Dawei Yuan, Guocang Yang, and Tao Zhang. "UI2HTML: utilizing LLM agents with chain of thought to convert UI into HTML code". In: *Automated Software Engg.* 32.2 (Apr. 2025). ISSN: 0928-8910. DOI: 10.1007/s10515-025-00509-5. URL: https://doi.org/10.1007/s10515-025-00509-5.

[206] Chengzhi Zhang et al. "Generative image AI using design sketches as input: Opportunities and challenges". In: *Proceedings of the 15th Conference on Creativity and Cognition*. 2023, pp. 254–261.

[207] Tao Zhang et al. "A Literature Review of Research in Bug Resolution: Tasks, Challenges and Future Directions". In: *The Computer Journal* 59.5 (May 2016), pp. 741–773. ISSN: 0010-4620. DOI: 10.1093/comjnl/bxv114. eprint: https://academic.oup.com/comjnl/article-pdf/59/5/741/7903156/bxv114.pdf. URL: https://doi.org/10.1093/comjnl/bxv114.

[208] Yun Zhang et al. "Multi-factor duplicate question detection in stack overflow". In: *Journal of Computer Science and Technology* 30 (2015), pp. 981–997.