CODEV: Issue Resolving with Visual Data

Linhao Zhang[1] **Daoguang Zan**[2*] **Quanshun Yang**[1] **Zhirong Huang**[2]
Dong Chen[3] **Bo Shen**[3] **Tianyu Liu**[4] **Yongshun Gong**[1] **Pengjie Huang**[5]
Xudong Lu[1*] **Guangtai Liang**[3] **Lizhen Cui**[1] **Qianxiang Wang**[3]
[1]Shandong University [2]Chinese Academy of Sciences
[3]Huawei Co., Ltd [4]Peking University [5]Lingzhi-zhiguang Co., Ltd
linhaoz@mail.sdu.edu.cn daoguang@iscas.ac.cn dongxul@sdu.edu.cn

Abstract

Large Language Models (LLMs) have advanced rapidly in recent years, with their applications in software engineering expanding to more complex repository-level tasks. GitHub issue resolving is a key challenge among these tasks. While recent approaches have made progress on this task, they focus on textual data within issues, neglecting visual data. However, this visual data is crucial for resolving issues as it conveys additional knowledge that text alone cannot. We propose CODEV, the first approach to leveraging visual data to enhance the issue-resolving capabilities of LLMs. CODEV resolves each issue by following a two-phase process: data processing and patch generation. To evaluate CODEV, we construct a benchmark for visual issue resolving, namely Visual SWE-bench. Through extensive experiments, we demonstrate the effectiveness of CODEV, as well as provide valuable insights into leveraging visual data to resolve GitHub issues[1].

1 Introduction

Large Language Models (LLMs) have advanced rapidly in recent years, with their applications in the field of software engineering becoming increasingly widespread (Zan et al., 2023; Zheng et al., 2023; Zhang et al., 2023b; Chen et al., 2024b). Currently, LLMs' applications in software engineering have gradually expanded tasks at the code line and function level to more challenging repository-level tasks (Zhang et al., 2023a; Liu et al., 2024). Within repository-level tasks, GitHub issue resolving is a key challenge, where LLMs are tasked to resolve the issue based on the issue description and the defective codebase. (Jimenez et al., 2024; Xia et al., 2024). This task can accelerate program repair and is crucial for improving development efficiency.

Although recent approaches have made progress on this task, they focus exclusively on textual data in GitHub issues, neglecting visual data such as screenshots, diagrams, and videos (Chen et al., 2024a; Yang et al., 2024a; Zhang et al., 2024; Xia et al., 2024). However, this visual data is crucial for resolving issues, as it conveys additional knowledge that text alone cannot, including actual results, expected results, and error messages. Figure 1 shows a specific example where visual data illustrates the running result. Moreover, we statistically analyze SWE-bench (Jimenez et al., 2024), the most popular benchmark for issue resolving. The result shows that over 5% of GitHub issues contain visual data, with even higher percentages in visualization libraries like seaborn[2] and matplotlib[3], where they reach 45.5% and 27.2% respectively. This analysis further highlights the importance of resolving visual GitHub issues. However, existing approaches struggle to resolve them effectively, as they overlook visual data, which calls for new solutions that leverage visual data.

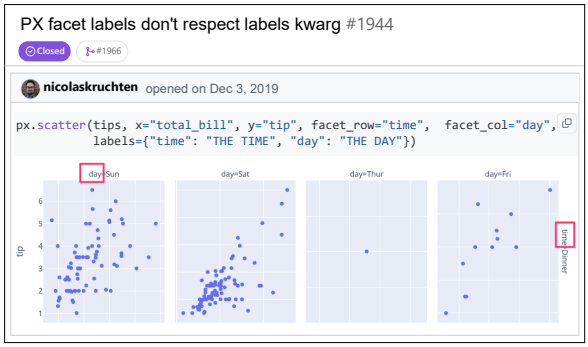


Figure 1: An example of a visual GitHub issue from Plotly issue #1944. The visual data illustrates that the label parameters ("time" and "day") do not take effect.

An intuitive approach is to extract visual data from the issue and include it as part of the prompt. While this approach seems to leverage visual data, it requires models with advanced multimodal and

*Corresponding authors
[1]https://github.com/luolin101/CodeV
[2]https://github.com/mwaskom/seaborn
[3]https://github.com/matplotlib/matplotlib

Findings of the Association for Computational Linguistics: ACL 2025, pages 7350–7361
July 27 - August 1, 2025

coding capabilities. Currently, only the latest commercial models, GPT-4o (OpenAI, 2024) and Claude 3.5 Sonnet (Anthropic, 2024), barely meet these requirements, but their capabilities remain highly limited. Moreover, these models are less suitable for issue resolving due to high computational costs. Based on our analysis, using these models within the popular SWE-agent approach (Yang et al., 2024a) to run through all issues in SWE-bench once is estimated to cost an average of over $4,700 (Xia et al., 2024), which imposes a significant financial burden on researchers. To address this, we propose CODEV, the first approach that leverages visual data to enhance the issue-resolving capabilities of LLMs at low cost. To resolve each issue, CODEV follows a two-phase process: data processing and patch generation. In the data processing phase, CODEV processes the visual data within the issue from both local and holistic perspectives. This phase produces fine-grained descriptions of the visual data and a structured summary of the entire issue. In the patch generation phase, CODEV leverages the processed information to assist LLMs in generating a patch to resolve the issue.

To evaluate our approach, we construct a benchmark specifically designed for evaluating visual GitHub issue resolving, called Visual SWE-bench. The benchmark comprises 133 task instances spanning 11 open-source GitHub repositories, each of which has undergone rigorous selection. Finally, we conduct a series of experiments to validate the effectiveness of our approach. Experimental results demonstrate that CODEV achieves a round 63.13% relative improvement in the percentage of resolved instances on Visual SWE-bench compared to Agentless. Additionally, through case studies, we analyze the role of each component of CODEV, providing insights into leveraging visual data to resolve issues. Overall, the contributions of this paper are as follows:

- We propose CODEV, a simple yet novel approach that leverages visual data to enhance the issue-resolving capabilities of LLMs.
- We construct a benchmark designed to evaluate the performance of LLMs in resolving visual GitHub issues, namely Visual SWE-bench. The benchmark comprises 133 realistic software engineering tasks sourced from 11 open-source GitHub repositories.
- We validate the effectiveness of our approach through a series of experiments and conduct in-depth analysis and summarization of the experimental results.

2 Approach

Figure 2 illustrates an overview of CODEV, which consists of two phases: data processing and patch generation. The first phase processes visual data and the second phase uses the processed information to assist LLMs in generating a patch. Below is a detailed description of each phase.

2.1 Data Processing

To process the issue's visual data, we adopt two components: fine-grained description and structured summarization. For fine-grained description, the Vision-Language Model (VLM) first generates an independent description for each piece of visual data based on its content. It then provides a contextual description that relates this data to the issue, resulting in a fine-grained description. In structured summarization, the VLM produces a summary that breaks down the complex issue into several clear sections. Below, we detail the implementation of each component.

2.1.1 Fine-Grained Description

We design the fine-grained description component to generate textual representations of the visual data. This component draws inspiration from how humans process visual data. When encountering visual data in an issue, humans first identify its raw features and then analyze its function in the context of the problem. Inspired by this, we design a two-step process to generate fine-grained descriptions.

Step 1: Independent Description. In the first step, we instruct the VLM to describe each piece of visual data based solely on its content. For visual data consisting purely of text, such as a screenshot of error logs, the VLM extracts the text and outputs it in Markdown format. For other types of visual data, like images or videos with non-textual content, the VLM generates a detailed description capturing all details.

Step 2: Contextual Description. In the second step, the VLM is prompted with the complete problem statement to establish the context. Subsequently, it is tasked with providing a comprehensive description and analysis of the visual data based on this contextual understanding. During this step, the VLM tightly connects the visual data with the problem's context to analyze its function.

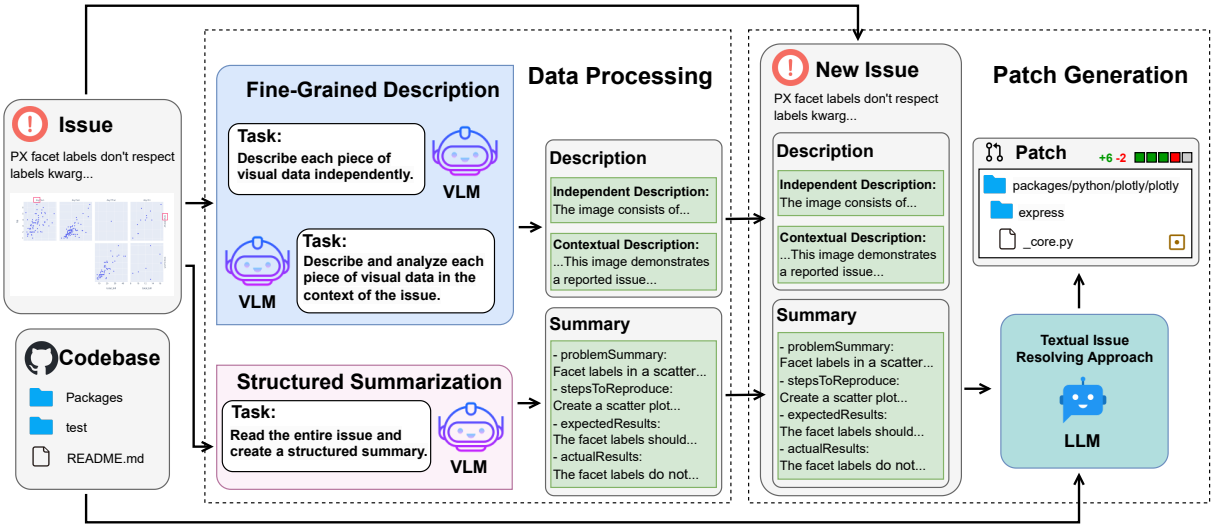


Figure 2: Overview of CODEV.

Through these two steps, we obtain fine-grained descriptions of all visual data in the issue. These descriptions capture not only the intrinsic features of the visual data but also its critical function in the problem's context.

2.1.2 Structured Summarization

Some GitHub issues are described in a structured format, including reproduction steps, expected results, actual results, and so on. This format enhances clarity and makes the key aspects of issues easier to understand. Inspired by this, we propose generating a structured summary to enrich issues and reduce the difficulty of understanding them.

In the structured summarization component, the VLM is prompted with the complete problem statement, including the visual data. It is then tasked with understanding and analyzing the issue to generate a structured summary. To guide this process, we supply the VLM with a template that consists of the following fields: a brief problem summary, background information, reproduction steps, expected results, actual results, descriptions of visual data, and additional notes. However, not all issues may fit perfectly with this template. Therefore, we allow the VLM to skip irrelevant or unclear fields. Additionally, the summary can also include new fields if needed, as long as it remains clear and useful for resolving the issue.

Unlike fine-grained description, which focuses on generating representations of visual data, structured summarization aims to provide an overview of the entire issue. It not only covers visual data but also gives a deeper understanding of the problem. These two components complement each other: fine-grained description captures the detailed features of local visual data, while structured summarization synthesizes global information. Through these components, we ensure that visual data is processed effectively to support LLMs in understanding and resolving the issue. Prompts related to these components are listed in Appendix A.

2.2 Patch Generation

After generating fine-grained descriptions and a structured summary in the data processing phase, the patch generation phase leverages this information to generate a patch. To support LLMs in efficiently utilizing this information, we splice them into the original issue. Specifically, the visual data is converted into fine-grained descriptions, and the issue is enriched with a structured summary, with an example provided in Appendix B.

To enhance the ability of LLMs to resolve textual issues, various approaches have been proposed. These approaches take different forms: some are *agent-based*, equipping LLMs with a set of tools that allow the agent to autonomously perform actions (Chen et al., 2024a; Yang et al., 2024a; Zhang et al., 2024); others are *agentless* (Xia et al., 2024). Regardless of their form, they typically input the issue and codebase, with the output being a generated patch. CODEV combines these approaches through a unified interface, automating patch generation. At this point, the newly generated issue and its corresponding codebase are fed into the textual issue-resolving approach, where LLMs follow predefined instructions to generate a patch.

3 Visual SWE-bench Benchmark

In current benchmarks for the issue-resolving task, only the recently released SWE-bench Multimodal (Yang et al., 2024b) focuses on visual issues. However, as of writing, SWE-bench Multimodal[4] lacks evaluation fields, and its evaluation script has not been made public, making it unsuitable for evaluation. To evaluate CodeV, we construct a benchmark for resolving visual GitHub issues, namely Visual SWE-bench. Below, we detail our benchmark construction process and its key features.

3.1 Construction

From the 2,294 instances in SWE-bench, we identify 128 task instances whose problem statement contains visual data. These visual data are presented through hyperlinks, with images embedded using HTML or Markdown syntax and videos provided as plain text hyperlinks. Building on these identified task instances, we adopt a four-stage construction process to further expand the tasks and conduct rigorous verification on all instances.

1. **Repositories selection and pull requests Collection.** We analyze the 128 task instances from SWE-bench, and the results show that most of them originate from visualization libraries. To expand our benchmark, we select three additional popular open-source visualization libraries (plotly.py, networkx, and altair) and crawl all their pull requests (PRs) from GitHub. Since SWE-bench only includes PRs created before August 2023, we select repositories with at least 10 visual task instances from SWE-bench (matplotlib, sympy, sphinx, and seaborn) and collect recent PRs from these repositories. These two rounds of collection yield approximately 10,000 PRs.

2. **Candidate instance construction.** Candidate instances are constructed from the collected PRs through the following steps:

   (1) We select only merged PRs that resolve at least one issue and include modifications to test files.

   (2) For each PR, we extract the text of all resolved issues, retaining only those PRs where the issue text contained hyperlinks to images or videos.

   (3) For qualifying PRs, we gather detailed information, including "instance ID", "patch", "test patch", and so on.

   This process results in 38 candidate instances from approximately 10,000 PRs.

3. **Execution verification.** For each candidate instance, we meticulously set up the runtime environment and testing commands, removing any instances that failed due to installation or runtime errors. Next, we apply the test patch to each instance and record the test results both before and after applying the gold patch. Instances without any tests where the status changes from fail to pass are excluded. This process leaves 31 viable candidate instances.

4. **Human verification.** We conduct human verification on 159 instances, comprising 128 task instances from SWE-bench and 31 candidate instances filtered through the previous stages. Each instance is evaluated based on the following criteria:

   (1) Whether the visual data can be fully converted to text.

   (2) Whether the visual data is essential for resolving the instance.

   (3) Whether the problem description contains sufficient information for effective resolution.

   Using these criteria, we exclude 4 instances where visual data can be fully converted to text via Optical Character Recognition (OCR) and 4 instances where visual data is not essential for resolution. Additionally, based on the content of their test cases, we exclude 18 instances that cannot be resolved due to insufficient problem descriptions. This process results in a curated, high-quality benchmark of 133 task instances. Additional human verification details are provided in Appendix C.

3.2 Features

As shown in Figure 3, Visual SWE-bench comprises 133 visual task instances sourced from 11 open-source GitHub repositories. These instances cover a wide range of functionalities, including but not limited to data visualization, machine learning, and document generation. This diverse set of tasks provides a comprehensive benchmark for evaluating the performance of LLMs in resolving visual issues automatically.

[4] https://www.swebench.com/multimodal.html

Repository	CodeBase		Issue Text	Gold Patch			Tests	Images	
	# Files	# Lines	# Length	# Lines	# Files	# Func.	# Lines	# File Size	# Resolution
altair	499	90K	98.5	27	2.5	3	15	24.12	99K
astropy	1578	445K	1352.5	10	1	2	69	19.76	170K
matplotlib	2388	592K	175	9	1	2	89.5	23.58	307K
networkx	849	108K	155	23	1	1	26	21.38	307K
plotly.py	14302	587K	44	15	2	1	11.5	23.58	307K
pylint	2712	92K	100	126	4	9	10	23.46	307K
scikit-learn	1343	277K	641	22	1	2	12	23.58	307K
seaborn	295	70K	143.5	13.5	2	3	109.5	23.1	307K
sphinx	1436	308K	157	8	1	2	35	27.08	286K
sympy	1907	477K	125	21	1	2	36	24.26	275K
xarray	320	123K	220.5	5	1	2	229.5	25.08	275K
mean	2512	288K	292	25.40	1.59	2.63	58.45	23.54	268K
max	14302	592K	1352.5	126	4	9	229.5	27.08	307K

Table 1: Summary statistics for Visual SWE-bench. The term "CodeBase # Files and # Lines" denotes the total count of files and lines within the codebase. "Issue Text # Length" indicates the median word count in the problem statement. "Gold Patch # Lines, # Files, and # Func." reflects the median number of lines, files, and functions modified per patch stored in the repository. "Tests # Lines" signifies the median line count of code present in test cases. "Images # File Size and # Resolution" represents both the median image file size (KB) and resolution (pixels).

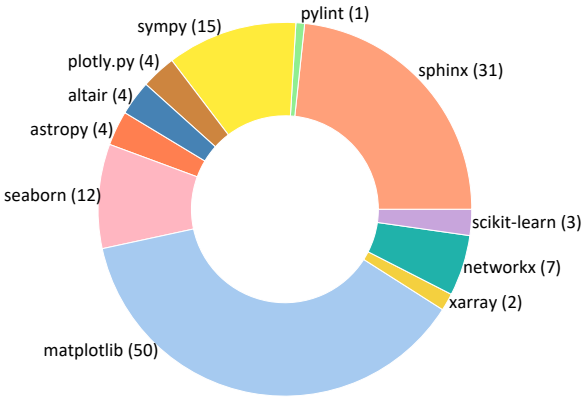


Figure 3: Distribution of Visual SWE-bench task instances across 11 open-source GitHub repositories.

Data statistics. Table 1 summarizes key statistics for the repositories in Visual SWE-bench, emphasizing their diversity and representativeness. Repository sizes range from 295 to 14,302 files and 70K to 592K lines of code, illustrating structural variation. Problem statements vary widely in length, with median word counts from 44 to 1,353, reflecting differences in task comprehension demands. Gold patches show diverse modification scopes, with median changes spanning 5 to 126 lines, indicating varying solution complexities. This broad spectrum of task characteristics provides a robust benchmark for evaluating LLMs' performance in resolving visual issues.

Visual data distribution. Across all Visual SWE-bench tasks, we identify 217 images and 2 videos, spanning a diverse range of visual processing challenges grouped into seven categories. These include code screenshots (21), error messages (8), and system information (2), which are linked to specific code library entities to facilitate error identification. Other categories include data visualizations (140), documentation results (33), function formulas (13), and keyboard shortcuts (2), illustrating challenges such as generating complex statistics and utilizing code functions within specific libraries. Additionally, two instances feature GIFs (matplotlib_matplotlib-19763) and videos (matplotlib_matplotlib-25631), providing more dynamic and detailed depictions of these challenges.

4 Experiments

4.1 Experimental Setup

Models. To execute CODEV for resolving visual issues, two model types are required: a VLM for processing visual data and an LLM for generating patches. To demonstrate the effectiveness of CODEV, we specifically avoid commercial models and use open-source models in our experiments. For the VLM, we select Qwen2-VL (Wang et al., 2024), a model renowned for its robust visual understanding capabilities, using three versions: 2B, 7B, and 72B. For the LLM, we choose two models: DeepSeek-V2.5 (DeepSeek-AI, 2024) and Qwen2.5-Coder-32B (Hui et al., 2024), both recognized for their powerful coding capabilities.

Approach	Model	Resolved (%)
Evaluation results on 111 instances from Visual SWE-bench		
Honeycomb (Honeycomb, 2024) 🔒	NA	10.81 (12)
Amazon Q Developer Agent (AWS, 2024) 🔒	NA	9.01 (10)
Factory Code Droid (Factory, 2024) 🔒	NA	9.01 (10)
AutoCodeRover (Zhang et al., 2024)	GPT 4o (2024-05-13)	10.81 (12)
AppMap Navie (AppMap, 2024) 🔒	GPT 4o (2024-05-13)	9.01 (10)
SWE-agent (Yang et al., 2024a)	Claude 3.5 Sonnet	6.31 (7)
	GPT 4 (1106)	8.11 (9)
	GPT 4o (2024-05-13)	1.80 (2)
RAG (Jimenez et al., 2024)	Claude 3 Opus	2.70 (3)
	Claude 2	0.90 (1)
CODEV + Agentless (Ours)	Qwen2-VL-72B + DeepSeek-V2.5	11.71 (13)
	Qwen2-VL-2B + Qwen2.5-Coder-32B	**13.51 (15)**
	Qwen2-VL-7B + Qwen2.5-Coder-32B	**13.51 (15)**
	Qwen2-VL-72B + Qwen2.5-Coder-32B	11.71 (13)
Evaluation results on all instances from Visual SWE-bench		
Agentless (Xia et al., 2024)	DeepSeek-V2.5	6.02 (8)
	Qwen2.5-Coder-32B	7.52 (10)
Agentless Plus	Qwen2-VL-72B	0.75 (1)
CODEV + Agentless (Ours)	Qwen2-VL-72B + DeepSeek-V2.5	9.77 (13)
	Qwen2-VL-2B + Qwen2.5-Coder-32B	**12.78 (17)**
	Qwen2-VL-7B + Qwen2.5-Coder-32B	**12.78 (17)**
	Qwen2-VL-72B + Qwen2.5-Coder-32B	11.28 (15)

Table 2: Results on Visual SWE-bench. The 111 instances are the overlapping instances between Visual SWE-bench and SWE-bench. 🔒 indicates closed-source approaches.

Baselines. In the patch generation phase, CODEV combines textual issue-resolving approaches. We specifically adopt the open-source Agentless approach (Xia et al., 2024), which resolves issues through a simple localization and repair process. We also compare CODEV with several textual issue-resolving approaches, including open-source and closed-source commercial products. These approaches have demonstrated strong performance on SWE-bench. To further contrast with VLM-based approaches, we design Agentless Plus, a modified version of Agentless that supports VLMs in processing visual data in issues to resolve them.

Metrics. We use Resolved (%) as our evaluation metric. The metric represents the percentage of Visual SWE-bench instances that have been successfully resolved. More details about the experiments can be found in Appendix D.

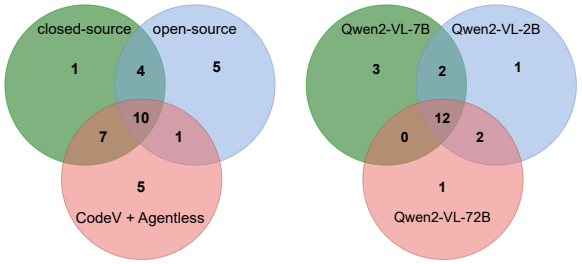
4.2 Evaluation

4.2.1 Main Results

Table 2 presents the results of all approaches. The results show that CODEV significantly enhances the issue-resolving capabilities of the LLM by leveraging visual data. Compared to all benchmarks, CODEV achieves the best performance. When combined with Agentless, CODEV achieves over a 50% relative improvement, whether using DeepSeek-V2.5 or Qwen2.5-Coder-32B to resolve issues. The performance of CodeV highlights the value of leveraging visual data to help LLMs understand and resolve issues.

Figure 4(a) depicts the distribution of issues resolved by CODEV compared to both closed-source and open-source baseline approaches. Notably, CODEV can resolve certain issues that either open-source or closed-source approaches cannot resolve. Furthermore, CODEV successfully resolves some complex issues that neither category of approaches could solve. This highlights not only the advantages of CODEV but also the importance of leveraging visual data to resolve issues.

From Table 2, it is evident that the performance of the VLM does not significantly impact CODEV. Among the three versions of Qwen2-VL, the 72B model is the most powerful, while the 7B and



(a) All approaches on 111 instances.

(b) CODEV with VLMs of different model sizes on all instances.

Figure 4: Venn diagrams of issues resolved from Visual SWE-bench.

Approach	Resolved (%)
CODEV + Agentless	**11.28 (15)**
w/o Fine-Grained Description	9.77 (13)
w/o Independent Description	9.02 (12)
w/o Contextual Description	9.02 (12)
w/o Structured Summarization	7.52 (10)

Table 3: Ablation studies on Visual SWE-bench (133 instances). The VLM is Qwen2-VL-72B and the LLM is Qwen2.5-Coder-32B.

2B models exhibit progressively weaker capabilities. However, even with the lower-performing 7B and 2B models, CODEV maintained robust issue-resolving capabilities, even outperforming the 72B model. Additionally, Figure 4(b) further illustrates the distribution of resolved issues across different VLMs. The issues resolved do not overlap entirely, indicating that each VLM has its strengths in processing different types of issues. This indicates that despite differences in VLM performance, CODEV can still exert the capabilities of VLM, resolve issues stably, and demonstrate strong robustness.

Additionally, we observe that using the VLM alone, while it leverages visual data, does not yield satisfactory results. For example, Agentless Plus combined with Qwen2-VL-72B resolves only one issue. This is primarily due to its weak coding capabilities. In comparison, CODEV effectively combines the VLM's visual understanding ability with the LLM's coding capabilities. This integration allows LLMs to leverage visual data to resolve issues at a low cost, making it a promising solution.

4.2.2 Analysis of Ablation Studies

We conduct a series of ablation studies on Visual SWE-bench, and the results in Table 3 show that

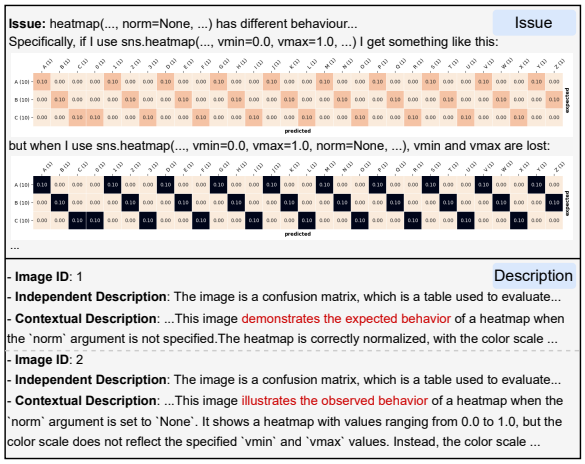


Figure 5: Fine-grained description example for the instance mwaskom_seaborn-3276, offering detailed insights into the visual data.

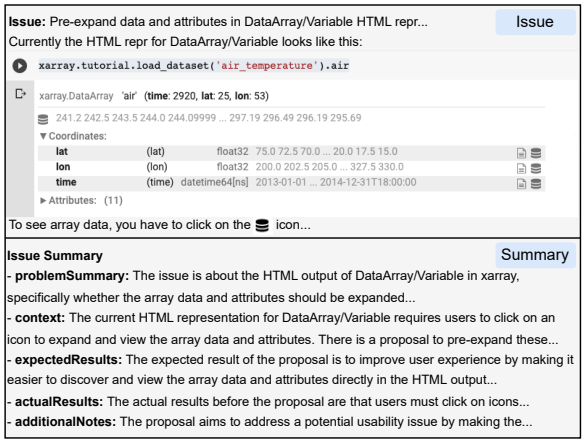


Figure 6: Structured summarization example for the instance pydata_xarray-4182, demonstrating a concise representation of its key information.

removing any component of CODEV leads to a decline in performance. This led us to further investigate the functions of the components in the data processing phase.

Analysis of Fine-Grained Description. The fine-grained description process consists of two steps: independent description and contextual description. In the independent description, the VLM captures the raw features of visual data, providing a direct and detailed representation. However, why is contextual description also necessary? Figure 5 shows an issue that is difficult to resolve without the contextual description. The figure shows two images, and the contextual description analyzes their respective function, explaining the information conveyed by each. In contrast, the independent description provides only a general overview,

missing critical details needed for a complete understanding of the issue. These details are essential for LLMs to grasp the issue accurately. Thus, while the independent description captures the raw features of the visual data, the contextual description extracts deeper, more nuanced information. Together, these two steps work in tandem to provide a comprehensive understanding of the visual data.

Analysis of Structured Summarization. As shown in Table 3, removing structured summarization significantly undermines the performance of CODEV. To explain this phenomenon, Figure 6 presents an issue that is more easily resolved with a summary. The summary generated by CODEV breaks down the complex issue into clear, digestible sections, providing LLMs with a full understanding of the issue's background, expected outcomes, and actual results. This structured format also helps LLMs grasp the core content more effectively. While the fine-grained description component attempts to convey the meaning of the visual data, relying solely on this still presents challenges in fully understanding the issue. By combining visual and textual data, the structured summary offers LLMs a more holistic understanding of the issue.

5 Related Works

Issue Resolving Approaches. To assist LLMs in resolving GitHub issues, many approaches have already been proposed. Retrieval Augmented Generation (RAG) (Jimenez et al., 2024) is a direct approach that resolves the issue by first extracting relevant code snippets from the repository and then using them to prompt LLMs to generate a patch. SWE-agent (Yang et al., 2024a) meticulously designs an agent-computer interface (ACI) that enables LLM agents to interact with repository environments to solve software engineering tasks. AutoCodeRover (Zhang et al., 2024) combines LLMs with code search, utilizes program structure, and conducts iterative searches for program improvement. CodeR (Chen et al., 2024a) is a multi-agent approach for issue-resolving tasks, adopting a multi-agent framework and pre-defined task graphs. Agentless (Xia et al., 2024) points out the limitations of using agents and proposes a simple two-phase process of localization and repair to solve software development problems. However, these existing approaches overlook visual data within issues. CODEV bridges this gap by processing visual data from both local and holistic perspectives, enhancing the capabilities of LLMs to resolve complex visual issues.

Code Generation Benchmarks. Code generation has long been a measure of LLMs performance (Austin et al., 2021). The emergence of HumanEval (Chen et al., 2021) provides a standardized framework for evaluating code generation models. In subsequent years, various benchmarks have been developed to enhance HumanEval by adding extensions to different languages (Cassano et al., 2022; Athiwaratkun et al., 2023; Orlanski et al., 2023), introducing variations in edit scope (Yu et al., 2024; Du et al., 2023), presenting similar yet novel code completion tasks (Muennighoff et al., 2024), and conducting more extensive testing (Liu et al., 2023). With the development of LLMs, existing benchmarks struggle to explore the boundaries of state-of-the-art LLMs' capabilities. To address this, SWE-bench (Jimenez et al., 2024) offers a direction by researching real-world GitHub issues, serving as a challenging benchmark for evaluating next-generation LLMs. Building on this, SWE-bench-Java (Zan et al., 2024) extends the benchmark to the Java ecosystem, creating a multilingual benchmark. Similarly, the latest work, SWE-bench Multimodal (Yang et al., 2024b) offers a multimodal upgrade to the benchmark, focusing on visual JavaScript problems. Given Python's increasing role in fields like data science, machine learning, and visualization, where visual data is crucial, we construct Visual SWE-bench focusing on visual issues in Python. By incorporating real-world visual issues, Visual SWE-bench encourages researchers to leverage visual data in solving complex software challenges.

6 Conclusion

We propose CODEV, an approach that leverages visual data to resolve issues automatically. It processes visual data and provides LLMs with valuable information that enhances their ability to resolve issues. To evaluate CODEV, we construct a benchmark for visual issue resolving, namely Visual SWE-bench. Through extensive experiments, we demonstrate the effectiveness of CODEV and find that it maintains robust performance across VLMs with varying model sizes. Additionally, through case studies, we analyze the function of each component of CODEV, offering insights on leveraging visual data to resolve issues.

Limitations

Although this study offers valuable insights into leveraging visual data to resolve GitHub issues, several limitations should be acknowledged:

- Due to the randomness in the responses generated by LLMs, there is a potential threat to the experimental results. Despite repeating each experiment twice to mitigate this, minor fluctuations in results may still occur.
- Due to the lack of suitable benchmarks, our experiments are conducted solely on the self-constructed benchmark. However, we conduct comprehensive experiments and analyses to validate our approach, and we hope future research will develop more publicly available benchmarks to further explore this direction.
- Due to the high costs of GPT-4o and Claude 3.5 Sonnet, we don't include them in our comparative experiments. Based on our estimates, using these models within the SWE-agent approach under similar experimental conditions would cost thousands of dollars. Nevertheless, we evaluate CODEV using two LLMs and three VLMs, conducting extensive experiments that confirm its effectiveness.

References

Anthropic. 2024. Introducing claude 3.5 sonnet. https://www.anthropic.com/news/claude-3-5-sonnet.

AppMap. 2024. Appmap speedruns to the top of the swe bench leaderboard. https://appmap.io/blog/2024/06/20/appmap-navie-swe-bench-leader.

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, and Ramesh Nallapati. 2023. Multilingual evaluation of code generation models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

AWS. 2024. Amazon q developer the most capable generative ai–powered assistant for software development. https://aws.amazon.com/q/developer.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227*.

Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, Jie Wang, Xiao Cheng, Guangtai Liang, Yuchi Ma, Pan Bian, Tao Xie, and Qianxiang Wang. 2024a. Coder: Issue resolving with multi-agent and task graphs. *CoRR*, abs/2406.01304.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Xiangping Chen, Xing Hu, Yuan Huang, He Jiang, Weixing Ji, Yanjie Jiang, Yanyan Jiang, Bo Liu, Hui Liu, Xiaochen Li, et al. 2024b. Deep learning-based software engineering: Progress, challenges, and opportunities. *arXiv preprint arXiv:2410.13110*.

DeepSeek-AI. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *Preprint*, arXiv:2405.04434.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *CoRR*, abs/2308.01861.

Factory. 2024. Factory bringing autonomy to software engineering. https://www.factory.ai.

Honeycomb. 2024. Honeycomb. https://honeycomb.sh.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. 2024. Graphcoder: Enhancing repository-level code completion via code context graph-based retrieval and language model. *CoRR*, abs/2406.07003.

Niklas Muennighoff, Qian Liu, Armel Randy Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2024. Octopack: Instruction tuning code large language models. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

OpenAI. 2024. Hello gpt-4o. `https://openai.com/index/hello-gpt-4o`.

Gabriel Orlanski, Kefan Xiao, Xavier Garcia, Jeffrey Hui, Joshua Howland, Jonathan Malmaud, Jacob Austin, Rishabh Singh, and Michele Catasta. 2023. Measuring the impact of programming language distribution. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 26619–26645. PMLR.

Peng Wang, Shuai Bai, Sinan Tan, Shijie Wang, Zhihao Fan, Jinze Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Yang Fan, Kai Dang, Mengfei Du, Xuancheng Ren, Rui Men, Dayiheng Liu, Chang Zhou, Jingren Zhou, and Junyang Lin. 2024. Qwen2-vl: Enhancing vision-language model's perception of the world at any resolution. *arXiv preprint arXiv:2409.12191*.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *CoRR*, abs/2407.01489.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024a. Swe-agent: Agent-computer interfaces enable automated software engineering. *CoRR*, abs/2405.15793.

John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press. 2024b. Swe-bench multimodal: Do AI systems generalize to visual software domains? *CoRR*, abs/2410.03859.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 37:1–37:12. ACM.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2023. Large language models meet nl2code: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 7443–7464. Association for Computational Linguistics.

Daoguang Zan, Zhirong Huang, Ailun Yu, Shaoxin Lin, Yifan Shi, Wei Liu, Dong Chen, Zongshuai Qi, Hao Yu, Lei Yu, Dezhi Ran, Muhan Zeng, Bo Shen, Pan Bian, Guangtai Liang, Bei Guan, Pengjie Huang, Tao Xie, Yongji Wang, and Qianxiang Wang. 2024. Swe-bench-java: A github issue resolving benchmark for java. *CoRR*, abs/2408.14354.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023a. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 2471–2484. Association for Computational Linguistics.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, pages 1592–1604. ACM.

Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2023b. A survey on language models for code. *CoRR*, abs/2311.07989.

Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2023. A survey of large language models for code: Evolution, benchmarking, and future trends. *CoRR*, abs/2311.10372.

A Prompts

Figures 7–10 show the prompts we use for images. The prompts for videos are almost identical, with "image" replaced by "video" in the text.

A.1 Independent Description

Figure 7 illustrates the prompt we use to generate independent descriptions, instructing the VLM to provide descriptions based on the image content.

A.2 Contextual Description

Figure 8 and Figure 9 present the prompts used for generating contextual descriptions. The prompt in Figure 8 instructs the VLM to describe images based on the contextual information, while the prompt in Figure 9 guides the VLM to analyze the function of images.

A.3 Structured Summary

Figure 10 illustrates the prompt designed for generating a structured summary. It instructs the VLM to produce a summary of the issue based on a referenced format.

B Example

Figure 11 presents an example where visual data is processed by the VLM, and the resulting information is appended to the original issue. Note that textual data, such as code snippets and metadata (e.g., library versions, environment details, etc.), are handled in the same way as regular text.

C Human Verification Process

Three annotators, each with at least four years of experience using Python, participate in the human verification process. Each annotator independently evaluates all 159 instances based on the three criteria outlined in Section 3.1, assigning a “yes” or “no” label to each criterion. Final labels for each instance are determined by majority vote.

We retain only those instances where the first criterion is labeled “no” (i.e., the visual data cannot be fully converted to text), and both the second and third criteria are labeled “yes” (i.e., the visual data is essential for resolution, and the problem description contains sufficient information). This filtering results in a final curated set of 133 high-quality task instances.

D Other Experimental Details

For models like Qwen2-VL and Qwen2.5-Coder-32B, we use vLLM for deployment on servers equipped with four NVIDIA H800 GPUs (each with 80GB of memory). For the DeepSeek-V2.5 model, we utilize the official API service provided by its developers. All experiments are conducted twice to determine the maximum number of instances that can be resolved. When using the Agentless approach, we employ version 1.0. Table 4 summarizes the hyperparameter details used in our experiments.

Phase	Component	Details
Data Processing	Independent Description	temperature = 0.2
	Contextual Description	temperature = 0.3
	Structured Summary	temperature = 1.0
Agentless 1.0	Localization Phase	temperature = 0.8. For each instance, 4 sets of locations are sampled, each with a max length of 3, and the sets are merged into 2 groups (first 2 and last 2).
	Repair Phase	Uses a context window of ±10 lines around each edit location. For each group, 21 patches are generated (1 greedy with temperature = 0, and 20 samples with temperature = 0.8).

Table 4: Hyperparameter settings for each phase and component of our experiments.

```
You are a technical image descriptor. For the given image:
1. If it contains only text, present the exact text in markdown format
2. If it contains visual elements:
- Describe the main technical content
- Include specific measurements, numbers, and text
- State the relationships between visual elements
- Focus on technical details over visual style

Your description should be detailed enough for an AI model to understand the technical content without seeing the image.
```

Figure 7: Prompt for generating independent descriptions of images.

You are a technical image descriptor for software issues. Your task is to create detailed descriptions of ALL images in the issue that will help other AI models understand the issue without seeing the actual images.
For EACH image in the issue:
1. Read and understand the entire issue context including:
- Bug description
- Code samples
- Error messages
- Expected behavior
- Actual results

2. Create a comprehensive description that:
- Details exactly what is shown in the image
- Connects the image content to the issue context
- Includes any visible technical information that's crucial for understanding the issue
- Provides enough detail that an AI model could understand the issue's visual aspects without seeing the image

Please provide your descriptions in this specific JSON format:

```
{
  "images": [
    {
      "image_id": "<sequential number>",
      "description": "<detailed technical description that fully captures the image content and its relationship to the issue>"
    }
  ]
}
```

CRITICAL: Ensure you describe EVERY image present in the issue - missing any image would make the issue harder to understand for AI models that cannot see the images.

Figure 8: Prompt for generating descriptions of images based on the contextual information.

You are a specialized technical image analyst for software issues. Your task is to analyze how each image connects to and supports the reported issue. Focus on providing a comprehensive analysis that explains the image's role and value in the issue context.

For each image, analyze:

1. Direct Issue Connection
   - How does this image specifically demonstrate or relate to the reported issue?
   - What aspects of the issue does this image capture or verify?
   - Why was including this image necessary for documenting this issue?

2. Technical Value
   - What key technical details does this image reveal about the issue?
   - How do specific elements in the image help understand the problem?
   - What insights does this image provide for troubleshooting or resolution?

3. Documentation Importance
   - What unique information does this image convey that text alone couldn't?
   - How does this image strengthen the overall issue documentation?
   - What critical details should developers focus on when reviewing this image?

Provide your analysis in this JSON format:

```
{
  "images": [
    {
      "image_id": "<sequential number>",
      "analysis": "<comprehensive analysis covering the image's connection to the issue, its technical value, and documentation importance. Focus on explaining why this image matters for understanding and resolving the specific issue at hand. Include relevant technical details and their significance to the issue context.>"
    }
  ]
}
```

Key Guidelines:
- Create a narrative that clearly connects the image to the issue context
- Focus on why this image is necessary for understanding the specific issue
- Include relevant technical details and their significance
- Explain how the image contributes to issue documentation and resolution
- Be thorough but concise in your analysis

Figure 9: Prompt for analyzing the function of images within the given issue.

You are an issue organizer and analyzer. The user will provide you with an issue that includes text descriptions and images. Your task is to analyze this information thoroughly and output a structured summary of the issue in JSON format.

The output should include relevant elements as applicable, but you are not required to fill in every field if the information is not available or cannot be accurately summarized. Aim to include:

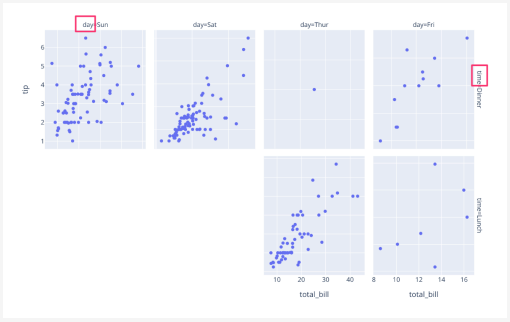
```
```json
{
  "problemSummary": "<a concise summary of the problem>",
  "context": "<any relevant background information>",
  "stepsToReproduce": [
    "<step 1: describe the action taken>",
    "<step 2: describe the next action>",
    "...<more steps as necessary>"
  ],
  "expectedResults": "<what the user expected to happen>",
  "actualResults": "<what actually happened>",
  "supplementaryImages": [
    "<descriptions of the images provided>"
  ],
  "additionalNotes": "<any other relevant information or notes>"
}
```

Feel free to omit any fields that are not applicable or where information is uncertain, while ensuring the output remains clear and informative to assist other models in understanding and resolving the issue effectively.

Figure 10: Prompt for generating a structured summary.

PX facet labels don't respect labels kwarg — Issue

```python
px.scatter(tips, x="total_bill", y="tip", facet_row="time",  facet_col="day",
        labels={"time": "THE TIME", "day": "THE DAY"})
```



Reported in https://github.com/plotly/plotly_express/pull/164

PX facet labels don't respect labels kwarg — New Issue

```python
px.scatter(tips, x="total_bill", y="tip", facet_row="time",  facet_col="day",
        labels={"time": "THE TIME", "day": "THE DAY"})
```

This image is part of the problem description. Here is the relevant information:
Image Details:

- **Image ID**: 1
- **Raw Description**:
The image consists of four scatter plots, each representing the relationship between the total bill and the tip amount for different days and times. The plots are arranged in...
- **Contextual Description**:
The image is a scatter plot with multiple facets, showing the relationship between 'total_bill' and 'tip' across different days and times. The x-axis is labeled 'total_bill' and ranges from 8 to 16...
- **Analysis**:
This image demonstrates a reported issue with the Plotly Express library where facet labels do not respect the labels keyword argument. The scatter plot is intended to show the relationship between 'total_bill' and 'tip' for different days and times, with facet labels set to 'THE TIME' and 'THE DAY'...

Reported in https://github.com/plotly/plotly_express/pull/164
Issue Summary (Structured)
- **problemSummary**: Facet labels in a scatter plot do not respect the labels keyword argument.
- **context**: The issue was reported in a GitHub pull request.
- **stepsToReproduce**: ['Create a scatter plot using Plotly Express.', 'Use the `facet_row` and `facet_col` parameters to create facets.', 'Use the `labels` parameter to specify custom labels for the facets.']
- **expectedResults**: The facet labels should reflect the custom labels provided in the `labels` parameter.
- **actualResults**: The facet labels do not change and display the default labels.
- **supplementaryImages**: ['A scatter plot with facets showing the default labels instead of the custom labels.']
- **additionalNotes**: The issue was reported in a GitHub pull request, indicating it is a known bug.

Figure 11: An example of a processed visual issue. The issue from Plotly issue #1944.
```