Multimodal AI Code Assistants for the Next Generation of Software Development

Luís Filipe Fernandes Gomes

CMU-S3D-26-112

August 2026

Software and Societal Systems Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Jonathan Aldrich, Carnegie Mellon University (Co-Chair)
Rui Abreu, University of Porto & META (Co-Chair)
Vincent Hellendoorn, Carnegie Mellon University & Google Deepmind (Co-Chair)
Bogdan Vasilescu, Carnegie Mellon University
David Lo, Singapore Management University
João Saraiva, University of Minho

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Software Engineering.

Copyright © 2026 **Luís Filipe Fernandes Gomes**

Support for this research was provided by Fundação para a Ciência e Tecnologia through the Carnegie Mellon Portugal Program under Grant PRT/BD/152343/2021.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution or any other entity.

Keywords: AI code assistants, multimodal human–AI interaction, sketch-based code generation, visual documentation, program comprehension, code review, software visualization

To my family.

Abstract

Software development is an inherently multimodal activity. Developers reason about systems through mental models, and two decades of empirical research show that they externalize those models visually: as whiteboard drawings, napkin diagrams, and informal sketches that mix boxes, arrows, and annotations at a deliberately high level of abstraction. These artifacts are not decoration around the real work; they are the primary medium through which developers build and share understanding of the systems they create.

Modern AI code assistants ignore this practice almost entirely. Built around Large Language Models (LLMs), they consume and produce text: developers must compress visual reasoning into linear prompts, and receive in return large volumes of generated code with no visual handle for inspecting it. Even models that accept images treat them as auxiliary context rather than as carriers of intent, discarding the structure, ordering, and dependencies that a sketch encodes. The consequences are well documented: developers struggle to express design intent to these tools, over-trust generated code that looks plausible, and miss subtle flaws in output they did not write. As AI assistants shift developers from authors to reviewers of code, this mismatch between how developers think and how their tools communicate becomes increasingly costly.

This dissertation demonstrates that AI code assistants can interpret, produce, and maintain the informal visual artifacts developers already use, and that doing so measurably improves how developers express intent to and verify the output of these systems. We focus on three processes of AI-assisted development where the absence of a visual channel is most costly, and contribute systems and evaluation methodologies for each.

(1) Expressing intent to a code generator through text alone forces developers to abandon the sketches in which they naturally plan. To address this, we contribute the first *Visual Code Assistant*, an in-IDE tool that treats informal, hand-drawn sketches of machine learning workflows as first-class specifications and converts them into executable Jupyter notebooks. A user study with 19 data scientists produced a dataset of paired sketches and notebooks, an empirical characterization of how developers sketch, and an LLM-as-a-Judge evaluation methodology, validated against human grading, that shows such sketches are viable specifications for code generation.

(2) Understanding code, especially AI-generated code, is hard, and existing tools explain code with yet more text. To address this, we build VisDocSketcher, an agentic system that generates informal, developer-style diagrams directly from source code, together with AutoSketchEval, a reference-free framework that evaluates a diagram by reconstructing code from it, sidestepping the fact that no canonical reference diagram exists. We then establish that such diagrams actually help: in a controlled within-subjects study with 26 developers, JupyterDraw, a tool that renders whiteboard-style overviews of Jupyter notebooks, raised complete detection of silent methodological flaws in AI-generated notebooks from 29% to 67% of review sessions, at no cost in time or workload, and reviewers credited the diagram as their primary detection route five times as often as an equivalent textual summary.

(3) Software does not hold still, so any generated diagram is stale documentation in the making. To address this, we contribute Visual Loop, a framework in which sketch and code co-evolve: it maintains fine-grained correspondences between visual regions and code segments, so that an edit to either artifact propagates locally to the affected parts of the other rather than forcing wholesale regeneration.

The goal of my dissertation is to establish informal visual artifacts as first-class citizens of human–AI collaboration in software engineering. Beyond the systems themselves, my work contributes datasets of developer sketches, evaluation machinery for artifacts that admit no single ground truth, including the AutoSketchEval round-trip metric, and controlled human-subjects evidence that visual overviews change what reviewers can catch. Together, these contributions chart a path toward AI tools that participate in the same multimodal, informal collaboration practices that already structure how humans build software with each other.

Acknowledgments

To Rui Maranhão, who was the first to show me that a PhD could actually be a possibility for me: thank you for encouraging me to apply to CMU, and for believing in me and wanting to work with me from the very start. That first spark of confidence is the reason any of what follows exists.

To Jonathan Aldrich for always being available and open to discuss any topic. Thank you, above all, for your kindness even when the results were not what we hoped.

To Vincent Hellendoorn, I admire you deeply as a professional, and I am endlessly grateful for the expertise and knowledge you brought to my path. Thank you for the many hours you spent coding and writing papers alongside me. A PhD is a lonely journey, and your help carried me through some of its most critical moments.

To Bogdan Vasilescu, thank you for teaching me most of what I used throughout this research; a great deal of this work carries your fingerprints. To Christian Kästner, thank you for your feedback throughout the SSSG talks, for teaching me about ML in production, and for teaching me how to teach, giving me the chance to work alongside you as a TA.

To David Lo, thank you for receiving me in Singapore and for your guidance during my time there.

To Eduardo and Mariana Miranda, you have been like family to me all these years. You were my first contact in Pittsburgh: I arrived at night, not knowing a single person in the country, with only a phone number, no data on my phone, and a photograph to go by. Thank you for letting me stay at your place without knowing me, and thank you for all the help to set everything up during such difficult times. I will never forget that.

Thank you to Connie Herold and Alisha Roudebush. Thank you for your help navigating the administrative hurdles behind the scenes.

To every person I met during these years, especially my cohort at S3D (ISR, at the time I joined). Thank you for all the coffees, the walks at Schenley Park, the wine nights, the dinners, the movies, the concerts, the skiing, the nights at 5801, Mario's and Cavo, the ping pong, tennis, the soccer competitions with Shady Squirrels, the spring breaks and all the travels.

And behind the scenes, thank you to Siti Nurhaliza, the one who has stood by my side through the final and hardest stretch of this journey. Thank you for your patience, your love, and for making the ending of this chapter so much brighter.

To my high school, especially the twins João and Bruno Ribeiro, and Rúben Costa, and also my closest undergrad friends, Rui Calheno, João Alves, and Elisa Valente, who have supported me all along, even from thousands of miles away. We still make time for calls, for visits whenever we can, and for the trips that bring us back together no matter how scattered we've become. Thank you for staying close despite the distance.

And to those who shared a chapter of this journey with me and are no longer part of the story, thank you for what you taught me about myself along the way.

Above all, to my family who call every day to check on me and who have always been supporting me from the other side of the ocean.

Contents

1 Introduction **1**
- 1.1 Challenges in Multimodal AI-Assisted Software Development 2
- 1.2 Thesis Statement 8
- 1.3 Document Outline 9

2 Background **11**
- 2.1 Mental Models and Sketching in Software Engineering 11
- 2.2 Code Generation 12
- 2.3 Code Explanation 12
- 2.4 Code Evolution 13
- 2.5 Artifact Evaluation 14

3 Code Generation **15**
- 3.1 Related Work 16
  - 3.1.1 Sketching in SE 16
  - 3.1.2 Images to Code 17
- 3.2 Visual Coding Assistant 18
  - 3.2.1 Motivating Example 19
- 3.3 Methodology 20
  - 3.3.1 Recruitment 20
  - 3.3.2 Protocol 21
  - 3.3.3 Tasks 22
  - 3.3.4 Data and Artifacts 23
  - 3.3.5 Evaluation 23
- 3.4 Results 26
  - 3.4.1 How Do Data Scientists Sketch? 26
  - 3.4.2 Sketch to Code Generation 29
  - 3.4.3 Perspectives on Visual Code Assistants 30
  - 3.4.4 Threats to Validity 32
- 3.5 Summary 33
- 3.6 Takeways 33

4 Code Explanation: Diagram Alignment **35**
4.1 Related Work . . . 37
4.1.1 Visuals in SE and Code Assistants . . . 38
4.1.2 Generating Code Explanations . . . 38
4.1.3 Notebook Tooling and Visualization . . . 39
4.2 Automatic Visual Documentation Generator . . . 40
4.2.1 Single Agent Setup . . . 41
4.2.2 Multi Agent Setup . . . 42
4.2.3 Evaluation Framework . . . 44
4.3 Evaluation Design . . . 46
4.3.1 Experimental Setup . . . 48
4.4 Results . . . 48
4.5 Discussion . . . 55
4.6 Threats to Validity . . . 56
4.7 Summary . . . 57
4.8 Takeaways . . . 57

5 Code Explanation: Diagram Usefulness **59**
5.1 Contributions . . . 60
5.2 Related Work . . . 61
5.3 JupyterDraw: AI-Generated Diagrams . . . 63
5.3.1 Motivating Example . . . 64
5.4 Methodology . . . 66
5.4.1 Recruitment . . . 66
5.4.2 Protocol . . . 66
5.4.3 Tasks . . . 67
5.4.4 Evaluation . . . 68
5.4.5 Data and Artifacts . . . 69
5.5 Results . . . 70
5.5.1 RQ1: Detection . . . 70
5.5.2 RQ2: Mechanism . . . 72
5.5.3 RQ3: Perspectives . . . 74
5.6 Threats to Validity . . . 76
5.7 Summary . . . 78

6 Code Evolution: A Preliminary Study **79**
6.1 Visual Loop: The Vision . . . 80
6.1.1 Motivating Examples . . . 80
6.1.2 Prototype . . . 82
6.2 Proof-of-Concept: Weather App Demonstration . . . 84
6.3 Discussion . . . 85
6.4 Summary . . . 85
6.5 Takeaways . . . 85

7 Summary of Contributions **89**

8 Discussion **91**

9 Future Work **95**

10 Conclusion **99**

Bibliography **101**

List of Figures

1.1 Human developers externalize intent and negotiate meaning through multimodal artifacts. This thesis asks how AI systems can participate in the same loop. . . . 2

1.2 The multimodal human–AI collaboration loop: (a) from mental models to code (P1); (b) from code to mental models (P2). 3

3.1 Example of usage of our prototype. Left editor tab: The whiteboard sketch uploaded by the participant to be used by the code assistant to generate code. Right editor tab: Generated Jupyter Notebook with code and Markdown explanations. . 19

3.2 Overview of our benchmarking using LLM-as-a-Judge. To evaluate a Jupyter Notebook generated from a sketch (1), the Judge has access to knowledge elements related to the original drawing: a Task description that relates to users' coding intentions depicted on the drawing (2), and the grading criteria for the Outline and Instantiation metrics (3 and 4). 24

3.3 Performance comparison of top visual LLMs on the 19 notebooks. No statistically significant difference was detected in either dimension in our study. 31

4.1 Example of a sketch generated by VisDocSketcher. 41

4.2 Overview of the multi-agent architecture. A supervisor agent receives the user query and orchestrates the workflow by delegating tasks to specialized agents. Each agent communicates exclusively with the supervisor (no inter-agent communication). Agents are equipped with tools to access and manipulate the local filesystem. 43

4.3 **Experimental Setup.** 1) *Human Analogy*: The similarity between the code produced by developer B and the original code depends on how informative the diagram is. 2) *Autoencoder Analogy*: Reconstruction similarity (round-trip correctness) indicates the quality of the latent representation. 3) *Single-Agent Evaluation*: Using a tested Sketch2Code model, semantic similarity reflects the effectiveness of VisDocSketcherin generating code-aligned sketches. 4) *Multi-Agent Evaluation*: Using the same setup, we compare single and multi-agent architectures. 44

4.4 AutoSketchEval: Framework validation results based on expert-authored ground-truth visuals. *Aligned*: Samples where the diagram and code corresponds to the same task. *Unaligned*: Samples where the diagram and code correspond to different tasks. 49

4.5 Distribution of Pairwise Differences Across Metrics. All metrics show a slight rightward shift (> 0), reflecting consistent but modest improvements of Multi-Agent over Single-Agent architectures. 53

5.1 Example of a Jupyter notebook overview generated using JupyterDraw. These are the overviews shown to participants in the visual (V) condition. 61
5.2 Example of the textual condition (T) overview. 65
5.3 Flaws detected per task by condition ($n = 24$). Left: individual paired observations with bootstrapped condition means; right: the within-participant difference (V − T) with its bootstrap 95% CI. 71
5.4 Standardized effect (Cohen's d, V − T) of the visual condition across every regression. 73
5.5 Distribution of the per-participant difference in perceived helpfulness. 74
5.6 Attribution of caught flaws to each overview (V, T) by criterion, among the flaws participants detected (Fisher exact, $^{*}p < 0.1$, $^{***}p < 0.01$). 75

6.1 Visual Loop: a continuous visual development environment where code and sketches co-evolve through AI synchronization. Our prototype connects the IDE with a tablet interface, enabling developers to visualize, navigate, and modify systems through informal sketches: (1) desktop *IDE plugin* with static analysis and visualization hooks, (2) a tablet-based *Visualization Engine* for sketching and interaction, and (3) an *AI Synchronization Layer* powered by foundation models that maintain consistency between code and visuals in real time. 80
6.2 The two Visual Loop surfaces: (a) the vscode extension panel; (b) the tablet annotation canvas. 83
6.3 Proof-of-Concept Example: exploring code and rearranging diagrams to sketch changes, interpreting them through AI, and synchronizing updates between code and visuals. 86

List of Tables

3.1 Tasks and Subtasks 22
3.2 Usage of element by subtask. Textual annotations are the most commonly used element in all tasks. 28
3.3 Summary of regression models predicting code generation outcomes based on sketch time, last sketching activity, and programming experience. Reported p-Values are after Benjamini-Hochberg correction. 29

4.1 AutoSketchEval – Separation Between Aligned and Unaligned Samples. CodeBERTScore's F1 and Precision achieve perfect discriminative power (AUC = 1.000, Cliff's Δ = 1.00), fully separating aligned from unaligned code-sketch pairs, while CodeBLEU-based metrics show strong but incomplete separation. . 50
4.2 Robustness analysis: pairwise comparison of LLM model combinations, including same-model pairs and cross-model examples. The *Significant* column indicates whether differences in the distributions between the two setups results are statistically significant. Full results and analysis are available in the replication package. 50
4.3 Comparison of baseline and VisDocSketcher performance, reporting the 5th percentile and average values. F1 and precision denote CodeBERTScore components (embedding similarity between original and reconstructed code), not classification metrics. 52
4.4 Regression Results: Dataflow and Notebook Complexity. 54

5.1 Planted methodological flaws per notebook. 68
5.2 Linear mixed model predicting the number of detected flaws (0–2) per task, with a random intercept per participant (n = 24). The visual condition increases detection net of notebook and task order. 72

Chapter 1

Introduction

Software development is an inherently collaborative activity. Developers continuously communicate ideas, intentions, and mental models with others to design, implement, and evolve software systems. Critically, this communication is *multimodal* as shown in Figure 1.1: developers speak, draw diagrams, sketch flows on whiteboards, and walk through code interactively. Visual and verbal channels are not supplementary to software work, they are central to it, particularly when reasoning about architectures, workflows, and design trade-offs.

Modern AI code assistants, by contrast, remain fundamentally text-centric. Even when they accept image inputs or produce text-based explanations, they treat visual communication as secondary, implicitly assuming that developer intent must be specified and conveyed through language. This design choice forces developers to translate rich, informal mental models into linear textual descriptions, stripping away the spatial structure, emphasis, and intentionality that visual representations carry. As a result, AI systems struggle to capture what developers *mean*, rather than what they can explicitly encode in text.

While multimodal code generation has begun to explore visual inputs (predominantly for frontend tasks such as generating UI code from mockups), broader, visually rich activities remain underexplored. These include reasoning from informal system architectures and workflow sketches, generating visual explanations that match how developers actually think, and co-evolving code and visual artifacts incrementally over time. Existing approaches to code explanation focus on formal representations such as UML, overlooking the informal, sketch-like visualizations that developers naturally produce. More fundamentally, no current framework treats informal visual artifacts as first-class, co-evolving representations within a sustained human–AI collaboration loop.

The absence of AI systems that can engage with developers through visual, informal, and evolving representations creates a persistent gap between human mental models and AI reasoning. A gap that is not merely a usability inconvenience, but a conceptual limitation of current AI-assisted development paradigms. Closing this gap has practical consequences: it enables AI tools to participate in the kinds of shared understanding that happen at whiteboards and design reviews, not just in chat prompts. Without addressing it, AI systems will remain transactional code generators rather than genuine cognitive collaborators.

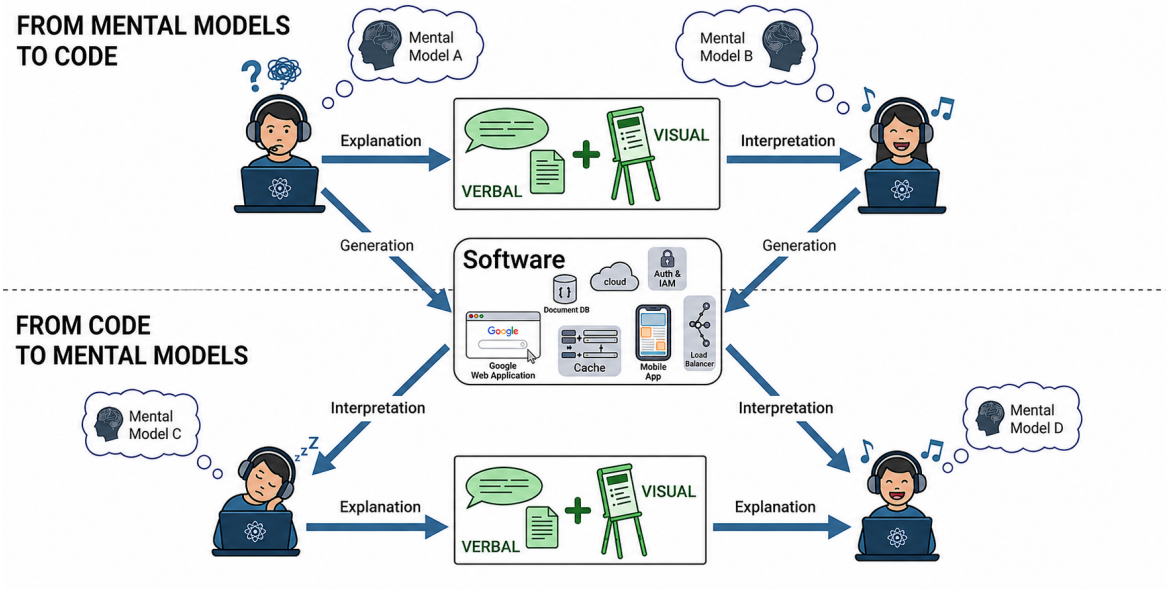


Figure 1.1: Human developers externalize intent and negotiate meaning through multimodal artifacts. This thesis asks how AI systems can participate in the same loop.

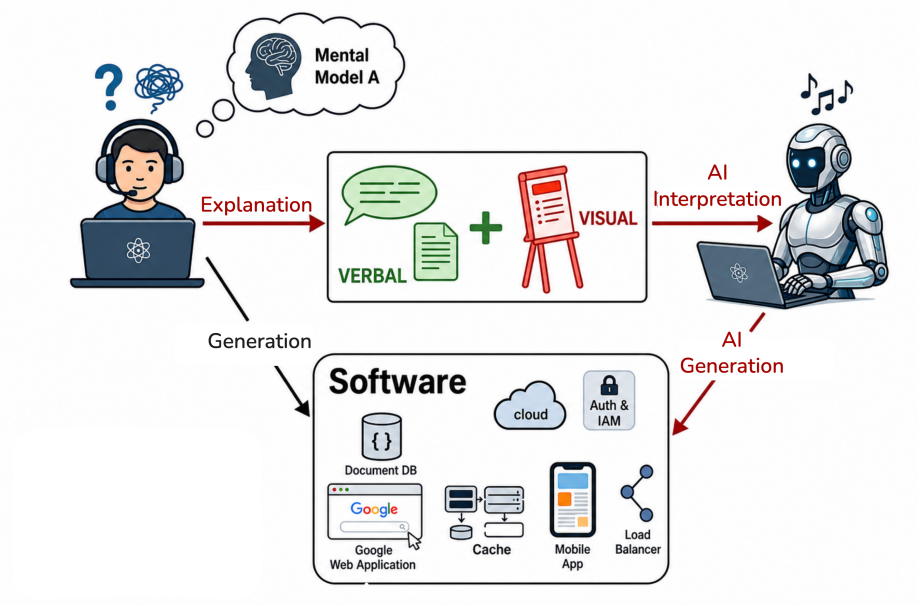
1.1 Challenges in Multimodal AI-Assisted Software Development

In Figure 1.1, we see the core interactions between human developers, their programming peers, and code. When introducing LLM agents into software development, the goal is for them to function as programming peers, preserving these core interaction patterns as closely as possible, as they reflect interaction structures that have evolved to effectively support mental model alignment.

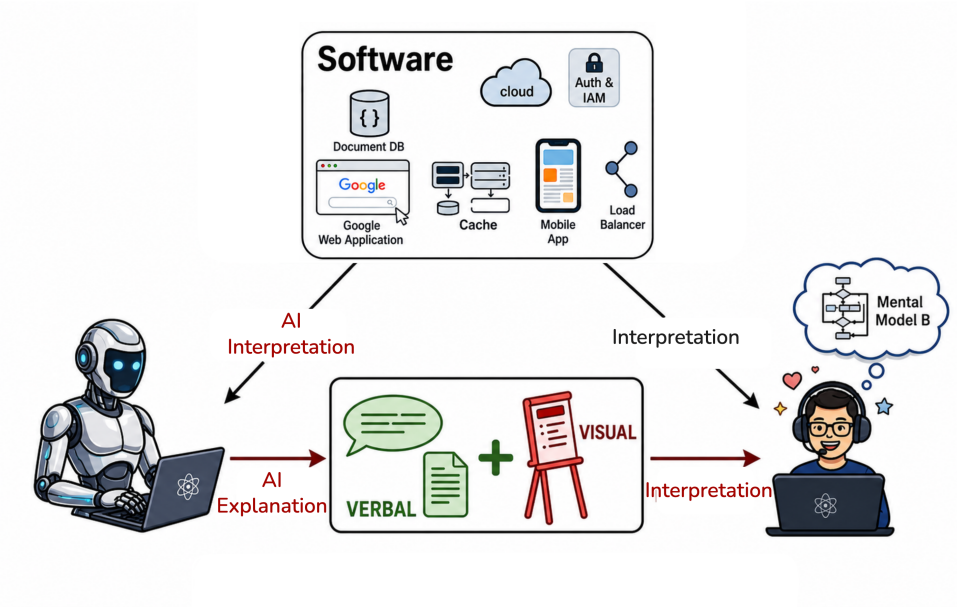
Figure 1.2 shows the two directions of that exchange. My thesis identifies three core challenges that together form a closed human-AI interaction loop, plus an orthogonal challenge of evaluating multimodal artifacts. Process P1 captures how AI should interpret informal human representations to generate code; P2 captures how AI should explain code back to humans through their natural communication modalities. Process P3 closes the loop: as code and visual artifacts evolve, changes in one must propagate selectively to the other, sustaining a shared, up-to-date representation of the system over time. The first two directions, P1 and P2, are sustained over time, so that as code and visual artifacts evolve, changes in one propagate selectively to the other rather than forcing either artifact to be fully regenerated from scratch. Together, these three processes move the interaction model from isolated, text-only exchanges toward a continuous multimodal collaboration between developer and AI.

Code Generation

Challenge. AI code assistants are not effectively being used to interpret informal visual representations of software systems as intent specifications. When developers sketch a workflow or architecture diagram, they encode structural intent, implied behavior, and design priorities that



(a) Process P1 – from mental models to code.



(b) Process P2 – from code to mental models.

Figure 1.2: The multimodal human–AI collaboration loop in software development. **(a)** Process P1: a developer externalizes an intended design (Mental Model A) through verbal and visual artifacts, which an AI agent interprets to generate code. **(b)** Process P2: the direction inverts, the agent explaining code back through the same two channels so the developer can reconstruct it (Mental Model B). When run iteratively, with region-to-code correspondences preserved as both artifacts change, they constitute P3 (Chapter 6).

cannot be trivially recovered from the visual surface. Current code generation pipelines either ignore this visual channel entirely or treat images as secondary context, effectively discarding the information that makes sketches valuable. This forces developers to re-articulate in text what they have already expressed visually, introducing loss and friction.

Hypothesis. Assuming that informal visual artifacts encode the developer's intended mental model. Our central claim is that, given a human-produced visual artifact, an AI system can construct code that preserves the intended semantics embedded in the artifact.

Rather than eliminating ambiguity, an effective system should *operationalize* it, reasoning about what the sketch implies and making principled decisions where the visual representation underspecifies behavior.

Solution. To study this challenge, we introduce the concept of *Visual Code Assistants* [39], a framework and prototype that takes informal visual representations (sketches, hand-drawn diagrams) as primary inputs for code generation. The system uses multimodal LLMs to interpret visual structure, resolve ambiguities, and produce code that reflects the developer's drawn intent.

Takeaway. Results demonstrate that Multimodal LLMs generate code outlines with 70–80% accuracy from whiteboard sketches alone, with each additional minute of sketching yielding a 2.5 percentage point improvement, confirming that the visual modality is an independent, information-bearing channel. Sketching also reduced lines of code written by 49% on average, with instantiation accuracy reaching 25–40% despite sketches frequently omitting low-level implementation details. 47.4% of participants revisited their sketches during coding without instruction, treating them as persistent references rather than one-shot prompts. This naturally occurring behavior directly motivates the co-evolution challenge: developers already maintain a live connection between sketches and code and it is important to know how to sustain it automatically. Evaluation methodology and findings are discussed in Chapter 3.

Publications.
Transforming Ideas into Code: Visual Sketching for ML Development [38]
An Exploratory Study of ML Sketches and Visual Code Assistants [39]

Code Explanation

Challenge. AI systems predominantly produce textual explanations of code. Yet developers often construct understanding through visual abstractions, structural simplifications, and informal diagrams. Even though AI assistants possess such capabilities, they are often not explicitly surfaced in the interface in the same way textual prompts are, since chatbot-style interaction remains the primary interface for coding agents. As a result, developers are less likely to seek these capabilities and visuals are rarely treated as first-class components of the generated documentation. This challenge decomposes into two sides of the same coin, which this thesis studies in separate chapters: (1) *alignment* – does the generated visualization faithfully represent the underlying code? – and (2) *usefulness* – does it actually help a developer build an appropriate mental model when working with unfamiliar code? A diagram can be perfectly faithful yet unhelpful, or persuasive

yet wrong; only an explanation that satisfies both properties earns a place in the development workflow. Existing work on code visualization focuses on formal notations (e.g., UML), which do not reflect the informal style in which developers naturally reason, and rarely measures either property directly.

Hypothesis. AI systems can externalize structural and behavioral code reasoning through informal diagrams, generating visual explanations that mostly align with the code and match the representations developers would produce. If that alignment holds, then surfacing these diagrams during real developer tasks, such as reviewing code they did not write, should measurably improve comprehension outcomes relative to purely textual explanations.

Solution. The alignment side is studied through *VisDocSketcher*, a system that takes source code as input and generates informal visual explanations. The system reasons about which structural and behavioral aspects of the code are most salient, and renders them in a sketch-like visual format using Mermaid syntax. We conducted a comparative analysis of single- and multi-agent architectures, and analyzed how code size and structural complexity affect the quality of the generated visuals. The usefulness side is studied through JupyterDraw, a tool that draws whiteboard-style overviews of Jupyter Notebooks on an infinite canvas. Using it, we ran a controlled within-subjects experiment (Latin-square counterbalanced, 26 developers from academia and industry) in which participants reviewed AI-generated notebooks containing planted silent methodological flaws. The code runs and looks plausible but yields misleading conclusions. In one of the sessions participants are supported by a JupyterDraw visual overview (V) and in another by an equivalent textual summary (T).

Takeaway. Findings indicate that the multi-agent system outperforms the static-analysis-based baseline by 26.7-39.8% across code-level metrics, and AutoSketchEval reliably distinguishes code-aligned from non-aligned sketches (AUC $\geq$ 0.87), establishing that sketch-code alignment is a measurable property even in the absence of ground-truth visual representations. The multi-agent variant outperforms the single-agent in 59.3% of cases, though at 11.3 $\times$ the processing cost. Sketch faithfulness degrades with code complexity: every 100 additional lines of code yields an 8% drop in dataflow score, revealing that full-regeneration approaches do not scale. This finding directly motivates the incremental co-evolution mechanism in the Code Evolution challenge. The user study then confirms that aligned visuals translate into measurable comprehension gains: with the visual overview, the proportion of review sessions in which participants caught *both* planted flaws rose from 29% to 67%, with no increase in review time or perceived workload. Participants rated the visual overview as more helpful and more accurate than the textual one, credited it as their primary detection route far more often than the text (39% vs. 8% of caught flaws), and 75–80% preferred it for future reviews. By enabling AI to communicate through human-aligned visual representations, this component supports the *externalization* direction of the interaction loop—AI-to-human. Full results are discussed in Chapters 4 and 5.

Publications.

VisDocSketcher: Towards Scalable Visual Documentation with Agentic Systems [41]

JupyterDraw: An Empirical Study of Visual Overviews for Reviewing AI-Generated Notebooks

Code Evolution

Challenge. Closing the interaction loop requires that code and visual artifacts not merely be translatable into one another, but that they *co-evolve*. It might be tempting to assume that this follows automatically once bidirectional translation between code and sketches is available; in practice, however, collaboration demands incremental, localized updates. Developers do not regenerate an entire codebase or redraw an entire diagram after every small change. Instead, edits in one artifact should propagate selectively to the relevant parts of the other. This is analogous to how a request to "Change function `foo`" should not affect an unrelated function `bar`. Code assistants evolved to integrate the whole file or codebase as context, but only modify that specific function. Similarly, we want a visual assistant to understand a sketch as a whole, but modify only some parts of it according to the changes in the code and vice-versa. Because informal visualizations intentionally omit detail, propagating changes from sketch to code and back is non-trivial: information is lost in the abstraction. A multimodal assistant must reason about diffs in both artifacts, interpret code changes, communicate them visually to the developer, and symmetrically interpret visual edits and propagate them back to code.

Hypothesis. Given correspondences between regions of a visual abstraction and segments of code, an AI system can propagate incremental changes between artifacts selectively and correctly, preserving the integrity of unaffected regions in both the code and the visualization. This type of interaction is beneficial for developers, improving their navigation and understanding of a codebase.

Solution.

This challenge is addressed by *Visual Loop* [33], a design for *human–AI co-development* in which informal visual representations serve as first-class collaboration artifacts, together with a proof-of-concept prototype. The design establishes region-to-code correspondences intended to enable change propagation in both directions; the prototype demonstrates the sketch-to-code direction, and the propagation mechanism itself remains open for future work.

Takeaways. Successfully addressing this challenge closes the human-AI interaction loop: AI systems will be able to not only generate code from sketches and explain code as sketches, but also sustain shared understanding as both artifacts evolve over time. This is the most exploratory component of the thesis and the one with the highest potential to redefine how AI participates in software co-development. Preliminary design and evaluation plans are discussed in Chapter 6.

Publications.

Visual Loop: Bridging the Cognitive Gap in Software Development Through Visual-AI Collaboration [33]

Supporting Challenge: Artifact Evaluation

The three challenges described above all require measuring the quality of AI-generated artifacts. This is not a minor implementation detail, it is a hard problem that has required dedicated methodological work that runs orthogonally through all three challenges. Unlike the generation, explanation, and evolution components, artifact evaluation does not occupy a single stage of the

human–AI interaction loop; it is instead how the claims about each of those stages are empirically validated. This subsection describes evaluation methodologies developed across three first-author contributions, where evaluation is a secondary but necessary component. It also draws on *SE-Jury*, a co-authored study in which evaluation is the primary focus.

Challenge. Evaluating the quality of AI-generated software artifacts is significantly harder when those artifacts include visual representations. For code alone, evaluation is already non-trivial: traditional match-based metrics such as BLEU and CodeBLEU measure surface similarity rather than semantic correctness. For visual artifacts, the challenge compounds: the same codebase admits many plausible visualizations, and similar sketches can reasonably correspond to several implementations when the context is changed. There is no canonical ground truth against which to score a diagram. Human evaluation remains the most accurate method, but it is labor-intensive, subjective, and does not scale. The absence of reliable, automatic evaluation mechanisms creates a bottleneck for the research methodologies: without principled ways to assess artifact quality, claims about generation, explanation, and evolution cannot be empirically grounded.

Hypothesis. Two complementary ideas address this gap. First, LLMs can be used as evaluators, leveraging their code comprehension and reasoning abilities to assess correctness in a flexible, scalable way, without requiring ground-truth test cases or hand-crafted metrics. Second, the alignment between a visual representation and its underlying code can be assessed indirectly through a *round-trip* procedure: if a sketch faithfully encodes the software structure and semantics,we are able to recover code that resembles the original. The fidelity of this reconstruction serves as a proxy for sketch quality, requiring no manually authored reference diagrams.

Solution. This thesis addresses artifact evaluation through three core contributions, supplemented by a fourth, co-authored work exploring the same LLM-as-a-judge approach.

LLM-as-a-Judge in Visual Code Assistants. Evaluating code generated from informal sketches is particularly challenging: participants in a user study may use any library, any coding style, and any level of detail, making execution-based metrics and manually designed test suites impractical. In *Visual Code Assistants* [39], we address this by using GPT-4 as an LLM judge that scores generated Jupyter Notebooks against a predefined rubric of task-specific criteria, mimicking how a programming instructor would grade a coding assignment.

SE-Jury: LLM-as-Ensemble-Judge (co-authored work). Building on the same LLM-as-a-Judge concept, I contributed to *SE-Jury* [152]. The core limitation of a single judge with a fixed prompting strategy is that it brings only one evaluative perspective. That work addresses this by defining five distinct evaluation strategies: direct assessment, self-reflection and rethink, equivalence comparison against a reference, criteria-based key-property analysis, and test-generation-then-assessment. Then dynamically assembling the most effective subset for each dataset via lightweight calibration on 20 annotated samples. Evaluated across four human-annotated benchmarks spanning code generation, automated program repair, and code summarization in five programming languages, SE-Jury outperforms all baselines by 29.6%–140.8% in correlation with human judgment and approaches inter-annotator agreement in code generation and program repair.

AutoSketchEval: Round-Trip Accuracy for Visual Documentation. Evaluating generated sketches without ground-truth diagrams is a fundamental challenge for the code explanation

work. In *VisDocSketcher* [41], we present *AutoSketchEval*, a framework inspired by the autoencoder concept. A Code2Sketch system (encoder) generates a visual representation from code; an independent, validated Sketch2Code system (decoder) then reconstructs code from that sketch alone. The similarity of the reconstructed code, measured using CodeBLEU and CodeBERTScore against the original, serves as a proxy for sketch fidelity. Validated against human-authored ground-truth sketches, AutoSketchEval achieves an AUC above 0.87, confirming strong discriminative power between high-quality (code-aligned) and low-quality (non-aligned) sketches without requiring any manually crafted reference diagrams.

Task-grounded human evaluation in JupyterDraw. Automatic proxies can establish that an artifact is faithful to the code, but not that it changes what a developer can do with it. For claims about usefulness, the evaluation instrument must be the developer themself. In JupyterDraw, we designed a controlled within-subjects protocol in which 26 developers reviewed AI-generated notebooks seeded with silent methodological flaws, alternating between visual (V) and textual (T) overviews under a Latin-square counterbalanced design. The protocol triangulates three kinds of evidence: behavioral outcomes (which flaws were caught, and through which representation), standardized instruments (NASA-TLX workload, Likert-scale helpfulness and accuracy ratings), and qualitative preference data. This complements the automatic frameworks above by grounding the thesis's usefulness claims in observed developer behavior rather than proxy metrics.

Takeaways. Standard automatic metrics are insufficient for assessing the alignment between code and informal visual representations. LLM-based evaluation, when carefully designed with multiple complementary strategies, can approach human-level agreement for code artifacts. Round-trip reconstruction provides a principled, reference-free alternative for evaluating visual artifacts. And for the questions no proxy can answer, whether an artifact actually helps, task-grounded human-subjects protocols close the loop. These approaches are embedded in the empirical methodology of the three main challenges and collectively support the broader claim that progress in multimodal human–AI collaboration can be measured, despite the subjectivity in some artifacts. **Publications.**

An Exploratory Study of ML Sketches and Visual Code Assistants [39].

VisDocSketcher: Towards Scalable Visual Documentation with Agentic Systems [41].

JupyterDraw: An Empirical Study of Visual Overviews for Reviewing AI-Generated Notebooks.

(*Co-authored*) SE-Jury: An LLM-as-Ensemble-Judge Metric for Narrowing the Gap with Human Evaluation in SE [152].

1.2 Thesis Statement

Current AI systems for software engineering operate through text-centric interfaces that fail to reflect how developers construct and align mental models. The work presented in this thesis is organized around the following claim:

Thesis Statement

By embedding AI systems for software engineering within multimodal, iterative interaction loops that mirror human–human collaboration, we can improve mental model alignment, thereby transforming AI tools from transactional code generators into cognitive collaborators.

The three challenges described in Section 1.1 are designed to collectively evaluate this claim. Code generation (Chapter 3) tests whether AI systems can *interpret* informal visual intent. Code explanation (Chapters 4 and 5) tests whether AI systems can *externalize* reasoning in developer-aligned visual forms. Code evolution (Chapter 6) tests whether AI systems can *sustain alignment* through iterative, incremental co-development. Each component moves the interaction model away from text-only, request–response exchanges and toward a richer, multimodal collaboration loop.

The artifact evaluation work described in Section 1.1 provides the methodological foundation for measuring progress on all three challenges, and contributes independently to the open problem of evaluating informal, interpretation-dependent AI outputs in software engineering.

Taken together, these contributions propose a concrete path toward closing the gap between how developers naturally communicate and how current AI tools expect them to communicate.

1.3 Document Outline

The remainder of this document is structured as follows. Chapter 2 situates the work in prior research on mental models, visual code artifacts, bidirectional synchronization, and evaluation methodologies for software engineering. Chapter 3 addresses Process 1 (Code Generation), reporting the Visual Code Assistants study on sketch-to-code generation and its LLM-as-a-Judge evaluation. Process 2 (Code Explanation) spans two chapters, because its two sides must be studied separately: a diagram must first be shown to faithfully represent the code, and independently shown to help the developers who read it. Chapter 4 addresses the first side with VisDocSketcher and AutoSketchEval, which generate visual documentation from code and measure its alignment without ground-truth diagrams; Chapter 5 addresses the second with JupyterDraw, a controlled user study of whether such diagrams improve developers' review of AI-generated code. Chapter 6 addresses Process 3 (Code Evolution), introducing the Visual Loop co-evolution framework for synchronized code and informal visual artifacts. Chapter 7 consolidates the contributions, and Chapters 8–10 discuss broader implications, future work, and concluding remarks.

Chapter 2

Background

Building software is an exercise in understanding: developers form and communicate mental models of what a system does and how it should change. This chapter reviews the research that motivates treating those mental models, and the informal visual artifacts that carry them, as first-class elements of AI-assisted development. Section 2.1 summarizes what is known about how developers externalize mental models through sketches, the empirical premise on which the rest of the thesis stands. Sections 2.2 through 2.4 then review the state of the art for each process of the interaction loop introduced in Chapter 1, and Section 2.5 reviews how AI-generated artifacts are evaluated, the methodological question underlying all three.

2.1 Mental Models and Sketching in Software Engineering

Two decades of empirical research converge on a consistent account of how developers manage understanding. LaToza *et al.* surveyed 280 engineers and found that mental models of code are expensive to build and maintain. These models however, live almost exclusively in developers' heads or in transient whiteboard sketches; they explicitly call for "tools that capture informal whiteboard or paper designs" [64]. Field studies of sketching practice filled in the details. Developers' drawings are usually temporary and lost, because transcribing them into durable electronic form costs more than it returns [19]. When developers do sketch, they shift focus rapidly, keep detail deliberately low, and mix notations freely [75]; they improvise visual representations for each problem at hand, often using words themselves as visual objects rather than mere labels [120, 121]. Most sketches are informal, made on analog media, and pitched at a higher level of abstraction than the code they describe. Developers value them precisely as aids for understanding their code [6]. Even among professional software architects, who do routinely photograph and archive their whiteboards, the sketches capture the big picture while leaving the eventual solution under-specified [108].

This preference for drawing is grounded in cognitive psychology. It is attributed to the picture superiority effect, by which visual information is processed and remembered more effectively than text [34, 133], and to dual-coding theory, by which visuals engage the verbal and visual memory systems simultaneously [86, 87]. Sketches, in other words, are not decoration around the real work; they are the externalized form of the mental models that the work depends on.

Tooling research has tried to keep these artifacts alive: SketchLink links captured drawings to the code they describe from within the IDE [7], and LivelySketches supports versioning sketches across design iterations [8]. Modern AI code assistants are organized around text: even when they accept images, they treat them as auxiliary context rather than as a primary specification of intent, discarding the structure, ordering, and dependencies that a sketch encodes and that no equivalent text prompt recovers. If AI systems could instead interpret, produce, and maintain informal visual artifacts alongside code, they would participate in the shared understanding that today exists only between humans at a whiteboard. The following sections review how far existing work goes toward each piece of that capability, and where it stops.

2.2 Code Generation

Efforts to turn pictures into programs began where the mapping is most literal: user interfaces. pix2code generated markup from GUI screenshots [13], Ellis *et al.* reconstructed LaTeX drawing programs from hand-drawn figures [28], and PSDoodle retrieved app screens from quick sketches [79]. Closer to this thesis's domain, DLPaper2Code parsed the neural-architecture diagrams in research papers into code skeletons [110], and d'Amorim *et al.* argued that data science, organized as it is around dataflows, is a natural fit for sketch-driven programming [26].

Multimodal LLMs have since widened what image-to-code can mean. MathVista and MMMU probe visual reasoning in mathematics and college-level subjects [70, 145]; Design2Code and Plot2Code measure how faithfully models regenerate web pages from screenshots and plots from rendered figures [111, 135]; MMCode targets programming problems whose statements include images [65]; and Pădurean and Singla study visual programming tasks aimed at elementary-school students [94].

Across both generations of this work, the input image mirrors the structure of the desired output: the mockup looks like the interface it specifies, and the plot looks like the figure the code must reproduce. A whiteboard sketch of a data pipeline or a system architecture obeys no such correspondence. Its boxes, arrows, groupings, and annotations specify behavior and structure rather than appearance, at the deliberately low level of detail documented in Section 2.1. No existing system or benchmark evaluates models on this class of informal, intent-bearing input, where the sketch's ambiguity must be treated as information to reason over rather than noise to eliminate. Chapter 3 studies exactly this setting, asking whether multimodal models can turn real developers' whiteboard sketches into executable code.

2.3 Code Explanation

The explanation direction inverts the generation problem: instead of reading a human's visual intent, the AI must render code into a form a human can absorb. Developers spend more of their time understanding code than writing it [103], and it has sharpened as AI assistants recast developers as reviewers of code they did not author, reviewers who demonstrably over-trust output that looks plausible and runs without error [59, 60, 91, 119].

Software visualization has long promised relief. Reverse-engineering tools derive class,

sequence, and dependency diagrams from code [68, 100]; Software Cities render code metrics as navigable spatial metaphors [129, 130, 131]. Developers themselves report that appropriate visualization substantially reduces the effort of maintenance activities [11], and controlled studies confirm that richer code presentation speeds comprehension and helps newcomers orient and localize defects [3, 88]. But these systems all speak formal notations, and Section 2.1 established that developers do not: their sketches mix boxes, arrows, icons, and text in flexible, task-specific ways [6, 75]. UML in particular sees little sustained industrial use, drifting out of date and integrating poorly with everyday workflows [104].

Generative AI has so far filled the documentation gap with more text. LLM-based systems, including multi-agent architectures such as DocAgent, produce docstrings and repository-level summaries [142]. On the visual side, pixel-space generators (GANs, diffusion models, and autoregressive image transformers) cannot maintain the structural and semantic consistency a diagram requires [5, 83, 126, 140, 147], while structured text-to-diagram systems such as DiagramAgent start from natural-language descriptions rather than from source code [126]. Automatically producing the informal, code-grounded diagrams that developers actually draw remains an open problem.

Solving it means answering two separable questions, and this split structures the thesis. The first is *alignment*: does a generated diagram faithfully represent the code? Alignment is difficult even to measure, because many different diagrams validly depict the same program, so no reference-based comparison applies. The second is *usefulness*: does the diagram change what a developer can understand while reviewing unfamiliar code? The two are not interchangeable. An aligned diagram that helps no one is decoration, and a helpful-looking diagram that misrepresents the code is a liability, inviting exactly the misplaced trust it should guard against. Chapter 4 addresses alignment with `VisDocSketcher`, an agentic system that generates informal visual documentation from code, and `AutoSketchEval`, a reference-free evaluation framework based on round-trip reconstruction. Chapter 5 addresses usefulness with JupyterDraw, through a controlled experiment measuring whether generated visual overviews improve developers' detection of silent methodological flaws during notebook review.

2.4 Code Evolution

Translation between code and visuals in either direction, however good, produces a snapshot; software does not hold still. Developers must keep coherent mental models of systems that change daily, through interfaces that after decades of IDE innovation remain almost entirely textual [51, 139]. The cognitive evidence in Section 2.1 suggests this mismatch is costly, and empirical studies link it to the growing mental effort of maintaining understanding as systems evolve [14, 19, 64, 90].

Prior attempts to give code a living visual counterpart fall into three families, each stopping short in its own way. Derived visualizations, from UML and call graphs to Software Cities, are regenerated views: informative for comprehension, but read-only, and quick to fossilize into stale documentation once the code moves on [68, 100, 104, 129, 130, 131]. Low-code and visual programming platforms make the visual primary and editable, but only within rigid, predefined vocabularies that constrain expression and do not reflect how developers conceptualize their own

systems [1, 46, 61, 84, 98]. Sketch-linking tools such as SketchLink and LivelySketches attach drawings to the code they describe, but the link is archival: edits to the drawing never reach the code, and an evolving codebase quickly severs the correspondence [7, 8].

The formal version of this problem is well studied. Model-driven engineering and bidirectional transformation research maintains consistency between related artifacts under incremental change [35, 49], but it presupposes structured artifacts and well-defined transformation rules. Informal sketches, whose elements, groupings, and layout are unconstrained and deliberately abstract, satisfy neither assumption. Keeping such a sketch and a codebase synchronized as both evolve, propagating each edit selectively to the affected regions rather than regenerating either artifact wholesale, is the open problem taken up in Chapter 6.

2.5 Artifact Evaluation

Every claim in the three preceding sections presupposes an answer to a prior question: how do we know whether a generated artifact is any good? Human judgment remains the most accurate instrument but does not scale [31], while execution-based metrics such as pass@k presuppose comprehensive test suites that are expensive to author and rarely complete [25].

LLM-as-a-Judge approaches offer a scalable middle ground: ICE-Score scores generated code against predefined criteria [153], CodeJudge focuses on functional correctness [118], and CodeScore estimates pass ratios and executability [24], with further extensions covering non-functional properties such as readability and style [123, 132]. Each of these, however, commits to a single evaluation strategy and prompting perspective; before SE-Jury, no method combined complementary strategies into an ensemble or demonstrated agreement approaching inter-annotator levels. Visual artifacts pose a harder problem still. Because many different sketches faithfully represent the same code, there is no canonical reference to compare against, and existing documentation-evaluation practice, human raters or text metrics applied to captions, does not capture the alignment between a diagram and the code it depicts. Prior to this thesis, no automatic method for evaluating generated visual documentation existed.

This thesis contributes evaluation machinery at four points along this spectrum: an LLM-as-a-Judge rubric, validated against human grading, for scoring code generated from sketches (Chapter 3); the co-authored SE-Jury, which dynamically assembles the most effective subset of five complementary judging strategies per task; AutoSketchEval, a reference-free round-trip framework for measuring diagram-code alignment (Chapter 4); and, for the usefulness claims that no automatic proxy can settle, a controlled human-subjects protocol (Chapter 5). Together they make the thesis's central claims measurable despite the subjectivity of the artifacts involved.

Chapter 3

Code Generation

Developing software is a collaborative process that relies on the collective creativity and problem-solving skills of a team [134]. It often begins with brainstorming sessions where developers gather around a whiteboard to sketch out initial ideas and concepts [75]. This collaborative sketching helps in aligning everyone's understanding, creating a more solid foundation for the subsequent stages of development. Developers also use sketches as a form of planning their own code [19, 120], communicating and expressing their mental models through sketches, as in the typical whiteboard meetings [6]. Drawings and diagrams thus serve as visual aids to express and refine mental models of the code structure [64].

This ties into a recent development in Artificial Intelligence research: Large Language Models (LLMs) have evolved towards multi-modality, which includes strong vision capabilities [82, 116]. While the use of natural language to generate code (text-to-code) is already widespread and a common focus of research with many Code LLMs being developed [18, 43, 105], using image-to-code techniques has almost exclusively been explored for tasks such as front-end development [111] where the sketch directly captures the visual layout that the code should provide. This is orthogonal to sketches that capture the structure of a software system.

In this work, we bridge the gap between existing research on whiteboard sketches and their use to generate code, focusing specifically on Data Science (DS) systems. The demand for DS software has recently increased steeply [62]. Such systems typically involve a pipeline that transforms data in various ways, generating models and other artifacts (e.g., visualizations) along the way. As shown in this study, data scientists frequently sketch their systems and such sketches tend to be fairly rich in detail. More generally, our work explores the concept of *Visual Code Assistants*. Complementing the typical text-based assistants, a Visual Code Assistant is capable of understanding developers' sketches, helping them when they express their ideas visually.

This chapter contributes to both the research community and developers by offering guidance on developing generative code tools and providing insights into the mental models of ML developers. We implement a visual assistant by building an in-IDE sketch-to-code tool, which converts drawings of DS pipelines into IPython Jupyter Notebooks. We conducted a user study with 19 data scientists, the majority of whom regularly sketch. The study is composed of two parts: **sketching** and **programming**. In the first part, a task is given to participants and they are asked to behave as in a regular team meeting (or code interview), using the whiteboard to explain how they would solve it. In the second part, we use the Visual Code Assistant to generate an

initial code skeleton and the participant iteratively edits it to achieve the final solution. Both parts follow the think-aloud protocol [30]. The study is guided by three research questions:

- **RQ1**: What are the prevalent sketching patterns of ML developers?
- **RQ2**: To what extent can current sketch-to-code technologies support ML developers?
- **RQ3**: What are developers' perspectives on the value and potential of in-IDE Visual Code Assistants?

We observe that real drawings are highly varied, frequently mixing diagrams and textual content. Using a separate LLM to score the generated code against a set of requirements, we find that popular LLMs can generate valid outlines (the high-level structure) of the desired notebooks with 70%-80% accuracy, and implement 25%-40% of the requirements correctly from sketches alone. Longer sketch times correlate positively with outline quality, but not (significantly) with implementation details, in part because sketches often do not provide all details required to implement the code correctly. We collect rich qualitative insights from interviews conducted after the study. The contributions of our work are as follows:

1. We develop an in-IDE Visual Code Assistant prototype that converts sketches of ML-related tasks into complete Jupyter Notebooks.
2. We conduct the first exploratory study on *sketching* for ML-related tasks and the integration of vision capabilities in Code Assistants.
3. We carry out the first user study to evaluate the quality of code generated by LLMs based solely on images.
4. We provided guidance for future Visual Code Assistant development by reporting on developers' needs and identifying potential use cases where this technology can be most effectively applied.

3.1 Related Work

Programming is all about understanding tasks and creating code to execute them. To effectively work on software development, programmers must create a clear understanding of both the current code structure and any new code they need to write. This involves constructing mental models that help them visualize how the code works and how different components interact with each other.

3.1.1 Sketching in SE

In a survey with 280 software engineers, LaToza *et al.* [64] studied this phenomenon in 2006, arguing how mental models of code are expensive to create and maintain. Yet, they found that these mental models are only kept in developers' heads or "in transient forms such as sketches on a whiteboard". Furthermore, they pointed out how "paper and whiteboards were perceived most effective" while designing software. They suggested reducing the cost of using design documents by linking them to the code or building "tools that capture informal whiteboard or paper designs".

Building on LaToza *et al.* 's findings, many researchers studied sketching practices in SE. Cherubini *et al.* showed that developers often express their code in temporary drawings that are later lost, due to the cost of translating them into electronic renderings. The transient nature of whiteboard sketches can be reduced by Visual Coding Assistants. When uploading their sketches to generate code, developers make them persistent as a side effect. Branham *et al.* built ReBoard, which captures whiteboard drawings. They studied how whiteboard content is reused, identifying issues to be addressed when building augmentation systems. Similarly, Mangano *et al.* [75] identified three basic needs of developers when sketching: frequently shifting focus, using low detailed models, and using a mix of notations. They presented positive results with Calico, a tool to address these issues. We find that the flexibility of current AI vision models makes them suitable for working with this mix of notations, low-detailed models, and informal sketches. As developers frequently shift their focus, these models should be integrated into the IDE.

Walny *et al.* [121] followed the lifecycle of sketches and diagrams, reporting on their transitions and commenting on how informal sketches can usefully serve as memory aids and for communication purposes. In a second study [120], they observed 82 whiteboard sketches, concluding that participants have an "inventive capability to create [...] representation of whatever problems, processes or data they wanted to discuss". They also found that words were often used as primary objects and not only as labels. This highlights that developers have the ability to sketch different problems, and drawing is a natural way of expression.

In "Sketches and Diagrams in Practice" [6], Baltes *et al.* indicated that the majority of sketches were created informally on analog media, and were found to be valuable resources for understanding source code artifacts. They also pointed out that sketches usually represent higher levels of abstraction. They propose two new tools to bridge analog and digital artifacts: SketchLink [7] links code to sketches using a web app to capture the drawings and an IDE plugin to visualize them, and LivelySketches [8] supports the iterative process of sketching, versioning and linking drawings. More recently, Almeida *et al.* [108] interviewed 27 software architects, followed by a survey with 46 participants. Their findings contradict prior research, affirming that 76% of software architects actually document their whiteboard meetings, with photos of the whiteboard among the five most common documentation approaches. However, they also observe that in whiteboard meetings there is not "sufficient information about the problem to design the solution" and that "certain aspects of the solution are over-simplified" which makes the solution mismatch the original sketch. This matches the rate and characteristics of sketching in our work. These studies suggest that visual assistants should expect to mainly extract "big picture" information from sketches, rather than low-level implementation details.

3.1.2 Images to Code

The use of sketches as a specification for code generation has been explored previously, especially when the code is highly related to visuals. This is the case in graphic interface design, where the use of hand-drawn mock-ups is common and its translation to HTML code is very useful. Pix2Code [13] explored this problem and uses Convolutional and Recurrent Neural Networks to generate code from designers' mock-ups. Ellis *et al.* [28] convert drawings into LaTeX programs representing the intended graphics. Although not generating code, PSDoodle [79] uses sketches

as the input to perform an app screen search. On ML code generation, Sethi *et al.* present DLPaper2Code [110] that generates Python code from images of Deep Learning architectures.

With the advent of multi-modal LLMs, a range of innovative use cases are being successfully explored. Notable contributions include MathVista [70] and MMMU [145], which investigate the integration of multi-modal capabilities in mathematics and college-level reasoning, respectively. In the domain of frontend development, Si *et al.* [111] examine the performance of GPT-4 and Gemini in generating accurate web pages from images. Additionally, Pădurean and Singla [94] analyze visual programming generation performance tailored for elementary-level students. Plot2Code [135] extends the application of LLMs to code generation from scientific plots, while MMCode [65] focuses on multi-modal coding for evaluating algorithmic problem-solving skills in visually rich contexts. D'Amorim *et al.* [26], proposed the concept of sketch-to-code applied to the domain of DS, arguing that it is a highly visual discipline. We build on their vision, aiming for the broader goal of operating on complete data workflows, and generating Jupyter Notebooks based on whiteboard drawings.

In this chapter, we introduce the first benchmark specifically designed to assess the performance of leading LLMs in interpreting and generating code from whiteboard sketches.

3.2 Visual Coding Assistant

To study these phenomena, we developed a VSCode plugin where users can open a snapshot of a whiteboard sketch and have a Jupyter Notebook generated from it. This tool acts as part of the experimental materials and is used to test part of our theory regarding the usefulness of Visual Coding Assistants. In this section, we provide details about the prototype implementation and a motivating example.

Interaction. We want participants to focus on sketching and the conceptualization of in-IDE Visual Coding Assistants. The interaction that we seek to study is the naturalness of information exchange between the developer and the AI using the drawing in the same way a developer would do with other humans. We are not interested in studying the plug-in user interface interactions, leaving this factor open to future research. Therefore, our VSCode extension simplifies the user interaction to a single button that generates code. When a picture is opened in the IDE, the plugin detects the image and allows the user to click on the Generate Code button. Otherwise, a warning will tell the user that the open file is not an image, and therefore cannot be used to generate code. Once the user clicks on the button, a new panel is opened to the right of the active window, which displays the generated Jupyter Notebook.

Code Generation The code generation is delegated to OpenAI's GPT-4o with vision capabilities by using their web API. At the time the study was conducted, the model performs better than any other regarding Vision capabilities according to the benchmarks updated daily at HuggingFace [23, 71].[1] GPT4 is preferred both for the convenience of using the API and its performance. The prompt used in the final version of the prototype and the user study is the following:

[1] `https://huggingface.co/spaces/WildVision/vision-arena` &
`https://huggingface.co/spaces/opencompass/open_vlm_leaderboard`
Accessed: May 27th 2024

You are an expert code generation system and you are specialist in Python. You will receive an image representing a ML workflow and you will reply with the correspondent steps in JSON list format. The output for each step must have exactly two elements: "code" field with Python code; and "markdown" field containing the explanation. Markdown explanations must be short and easy to understand. No introductory text is allowed in the output. Retrieve only the list format, e.g.[{code: "...", markdown:"..."}, {code: "...", markdown: "..." }]
Create the code for the following diagram.

Assembling the Notebook After receiving the code generated from the model, the plugin gathers all the identified steps and assembles them into a Jupyter Notebook. Each step in the image is then translated into a pair of cells - a markdown cell that provides a brief explanation, followed by the code cell that contains the code to be executed.

3.2.1 Motivating Example

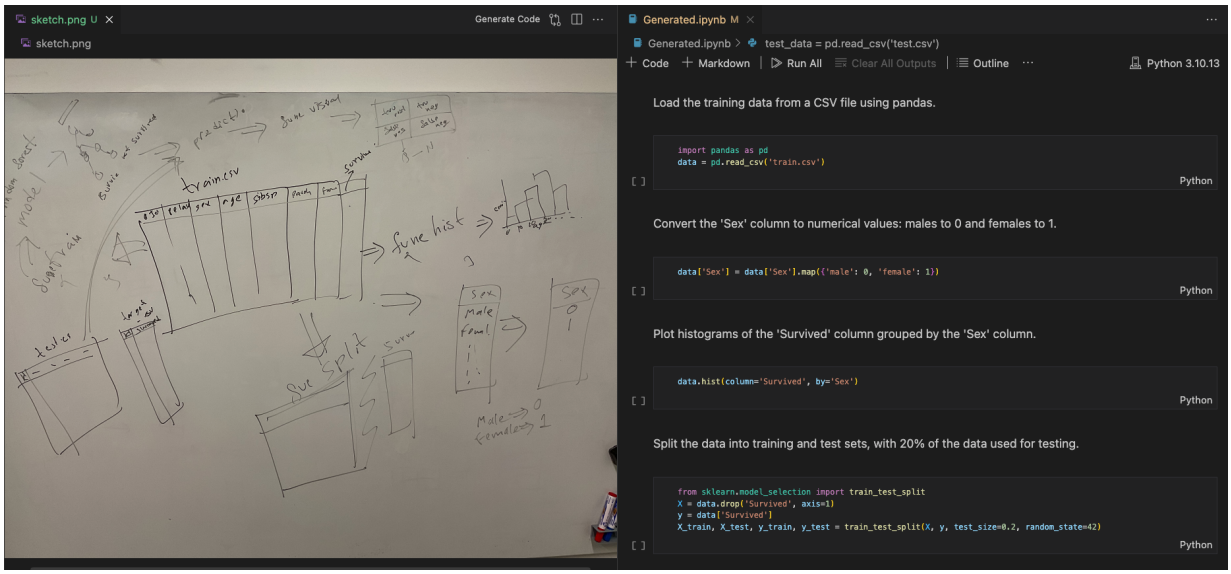


Figure 3.1: Example of usage of our prototype. Left editor tab: The whiteboard sketch uploaded by the participant to be used by the code assistant to generate code. Right editor tab: Generated Jupyter Notebook with code and Markdown explanations.

This study recreates situations where developers want to communicate their ideas using the whiteboard. We give a task to each participant and ask them to behave as in a regular meeting where they explain how to solve it. After sketching, they use our prototype to generate code. Figure 3.1 shows an example of one of our participants' usage of the prototype during the study. In this example, the participant explains how they want to load three csv files ("test.csv", "target.csv" and "test.csv"), visualize "age" with a histogram, map "Sex" (Male/Female) to numerical variables (0/1), train a Random Forest model and use a confusion matrix to see the results. The participant uses a diagram to represent the precedence of each operation. In the center, we have a data table that needs to be loaded from the "train.csv" file. Most of the other operations depend on this one,

so it must occur before the others. Causal dependencies are indicated with arrows. The participant represents the singular operations using a mix of visual icons, such as the representation of the histogram, and many short, textual annotations, such as "$Male \Rightarrow 0$".

The user uploads a picture of their whiteboard sketch to the repository they are working on and after clicking the "Generate Code" button, a Jupyter Notebook is created and opened next to the image. Using the drawing as the only source for code generation, we see the Code Assistant preserving the order of operations, while maintaining a coherent succession of steps. At the same time, it understands most of the participant intentions, such as the correct mapping, the use of a histogram function or the split of the variable "Survived" from the dataset. On the other hand, some inaccuracies are visible, such as the arguments to the histogram function or splitting the train dataset into train and test. These are things the developer may want to change or remove, according to their initial intention. In real scenarios, even with the inherent inaccuracies, this is useful to quickly generate a first code skeleton and discuss ideas at the code level during the meeting. It saves time by reducing the time on programming activities and making the design clearer to everyone in the meeting.

3.3 Methodology

The goal of our research study is to investigate developers' behavior while using the whiteboard and Visual Code Assistants. Our population of interest is individuals whose needs encompass ML and DS tasks, independent of their level of knowledge or expertise in this domain. Our sample comprises individuals from academia and industry, who have previously worked with Python and understand ML and DS concepts.

3.3.1 Recruitment

We recruited participants in two ways. The first was by identifying researchers and developers from personal contacts and contacting them directly via email. The candidates were selected taking into account previous experience. The second was by employing snowball sampling, asking previous participants to nominate new people who meet the criteria and they believed would accept to take part in the study. Each participant was compensated with a $20 Amazon gift card. We recruited 19 participants (11 female, 7 male, 1 non-binary) from different institutions in industry (4 participants - Meta, Ansys, TripAdvisor, Bandora) and academia (15 participants - CMU, NYU, Univ. of Porto, Univ. of Lisbon, Univ. of Waterloo, Brown Univ., TU Delft, Bowdoin College, Colby College, Bucknell University). Regarding the educational levels in our sample, 10 participants were pursuing or had completed a doctorate, five a master's degree, and four a bachelor's degree. Regarding their current occupation, four are working in ML or DS as their main activity, nine in Software Engineering or other programming-intensive fields, and six in other fields.

3.3.2 Protocol

Our IRB-approved protocol encompasses five stages after the participant agrees to take part in the study: 1) Pre-Task Survey, 2) Sketching, 3) Coding, 4) Interview and 5) Post-Task Survey. We designed the study to take a maximum of 60 minutes per participant. Stages 1 and 5 occur before and after the study, respectively. The study was conducted in person or virtually, depending on the participant's availability.

A pilot study with five participants was conducted to improve and validate the protocol and the tasks used in the final study. In this section, we detail each of the parts.

Pre-Task Survey. Participants complete a survey collecting demographic information and details about their sketching and coding habits. This information is crucial for understanding the diverse backgrounds and experiences of the participants, which can influence their interactions and measurements during the study. The survey is hosted at Google Forms and a link is sent to the participant before the study.

Sketching. Each participant is presented with an ML task and given a few minutes to get familiarized with the task and dataset. We allow the participant to ask clarifying questions about the task before starting the sketch. When participants confirm that they understand the task, they are asked to impersonate the team member responsible for solving the problem and simulate a whiteboard meeting where they explain their intended solution to an ML developer. At this point, a timer is started to capture sketching time, and audio is recorded to capture their thought process (think aloud). Participants were instructed to draw until they felt their explanation was complete. At this point, the timer and recording were stopped, and we captured a picture of the final sketch.

Coding: A break is made before proceeding to the coding stage. The break allows participants to switch to a new context since now they impersonate an ML developer and simulate a programming session. In the meantime we set up the in-browser VS Code IDE using GitHub CodeSpaces. This workspace includes the Visual Coding Assistant prototype presented in section 3.2 and common ML libraries that can be useful to solve the tasks. The drawing is uploaded to this workspace, allowing the user to generate code from it. Their sketch is used to generate code using the prototype and a Jupyter Notebook with the same base name as the image is generated. The participant renames the file to "generated.ipynb", and creates a copy of it calling it "solution.ipynb". The participant can make changes to the "solution" file until they are ready to submit their final solution. During the code session, a timer is started, and audio and screen are recorded through Zoom for further analysis. The think-aloud protocol is used and the participant is encouraged to speak while they do the task. During the coding session, the participant can utilize all tools that they would use in a real situation, such as Google or ChatGPT. Interactions with such tools are registered as well. To ensure the user study remains within the 60-minute time slot, participants are assigned 30 minutes to complete this step. To capture real scenario patterns, participants are not informed of this time limit beforehand, preventing any bias towards utilizing the entire 30 minutes.

Interview Questions. We conclude the user study with a semi-structured interview, focusing on three key areas to understand developers' needs regarding Visual Coding Assistants. The first area addresses sketching, discussing the intuitiveness of expressing coding ideas as sketches and how sketching patterns may influence code generation. The second area involves the participant reconstructing their work environment to identify potential use cases where they could

benefit from this technology. The final area explores future directions for developing Coding Assistants with vision capabilities and gathering participant feedback on potential features and improvements.

Post-Task Survey. Participants are asked to fill out a survey designed to collect feedback on various aspects of the Visual Coding Assistant tool. The Google Forms link is sent to the participant after the user study. The survey begins by assessing participants' initial confidence in the tool's ability to generate accurate code from their sketches. Participants rate the performance of the tool, providing their overall satisfaction with the accuracy of the generated code, and the frequency of discrepancies between their intentions and the code that was generated. Additionally, participants evaluate the tool's usefulness and provide their perceptions of how code generation from sketching could impact their productivity.

3.3.3 Tasks

We designed the tasks to simulate realistic DS scenarios in which developers want to express their ideas visually before coding, e.g., during whiteboard meetings. The tasks are similar to those encountered in ML course assignments and coding challenges. To increase the generalizability of the results, we designed three ML tasks that center around typical modeling settings: binary classification, regression, and image classification. Each task has the same types and number of subtasks, such as data loading and visualization. Table 3.1 shows a summary of the three tasks that are used in the user study.

Table 3.1: Tasks and Subtasks

SubTask	Task1	Task2	Task3
Load	Load one CSV file.	Load three CSV files.	Load two CSV files.
Transform	Split target. Split train/test	Split target. Variable encoding	Reshape image
Plot1	Boxplot	Histogram	Grid Plot
Plot2	Scatter Plot	Confusion Matrix	Line Plot
Model	Linear Regression	Random Forest	CNN

Each task was designed by starting from a Kaggle dataset and defining the five subtasks afterward. The subtasks were chosen taking into account common DS and ML routines.

The datasets were modified to guarantee that no extra complexity was added to the task and no extra execution time was needed, making it feasible to complete it in the stipulated time frame (1 hour). For Task 1, we use the House Prices dataset (`https://www.kaggle.com/datasets/vikrishnan/boston-house-prices`), for Task 2 the Titanic dataset (`https://www.kaggle.com/c/titanic/data`) and for Task 3 the Digit Recognition dataset (`https://www.kaggle.com/competitions/digit-recognizer/data`).

Tasks were assigned randomly for each participant, following a round-robin approach, with assignments made upon participant acceptance into the study. This procedure ensures a balanced distribution of tasks, with an identical number of participants per task.

3.3.4 Data and Artifacts

During the user study, we collect both qualitative and quantitative data. For privacy reasons, all the recordings and transcripts are not publicly available. This includes the screen recording of the coding sessions, and the audio recordings with the corresponding transcripts of the sketch sessions, coding sessions, and interviews.

In the publicly available data, alongside the original sketches and the generated Jupyter Notebooks, we include coding session annotations, interview thematic analysis, and sketching thematic analysis documents. The raw data of pre-task and post-task surveys is also available, after anonymizing participants' identities. All data and scripts can be found at `https://zenodo.org/doi/10.5281/zenodo.13184028`.

3.3.5 Evaluation

Our evaluation involves both qualitative and quantitative methods. In this exploratory study, we follow a constructivist paradigm [27], where we collect data and draw conclusions from it in an inductive manner. This section details the methodology used to answer each RQ in this study.

Sketch Analysis

In our study, we employed conceptual modeling [112] alongside the think-aloud protocol [30] to gather DS sketches. Participants were asked to simulate a whiteboard meeting, where they explained their approach to a given task using a whiteboard, paper, or a tablet. To analyze these sketches, we first applied open coding to identify and categorize recurring characteristics. This was followed by axial coding, which allowed us to consolidate the findings into two main categories, detailed in the results section.

Due to some inherent subjectivity in interpreting the sketches, our analysis was further supported by the audio recordings and transcripts from the sketching sessions. This additional data enabled us to trace the connection between specific subtasks and corresponding parts of the drawings, providing a clearer understanding of the sketching process and enhancing the reliability of our findings.

Code Generation

Sketching is inherently subjective, which makes the development of metrics for evaluating code generated from another person's drawings harder. Given the inherent limitations of visual representations, which often omit crucial details, it is unrealistic to expect the LLM to perfectly interpret and translate all aspects of a sketch into accurate code. To address this challenge, we established a controlled environment where participants follow specific tasks to guide their sketches. This setup enables us to create metrics based on these tasks and, by extension, infer the accuracy of the generated and submitted code. We rely on one **key assumption:** *Participants correctly comprehend and execute the provided tasks in their sketches.* This assumption is supported by audio recordings and transcripts of the sketching sessions, which are used during the sketch analysis.

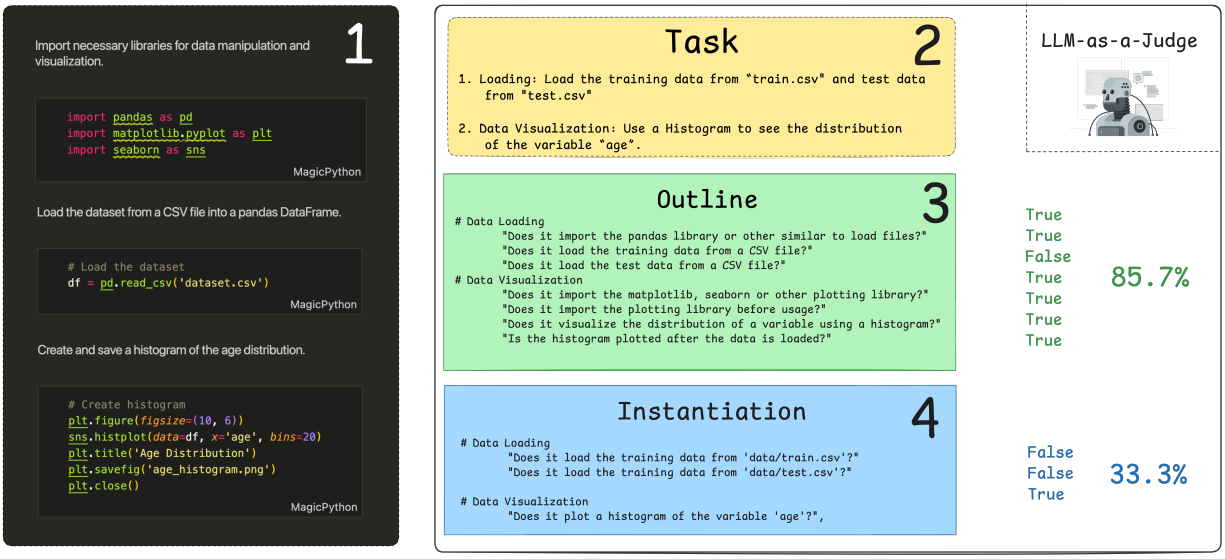


Figure 3.2: Overview of our benchmarking using LLM-as-a-Judge. To evaluate a Jupyter Notebook generated from a sketch (1), the Judge has access to knowledge elements related to the original drawing: a Task description that relates to users' coding intentions depicted on the drawing (2), and the grading criteria for the Outline and Instantiation metrics (3 and 4).

LLM-as-a-Judge

To calculate the metrics, we employ a code analysis strategy termed LLM-as-a-Judge. This approach leverages a large language model (LLM) to evaluate code based on predefined criteria, mirroring how a human evaluator would grade notebooks. The primary motivation for using LLM-as-a-Judge rather than manual grading is to ensure consistent and efficient evaluation across the dataset, particularly as new notebooks are added. Manual grading is time-intensive and lacks scalability, becoming increasingly impractical for larger datasets. In prior work, LLM-as-judge approaches achieved agreement levels comparable to human evaluators in previous work [149] and are widely used in settings like Chatbot Arena [20]. We used GPT-4 as the judge for pointwise grading as it has been reported to perform well in such task [47]. A simplified example of the evaluation process is illustrated in Figure 3.2. An evaluation of this judge in our setting is provided below.

Although this is not as precise as formal program analysis, it is more flexible in task presentation and evaluation. Participants in our study may use a wide range of libraries, e.g., pandas, csv, numpy, or dask for file loading. Implementing formal analyzers to check the correctness of pipelines would be prohibitively expensive without strict restrictions on the libraries and functions participants can use, which would reduce the realism of our study. Execution-level metrics are another common choice [18], but require fully functional notebooks. Notebooks generated from sketches are virtually never fully functional, nor is it feasible to specify complete test cases for them. Given this, we opted for the flexibility of using an LLM as a judge, combined with detailed specifications.

During the development of the LLM judge, ground-truth Jupyter Notebooks for each task were

used to verify how well the model performed against a perfect solution. For each subtask, specific parts of the code were manually removed to ensure the LLM correctly flagged the corresponding questions as 'False'. This process allowed us to curate a better evaluator, improving both the prompt and the "grading" questions. The final version of the prompt used in the evaluator was the following:

> You are a programming teacher responsible for grading Data Science coding assignments.
> You receive a Jupyter Notebook to be graded and a set of questions that you should answer with True or False.
> You should return the same list of questions answered with the corresponding answer 'True' or 'False'.
> For example, the output should be: Question1: True \n Question2: False\n etc.

Judge Validation. During the study, a second evaluation was performed, manually grading and comparing the results with the judge of a subsample of the notebooks submitted by study participants. To avoid confusion with the results presented in section 3.4, we refer to *judgment* metrics as the ones that are used to evaluate how well the LLM-as-a-Judge is capable of assessing the correctness of a Jupyter Notebook. These are intrinsic to the specific LLM that is used as the judge.

The Outline evaluation demonstrated a judgment accuracy of 92.6% (63/68), and a judgment precision of 92.2% (59/64). The Instantiation evaluation showed a judgment accuracy of 78.3% (36/46) and a judgment precision of 74.3% (26/35). These results suggest that the LLM-as-Judge methodology is reliable for this use case, as it demonstrates high levels of agreement with human judgment.

The *generation* metrics are specific to each input notebook, as shown in the example of Figure 3.2. These are reported in section 3.4.

Our quantitative benchmark evaluates Jupyter Notebooks based on the following metrics. **Outline Accuracy**: the percentage of correctly identified high-level steps in the notebook according to the sketch outline. It focuses on whether the notebook includes logic for all major sub-tasks, regardless of their specific implementation details. **Instantiation Accuracy:** the percentage of correctly implemented details within the notebook based on the sketch. This captures whether the specific elements or parameters match those sketched by the user. The instantiation accuracy is a sub-specification of the outline metric, and they are correlated. Models that perform well in the instantiation metric will perform well in the outline, but the reverse may not be true. The reason for having two different metrics is to understand the degree of detail captured by the vision capabilities of LLMs. This interacts with the sketches themselves: they virtually always contain the details required to construct the outline, but may omit details required to implement the code completely, such as the input file name. Hence, we typically do not expect a perfect instantiation accuracy.

While our main study uses GPT-4o, we conduct a retrospective analysis of three models on the collected sketches: Claude 3.5 Sonnet, Gemini Pro 1.5, and OpenAI's GPT-4o. For each model, we feed the same set of sketches, generating a Jupyter Notebook per sketch. We compare both the outline and instantiation performance. Please note that GPT-4o serves as both the judge and a participant in this experiment. While this might introduce bias, the judge only has access

to the generated notebook, the task, and the grading criteria, not the model name, so we expect that the risk of bias is slim.

Surveys and Regression Analysis

We use our benchmark to evaluate the code generated from sketches, analyzing the relationship between sketching time and code accuracy through regression analysis. This analysis considers participants' programming experience and sketching expertise, which we collected from our pre-task survey. We also control for various levels of programming knowledge and sketching expertise to understand their impact on code accuracy. Other variables from the pre-task survey were used primarily for demographic reporting, as detailed in the recruitment section. Similarly, data from the post-task survey provides additional evidence that supports our qualitative results and complements the insights gathered from the interviews.

Interview Analysis

The approach to analyzing interview data closely follows the methodology employed in sketch analysis. Initially, open coding is applied to the transcripts to identify and categorize the raw data. This preliminary step involves breaking down the text into discrete units and labeling them with codes that represent significant observations. Subsequent to open coding, axial coding is utilized to discern and establish relationships between these codes. This process helps to organize the data into coherent categories that address specific themes such as emotional responses, sketching practices, coding behaviors, and future perspectives.

Building upon these categories, an inductive thematic analysis is conducted. By identifying recurring patterns and thematic elements, this approach helps to summarize the findings into a comprehensive report that reflects the participants' viewpoints. To measure the robustness of the findings, the percentage of interviewees mentioning each theme is calculated. This quantitative measure provides insight into the preponderance of certain perspectives among participants, highlighting the strength of our conclusions. The audio and video recordings of the sketch and coding session are used as support during the analysis of the sketches and as confirmation of the coding session annotations, such as the number of times participants assessed ChatGPT.

3.4 Results

3.4.1 How Do Data Scientists Sketch?

To effectively design and implement visual methods in Code Assistants, understanding prevalent whiteboard sketching practices is essential. This section explores the findings that address **RQ1: What are the prevalent sketching patterns of ML developers?**

Sketching Practices: Quantitative Findings. In our pre-task survey, we assessed the last time participants used sketching to express their mental models during work activities. The results indicate that 36.8% of participants used sketching within the last week, 26.3% within the last month, and 21.1% within the last three months. Additionally, 15.8% had either not used sketching in over a year or did not recall their last use. These results suggest that sketching is

actively used by a substantial portion of participants, with 63.1% engaging in it within the last month alone.

Sketching Practices: Qualitative Insights. This finding aligns with interview insights that illustrate varied attitudes toward sketching. Participants who do not sketch often expressed a preference for having control over their code directly or using text prompts. P1 mentioned, "*[Sketching] is something that you don't do very frequently, you don't need to do it when you have a data set that you want to look at*". P13 and P17 show their preference for text prompts, "*I would be tempted to use the text [prompts] more often."* and "*I don't trust a computer [...] to recognize sketches. So, I prefer ChatGPT.*", respectively. Conversely, participants who sketch frequently cited its utility in clarifying and structuring their thoughts. P7 stated, "*Sometimes, I do the drawings for my benefit so that I don't get as confused*" and P12 elaborated, "*When you write down you have a whole structure [...] when you are doing the sketches, it's like you already run the whole thing in your mind.*". This suggests that sketching serves as a cognitive aid, helping developers to visualize and organize their ideas before coding, as an integral part of their problem-solving process. Along the same lines, some participants expressed their preference for the new possibility presented during the study of using sketch prompts to generate code. P9 observed that "*[Sketching] is faster than writing the code for sure*," and P11 found sketching to be more efficient compared to detailed prompt explanations, noting, "*[Prompting] also requires you more time to explain. So just being able to sketch some stuff [...] and generate the code was interesting to me*." This preference aligns with the survey data showing that a significant portion of participants use sketching regularly, suggesting it is valued for its speed and clarity in coding tasks.

Finding 1 - Sketching as a Development Tool

Developers commonly leverage sketching as a cognitive tool to visualize and organize ideas during the problem-solving process, with 63.1% of developers engaging in sketching within the past month. While some prefer traditional text-based prompts or direct coding approaches, **there is a significant demand for tools that can directly translate sketches into code**.

Sketch Styles. The first finding tells us *why* developers sketch. To understand *how* they sketch, we analyzed the sketches acquired during the study. As our focus is on people working in DS, the patterns we observed are linked to ML and data-related tasks. While it is expected given the nature of the tasks, a notable finding from our analysis is the strong emphasis developers place on maintaining order in their drawings. Participants exhibited clear mental models for structuring task dependencies, which were primarily expressed through visual representations. The following structural representations were found in the sketches:[2]

Diagrams: The most common approach (52.6%) was to use diagrams to represent relationships. Participants typically employed arrows to indicate dependencies, data flow, or the sequence of operations. These diagrams often exhibited a spatial orientation, such as left-to-right or top-to-bottom progression. Nodes within these diagrams represented components, objects, entities, or groups of operations.

[2]Note that multiple constructs can be combined on the same sketch so the percentages add up to more than 100%.

Sequential Lists: While less frequent than diagrams (42.1%), some participants used sequential lists to outline task order. These lists relied on spatial arrangement (e.g., top to bottom) to imply ordering.
Numbering: In some cases (36.8%), participants used numbers to clarify the order of operations
Freeform: Occasionally (10%), sketches lacked explicit structural elements, suggesting that participants were focusing on capturing ideas without immediate concern for order.

From the sketch analysis, we further identified patterns regarding how participants represented the operations within each subtask. Developers combine the following five elements:
Boxes: Participants frequently employed boxes or other enclosed shapes to delineate distinct operational units within their sketches. These boxes are often combined with labels or icons.
Iconography: Symbolic or graphical representations of data or concepts.
Annotations: Short text elements providing essential context for understanding the intended operations.
Descriptions: Longer and more descriptive textual explanations of operations, often using verbs.
Code: Python code, pseudo-code, or code-like expressions.

Table 3.2: Usage of element by subtask. Textual annotations are the most commonly used element in all tasks.

	Load	**Plot1**	**Transform**	**Model**	**Plot2**
Box	7	8	9	8	7
Iconography	6	10	6	4	9
Annotation	**13**	**14**	**15**	**11**	**14**
Description	2	1	2	4	3
Code	6	6	4	6	3

As illustrated in Table 3.2, textual annotations were the most frequently employed element across all subtask types, indicating a strong preference for textual descriptions of operations. While both Plot1 and Plot2 subtasks exhibited a pronounced reliance on iconography, suggesting a potential advantage of visual representations for data visualization, the creation of machine learning models was primarily conveyed through detailed textual descriptions. This was also mentioned during interviews. For instance, P4 mentioned "*[Sketching] is more intuitive than having to look up function calls on the APIs [...] there are different variations [of plots] and it's not exactly what I wanted visually.*" This also highlights that image- and text-based code generation are not mutually exclusive.

Finding 2 - Structure, Text and Icons

Developers use the whiteboard to organize their ideas by maintaining an order of operations. They rely on **diverse types of elements, including textual annotations for details and icons for visual tasks.** Effective visual assistants must prioritize accurate text recognition, while also considering spatial layout and visual symbols for enhanced comprehension.

Table 3.3: Summary of regression models predicting code generation outcomes based on sketch time, last sketching activity, and programming experience. Reported p-Values are after Benjamini-Hochberg correction.

	Instantiation		**Outline**	
Predictor	**Coeff.**	**p-Value**	**Coeff.**	**p-Value**
Intercept	24.5364	0.105	55.9498	0.004**
Sketch Time	1.4536	0.284	2.4678	0.105
Last Sketching	0.0266	0.686	0.0112	0.883
Experience	-0.4327	0.826	0.0310	0.976
R^2	0.156		0.297	
Adj R^2	-0.025		0.146	

P-Values: (*) $p < 0.1$; (**) $p < 0.05$; (***) $p < 0.01$
Adjusted p-values for multiple testing (B-H correction)

3.4.2 Sketch to Code Generation

In this section, we examine the results regarding **RQ2: To what extent can current sketch-to-code technologies support ML developers?** Our focus is on understanding the complexities of transforming whiteboard sketches into executable code, exploring developers' expectations for such tools, and assessing the capabilities of current large language models in this domain.

Results. On average, participants spent 9 minutes sketching (min: 2, max: 15) and 16 minutes programming (min: 4, max: 37[3]). This average sketch yields code with an outline accuracy of 79% and an instantiation accuracy of 36%. The median number of lines changed starting from the generated code in the final solution is 19.5, where the median solution has 40 lines in total. Given that, **sketching reduced the amount of written lines of code by 49%** on average.

To support this analysis, a manual verification of the results obtained from the submitted notebooks was conducted, with each subtask evaluated as either correct or incorrect. Subtasks were considered correct if they were executed successfully and produced the expected output. In total, 11 out of 19 participants successfully solved their assigned challenge, while an additional 3 participants completed 4 out of 5 subtasks correctly.

Individual Differences. To investigate the relationship between sketch characteristics and code generation outcomes, we conducted linear regression analyses to predict outline and instantiation accuracies. Three key variables were considered. First, we examined **sketch time**, the duration of the whiteboard sketching process, measured in minutes. Second, we explored **last sketching**, a proxy for sketching proficiency, representing the time elapsed since the participant's last sketching activity. This variable was included under the assumption that participants who had recently engaged in sketching activities would likely exhibit better drawing skills. Finally, we controlled for **programming experience**, measured in years.

Table 3.3 presents the results of the linear regression analyses predicting code instantiation and outline generation. The p-values are adjusted using the Benjamini-Hochberg correction to control the false discovery rate and account for multiple hypothesis testing. While neither

[3]An outlier; we instated the 30 minute coding time cap mentioned in the methodology after one participant spent 37 minutes on the coding portion.

sketch time, last sketching activity, nor programming experience significantly influenced code instantiation accuracy, the model for outline generation revealed a positive relationship between sketch time and generation accuracy. Specifically, each additional minute spent sketching was associated with a 2.4 percentage point increase in outline accuracy, controlling for other variables. The p-value for this variable (0.105) lies slightly outside the standard confidence intervals (0.1), which indicates that a larger sample size is needed to better estimate the true coefficient of this variable. This corresponds to an average outline accuracy of ca. 75% for sketches completed in 7 minutes to ca. 85% for ones completed in 11 minutes.

Worth noting that the intercept in the outline generation model indicates a baseline outline accuracy of 55.9% when the other predictors are zero. This does not imply that a completely blank sketch would generate any code. Rather, it suggests that minimal sketching efforts could lead to a reasonably accurate code outline. For instance, one participant's sketch made in under four minutes reached an outline accuracy of 90%.

Finding 3 - Sketch Duration and Accuracy

Even brief sketching periods have the potential to generate useful code outlines. Longer sketch durations are associated with improved outline accuracy. This suggests that code generation tools can be effectively adapted to user needs based on the desired level of code generation detail.

Model Comparison. In our prototype we used GPT-4o as the underlying model. We evaluate the performance of other leading commercial alternatives with similar functionalities, including vision capabilities and public APIs. The alternatives that we compare are Google's Gemini 1.5 Pro, Anthropic's Claude 3.5 Sonnet, and OpenAI's GPT-4o.

Figure 3.3 shows the results of the three models on our two metrics automatically calculated using the LLM-as-a-Judge approach. Outline accuracy for GPT-4o, Claude, and Gemini is 79%, 71%, and 68%, respectively. Similarly, GPT-4o leads in instantiation accuracy with a score of 36%, followed by Claude at 33% and Gemini at 26%. While statistical significance was not found in code generation performance among the models, there may still be some evidence of differences worth further exploration with a larger dataset.

Finding 4 - Generation Across Models

There is no significant difference in accuracy between GPT-4o, Gemini 1.5 Pro, and Claude 3.5 Sonnet. All three models performed substantially better at generating code skeletons (outline) compared to handling detailed code implementations (instantiation).

3.4.3 Perspectives on Visual Code Assistants

To maximize the utility of these models, it's crucial to integrate them into tools that assist developers in real-world scenarios. The concept of a Visual Code Assistant involves embedding vision capabilities into existing code assistants like GitHub Copilot. To understand the potential of this approach, we investigate developers' needs and expectations through **RQ3: What are**

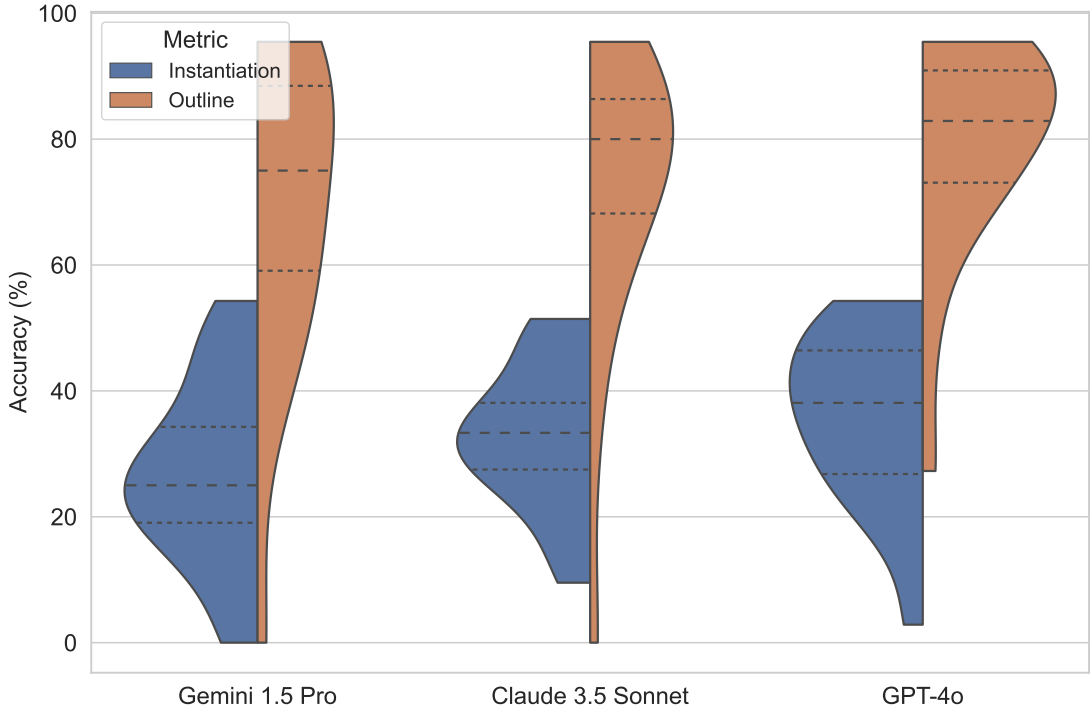


Figure 3.3: Performance comparison of top visual LLMs on the 19 notebooks. No statistically significant difference was detected in either dimension in our study.

developers' perspectives on the value and potential of in-IDE Visual Code Assistants?

Participants expressed a desire for a few key characteristics in visual code assistants. Explainability and guidelines were frequently mentioned, with users seeking to understand the rationale behind generated code and how to effectively sketch and utilize the tool. Some participants show their contentment about the generated markdown explanations: "*[it] was the most impressing part because already give these parts here that are not code, but just information*" (P3). On the other side, some participants wished they had more information, like P7 who captured the sentiment: "*if there are parts that don't require as much detail, I think those are important guidelines for the users. Like, this much detail helps the model generate better code*".

Iterative and interactive code generation was identified as another important aspect. Participants emphasized the dynamic nature of the programming process, highlighting the need to modify both sketches and code iteratively. As P10 stated, "[To change the code] I would have gone and like updated my drawing." Similarly, P11 noted the importance of "misunderstandings from the schematic and the code [being] easily fixed with some kind of back and forth interaction." Observational data verifies these findings, with 47.4% of participants revisiting their sketches at least once during the coding process, with an average of three references per participant. This suggests that developers rely on visual information to complement their cognitive models throughout the development process.

Finding 5 - Code Generation Properties

Developers need visual code assistants with clear guidelines and explainability qualities for managing users' expectations. They often revisit their sketches, indicating the need for dynamic, interactive features to enhance code generation and programming iterations.

Looking forward, participants expressed they would like the integration of sketching capabilities within the development environment. As P3 stated, "*[would help] even if you are in the code interface, and I can draw directly there*". A significant proportion of participants (21%) utilized ChatGPT during coding sessions, averaging five prompts per user. This suggests a demand for interactive tools that can assist in code refinement. As P11 noted, "*[would be useful] if you had some way to further prompt it into fixing the code itself instead of going manually yourself and trying to fix everything.*"

We found high agreement about possible use cases for visual code assistants, with participants highlighting their utility in three key areas: education, prototyping, and meetings or brainstorming. Six participants emphasized the value of visual code assistants in educational settings.

P10, for instance, highlighted their potential to aid students in learning code and design: "*Especially for students learning code and design of code, I think it would be really helpful.*". Seven participants found visual code assistants beneficial for prototyping. P6 mentioned their usefulness for creating high-level implementations or boilerplate code: "*For high-level implementations like building a boilerplate or skeleton of the tasks, this could prove useful.*" Four participants saw value in using visual code assistants for meetings and idea sharing. P12 envisioned using the tool to assign tasks to interns with a clear structural basis: "*I can just draw a sketch and give them a basic structure. So they can follow this for an earlier experiment.*" P17 saw its potential in cross-departmental settings, where visual aids could assist non-technical individuals: "*I could come up with this maybe during the meeting, especially the cross-department working environment with some non-technical people.*"

These insights reveal broad agreement on the practical applications of visual code assistants, highlighting their potential in education, prototyping, and collaborative settings.

Finding 6 - Future Use Cases

Participants identified many applications for visual code assistants, including **education, prototyping, and collaboration.** Future visual assistants should aim to support iterative refinement of code based on user feedback.

3.4.4 Threats to Validity

Construct validity. A number of metrics were introduced as useful, but imperfect proxies of real-world properties. We used years of programming experience as a proxy for domain knowledge as it is a common metric and all our participants had worked on DS tasks. However, individuals with fewer years of programming experience may possess a greater depth of knowledge in DS. We assessed sketching proficiency based on the recency of the last sketch. This assumption may not hold, but is substantially more cost-effective than formally evaluating drawing proficiency.

We also used the time spent sketching as a proxy for sketch quality, assuming that more time leads to more detailed and better sketches, which was true on average but not for all participants. Finally, we developed an accuracy metric to evaluate the quality of the generated code. This metric might be biased or misrepresent the true quality if it includes too much or too little detail. We leave more advanced metrics for future work.

External validity. The tasks used in this study were selected for their simplicity to ensure that they could be completed within the 60-minute timeframe. As a result, they may not fully capture the complexity of real-world scenarios typically encountered by Data Scientists. Given that the participants required ca. 20 minutes to complete the tasks on average, we believe that the tasks are nevertheless challenging enough to identify real effects. We made efforts to include a diverse group of participants with varying levels of expertise in DS, including students, researchers, and professional developers. Still, the sample may not fully represent the target population due to selection bias and the small sample size (n=19). This sample size proved sufficient to provide rich qualitative and some significant quantitative findings, but future work may extend this investigation to larger groups to identify more subtle effects.

Internal Validity. Given the model's extensive internet-based training, it may have encountered very similar tasks and memorized solutions. However, our data contradict this hypothesis. The LLM demonstrated a clear adherence to participant sketches, even when these diverged from common practices. For example, while an 80/20 train-test split is prevalent, participant P4 opted for a 75/25 split, which the LLM accurately implemented. Similarly, participant P13 incorrectly labeled the target variable, a mistake the LLM replicated. These observations suggest that the LLM was not merely relying on memorized solutions but actively processing and interpreting the provided sketches. We also note that the LLM knowing common DS pipelines does not negate findings on its usefulness for other DS tasks, given that such tasks typically have many elements in common.

3.5 Summary

We present the first study of developers using a tool that generates DS notebooks directly from whiteboard sketches. Our work provides key findings on the prevalence & characteristics of sketching: the majority of developers sketch regularly and mix textual annotations with iconography; the performance of visual LLMs on this task: as judged against a set of criteria, common vLLMs generate notebooks with more than 70% of the structural elements correct, but implementation details mostly need correcting; the benefits of sketching: even a quick sketch can reduce the coding effort in terms of lines of code written by around half and longer sketch times yield more accurate notebooks. Interviews with participants from both industry and academia offer rich insights into desiderata of future visual coding assistants.

3.6 Takeways

The results from *Visual Code Assistants* provide the first empirical grounding for the thesis claim that informal visual artifacts can serve as meaningful inputs to AI code generation. Three findings

are directly relevant. First, multimodal LLMs can generate code outlines with high accuracy from whiteboard sketches alone, demonstrating that the structural intent encoded in informal drawings is recoverable. This supports the hypothesis that the visual artifacts encode a latent mental model that AI can operationalize. Second, outline accuracy increases with sketch time (+2.5 percentage points per minute), suggesting that richer visual specifications yield better-aligned code: the visual modality is not merely a noisy proxy for text but an independent, information-bearing channel. Third, many participants revisited their sketches during the coding phase. Developers did not treat the sketch as a one-shot prompt to be discarded after generation—they returned to it as a living reference. This behavior, observed naturally and without instruction, is precisely the interaction pattern that motivates the co-evolution challenge: if developers already treat sketches as persistent artifacts that remain relevant as code evolves, the question is not whether to maintain that connection, but how to do so automatically.

Chapter 4

Code Explanation: Diagram Alignment

Visual artifacts are associated with developers' mental models and traditionally, developers have used informal sketches and diagrams as effective tools to reason about design, plan implementation, and communicate ideas with others [6, 19, 75, 120]. These visualizations, often drawn on whiteboards, played a key role not only in team-based discussions but also in individual workflows, helping developers align their common understanding and navigate evolving codebases more easily [64].

Field studies of sketching practices report that sketches are used mostly during the *planning* phases of coding and tend to lose relevance afterwards [6]. Even if an initial sketch is used to guide the implementation of the first version of the code, as the software evolves, the original visual representation becomes outdated and disconnected from the actual system [8]. This gap can become especially challenging when developers are inspecting new code and must reconstruct the underlying mental model without such visual support [85]. Up-to-date visual artifacts could therefore support developers in situations where they must build a mental model of unfamiliar code, such as when joining a project or reviewing code they did not write. Lowering the cost of creating and maintaining them would make this support available in settings where producing it manually is too expensive.

Motivation. One of the constant challenges in Software Engineering is code maintenance. Developers spend significant time trying to understand existing code before making modifications or adding new features [102]. Industry reports describe a growing share of new code being written with AI assistants [15], and researchers anticipate developers' routines shifting from authoring to reviewing generated code [95]. The resulting code still has to be understood and judged. Studies of current AI-assisted development report that developers are often overconfident in incorrect or insecure generated code [59, 91]. Current development environments let developers instruct an agent to modify code without reading it directly. However, formulating such instructions still requires at least a high-level understanding of what the code does. Without it, developers cannot specify what to change. This becomes more visible as code grows in size and complexity, since textual explanations alone become harder to parse. Coding agents could complement these explanations with visualizations, which prior work suggests help developers grasp code structure and behavior more quickly [42, 56]. For such visualizations to be effective, they must be both faithful to the underlying code and useful to the developer. In this work, we study the

first of these conditions: the alignment between code and the diagrams generated to represent it. Code-diagram alignment is a necessary precursor to establishing overall usefulness. We focus on a setting where these comprehension challenges are well documented: computational notebooks used in data science. Large-scale studies show that public Jupyter notebooks often lack explanatory text (one in four contains none at all) [106] and suffer from poor structural quality and low reproducibility [92, 93], while data scientists report code navigation and understanding data flow among their main pain points [16].

Generating textual documentation for code is already a common use case for LLMs and an active area of research [72, 142]. Visual documentation can complement it: textual documentation often provides low-level explanations of specific code segments or functions [96], whereas diagrams can convey a higher-level view of a program's structure and data flow, potentially reducing the effort of building an initial mental model of unfamiliar code [3, 88]. Despite these potential benefits, manually creating visual documentation is time-consuming, and its evaluation is subjective and hard to standardize and automate. Previous work has produced tools that visualize the structure of computational notebooks [50, 97, 125, 128]. These tools, however, render deterministic views of cell and dependency structure. Field studies of sketching practices report that the diagrams developers draw for each other look different: they are deliberately informal, organized around the ideas being communicated instead of the code's structure [6, 39, 75]. These are the artifacts we aim to mimic: informal diagrams developers sketch when explaining code to each other, rather than formal or exact representations. This keeps diagrams lightweight and human-like, while still capturing key information about the code. Automatically generating faithful sketches could give developers a faster path to understanding the code without deep exploration or reading long text explanations. Doing so safely requires diagrams that are demonstrably aligned with the code they represent, not just visually plausible. Despite this need, no existing approach targets the automatic generation of such informal sketches from code, and no established baselines or metrics exist for evaluating them. We tackle two challenges: *generating* informal visual documentation and *evaluating* the generated artifacts automatically, where the second is a prerequisite for measuring any progress on the first.

Our Work. This chapter explores the role of agentic systems in automatically generating diagrams that help developers understand code. We implement VisDocSketcher, a code-to-sketch system that automatically produces informal diagrams from code. Our approach combines static analysis with Large Language Model (LLM) agents to identify key elements in the code and produce corresponding visual representations. In addition, we propose an evaluation framework, AutoSketchEval, to automatically assess how well a sketch reflects its source code (*code-sketch alignment*) without reference ground-truth sketches. We selected the Data Science domain for our evaluation due to its inherently visual nature, characterized by dataflows and visualizations, as well as the availability of existing datasets with human-crafted sketches that support our experimental setup.

This work aims to answer the following research questions:

- **RQ1**: What metrics and methods can contribute to the evaluation of visual code documentation without relying on ground truth sketches?
- **RQ2**: How does our agentic sketch generation system VisDocSketcher compare to a static-analysis-based baseline?

- **RQ3**: How do notebook size and complexity affect the code–sketch alignment of the generated sketches?

We evaluate VisDocSketcher on two Jupyter Notebook benchmarks with complementary characteristics: *Visual Code Assistants Artifacts* [40] and *DistillKaggle* [81]. *Visual Code Assistants Artifacts* is the dataset derived from my first user study involving 19 data scientists and includes ground-truth, human-authored code-diagram pairs. *DistillKaggle* features code in the form of Jupyter Notebooks of varying complexity, making it suitable for evaluating the performance of sketch generators on more complex notebooks. The datasets cover diverse tasks such as data manipulation, analysis, and model training; our methodology operates on the Python code extracted from the Jupyter cells.

The experimental results show that AutoSketchEval separates aligned, human-made sketches from deliberately mismatched ones: the score distributions of the two groups differ significantly (Mann-Whitney U, $p < 0.001$), with an AUC above 0.87 and a large effect size (Cliff's $\Delta \geq 0.74$). These results support AutoSketchEval as an automatic indicator of code-sketch alignment. Moreover, VisDocSketcher outperforms the static-analysis-based baseline on all metrics (aggregate mean score 0.594 vs. 0.425, 5th percentile 0.460 vs. 0.363; $p < 10^{-6}$). The multi-agent variant scores higher than the single-agent variant in 59.3% of paired cases — a modest difference that, since only the multi-agent setup has tool access, cannot be attributed to orchestration alone. Lastly, code–sketch alignment declines significantly with notebook size: every 100 additional lines of code is associated with a 0.08 decrease in the CodeBLEU-Dataflow score (against a regression intercept of 0.27). These effects highlight the practical challenges of scaling sketch generation to complex notebooks.

The contributions of our work are as follows:

- We introduce VisDocSketcher, an agentic system for generating *informal*, high-level visual documentation directly from source code, and empirically compare two ends of its design spectrum: a single-agent setup and a multi-agent setup.
- We propose AutoSketchEval, an evaluation framework inspired by self-supervised learning concepts that reduces the dependence on manually crafted ground-truth diagrams. On expert-authored code–diagram pairs, it separates aligned from mismatched sketches with AUC ≥ 0.87.
- We provide empirical evidence on factors that influence agentic sketch generation: alignment degrades with notebook size and complexity, and the tool-equipped multi-agent configuration yields only a modest advantage over the minimal single-agent one, at an 11.3× time cost.
- We release a dataset of generated visuals, execution scripts, and intermediate representations to support reproducibility and further research.

Data Availability. All artifacts can be found at `https://github.com/luisgomes24/VisDocSketcher`.

4.1 Related Work

Our work sits at the intersection of three research threads: the evidence that visual representations match how developers reason about software, generative techniques for producing explanations

of code, and the tooling ecosystem that has grown around computational notebooks. We review each in turn and position VisDocSketcher and AutoSketchEval against them.

4.1.1 Visuals in SE and Code Assistants

Cognitive science offers two classic accounts of why diagrams help: pictures are recalled more reliably than words, the *picture superiority effect* [34, 133], and material encoded both verbally and visually engages two memory systems instead of one, as described by dual-coding theory [86, 87]. Programming, however, remains a textual activity [51], even though its core difficulty is representational: developers must hold a mental model of component interactions and dataflow that the source text never displays directly [64, 115]. Handing them a higher-level abstraction, rather than forcing them to reconstruct one from raw code, lowers cognitive load and error rates [42].

Developers have always bridged this gap by hand. Field studies of sketching in software teams find that drawings are ubiquitous, deliberately low in detail, notation-mixing, and mostly analog [6, 64, 75]. A single sketch often outlives its original purpose, turning from a thinking aid into documentation and a communication device [120]. Systems such as SketchLink [7] and LivelySketches [8] were built to keep these analog artifacts attached to the code they describe. With multimodal LLMs, the sketch has also become machine-readable. Models can turn practitioners' whiteboard sketches into executable notebooks [39]. WaitGPT [138] showed also that LLM-driven data analysis can happen through a diagram that materializes as the code is written. This work mostly studies the sketches humans draw and how to use them as *input* to code generation. The opposite direction, having the machine draw the sketch that explains existing code, lacks both generation techniques and a way to judge whether a generated sketch is faithful to the code without a human in the loop. Those are the two gaps we address.

4.1.2 Generating Code Explanations

Visuals Generation. An LLM-based system can produce a diagram in one of two ways: synthesize the image itself, or emit code in a textual visual grammar that a deterministic renderer turns into an image. Pixel-space synthesis has advanced from GAN architectures [140, 147] to diffusion [5] and autoregressive transformer models [83], and now produces convincing photorealistic imagery. what it does not reliably produce are artifacts such as flowcharts or architecture sketches, whose usefulness depends on exact spatial relationships and semantically consistent symbols rather than visual richness [126]. More recently, reasoning-augmented pixel-synthesis models have narrowed this gap by introducing an explicit layout-planning stage before rendering, predicting character-level or glyph-conditioned text layouts [17, 144] and producing infographics and labeled diagrams with legible text and plausible spatial arrangement [148]. However, the developer still lacks direct control over the generated artifact, since fine-grained or targeted revisions (e.g., moving a node, relabeling an edge, resizing a component) are not achievable through explicit edits and instead require re-prompting the model end-to-end, with no guarantee that unrelated regions of the image remain unchanged [55]. The second route sidesteps this limitation: modern code models are proficient in languages such as Mermaid and Graphviz, so a structured diagram can be generated as code and rendered afterwards. VisDocSketcher takes this route.

Multi-Agent. Splitting a generation task across specialized, coordinated LLM agents frequently outperforms a single model call [44]. Wei *et al.* 's DiagramAgent [126] applies this idea to text-to-diagram generation and editing, while Meta's DocAgent [142] applies it to code-to-text documentation, teaming a reader, searcher, writer, and verifier agents under an orchestrator. DocAgent pairs generation with an LLM-as-judge rubric (completeness, helpfulness, truthfulness), motivated by the observation that reference-based metrics require gold documentation and human judgment does not scale. We share both the architectural intuition and the concern with evaluation, but target a different artifact: none of these systems produces an informal visual representation of code for documentation purposes, and none evaluates generated visuals against the code they claim to depict. VisDocSketcher addresses the former. AutoSketchEval addresses the latter by adapting round-trip correctness [2] to the code-to-sketch setting, requiring no gold diagrams at all.

4.1.3 Notebook Tooling and Visualization

We instantiate and evaluate VisDocSketcher on computational notebooks, a domain whose comprehension problems are well documented. Analyzing over one million notebooks, Rule *et al.* [106] exposed a persistent tension between exploration and explanation, with a quarter of notebooks containing no narrative text at all. Chattopadhyay *et al.* [16] rank code navigation and dataflow understanding among data scientists' foremost pain points. Large-scale mining studies report widespread quality and reproducibility problems [92, 93]. More recently, an observational study of twenty scientists by Huang *et al.* [48] confirmed that managing state, dependencies, and abstractions is the central struggle of notebook work.

Head *et al.* 's code-gathering tools [45] clean up notebook messes by slicing the code down to what a chosen result depends on. Unlike them, we leave the notebook untouched and produce an external abstraction of it. Kery *et al.* [53] support the versioning practices of exploratory work, and Fork It [127] lets analysts branch and revisit execution states. Both assist the author during exploration, whereas VisDocSketcher explains the resulting artifact to a reader who was not part of that process. StickyLand [124] breaks the notebook's linear presentation by letting users pin cells to a two-dimensional canvas. Their spatial reorganization remains low-level, while our diagrams work with abstractions to help developers understand the flow of steps. B2 [136] and mage [54] interlink code with the *data* it manipulates, treating charts as query proxies and making GUI and code interchangeable views of the same cell, respectively. Both visualize the data being analyzed, whereas we visualize the structure of the analysis itself. NBSearch [67] supports semantic search over a notebook corpus with visual exploration of the result space. Its visualization relates many notebooks to each other, while ours explains a single notebook in depth. Closest to our goal, Themisto [122] generates cell documentation for notebooks with a human-in-the-loop workflow, but its output is textual and local to individual cells, and its evaluation relies on human assessment. We generate a global visual summary of the workflow and contribute an automatic, code-grounded measure of its fidelity.

Closer to our output format are tools that visualize notebook structure directly. Albireo [128] renders cells and their dependencies as an interactive force-directed graph. Ramasamy *et al.* [97] visualize the forking paths of data science workflows and show comprehension gains in a controlled experiment. Jiang *et al.* [50] extract labeled dependency graphs and automatically

tag cells with ML pipeline stages, and the SuperNOVA survey of 163 notebook visualization tools [125] maps the design space, highlighting the trade-off between visualization richness and notebook integration. These systems render what static analysis can see (cells, execution order, dependency edges) deterministically and exactly. VisDocSketcher aims at a different register: high-level, *informal* diagrams of the kind developers draw for one another [39], abstracting away from notebook mechanics. This distinction shapes our experimental design: the static-analysis-based baseline (Section 4.3) stands in for this deterministic family, adapted to the Mermaid format for a homogeneous comparison, and the interaction questions these tools investigate, namely how developers consume such visualizations, are complementary to the fidelity question that AutoSketchEval measures.

4.2 Automatic Visual Documentation Generator

We developed VisDocSketcher to automatically generate Mermaid diagrams from Jupyter Notebooks, enabling a lightweight and interpretable form of visual documentation that integrates easily with existing developer workflows. To ensure generalizability, VisDocSketcher analyzes general code properties (e.g., variables, functions, control flow) rather than domain-specific constructs or libraries.

Mermaid Diagram Mermaid diagrams are text-based visualizations generated from a lightweight markup language called Mermaid.js.[1] Designed for simplicity, Mermaid enables the creation of structured diagrams, such as flowcharts or class diagrams, using plain-text descriptions. This format is particularly well-suited for automated generation and reproducible documentation, as it allows diagrams to be rendered programmatically without manual graphical editing. In this work, our approach first generates the corresponding Mermaid code, which is then rendered into visual documentation using the Mermaid rendering engine.

Motivating Example We illustrate the problem with a concrete scenario: Alice has just joined a Data Science team that maintains a Jupyter-based Machine Learning workflow. Her first task is to find and fix problems in a notebook that performs data pre-processing, model training, and evaluation across multiple cells: she must add support for a new data source and an additional evaluation metric. Before she can change anything safely, she needs to understand how data flows through the notebook. The notebook contains Python code interspersed with Markdown and outputs, with several data transformations and model configurations scattered throughout. As a new intern, Alice faces some challenges:

- **Disjointed Logic**: The logic is split across multiple cells, making it hard to trace the flow of data.
- **No Overview:** There's no intuitive visual documentation that explains how the dataset moves from raw input to model evaluation.
- **Cognitive Load:** She has to mentally reconstruct the pipeline by reading and tracing code dependencies manually.

Solution. In this scenario, a visualization of the steps of the notebook, including data sources, variable dependencies, and ML model indications, can help Alice plan her change. Concretely, the

[1] `https://mermaid.js.org`

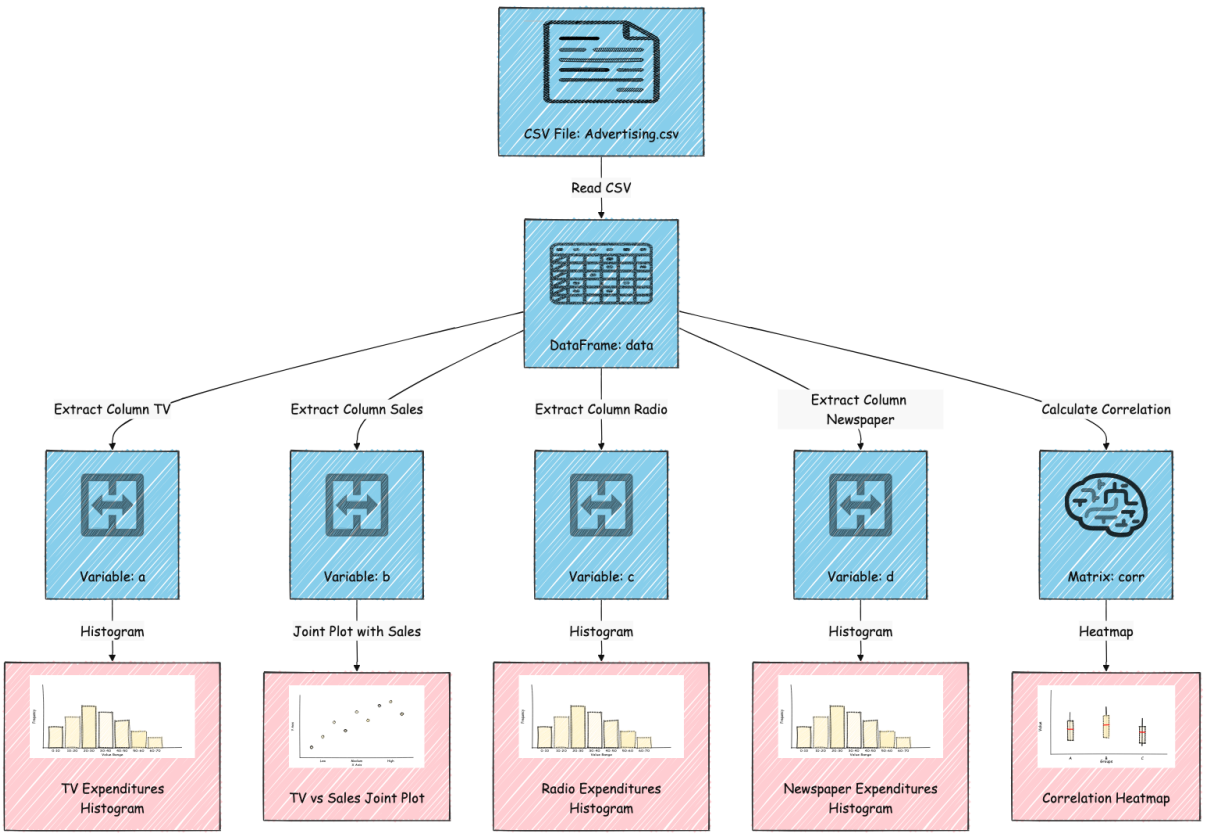


Figure 4.1: Example of a sketch generated by VisDocSketcher.

diagram lets her answer questions such as: *where is the raw data loaded? Which transformations are performed? Which variables feed the models and visualization?* With this high-level summary, Alice can locate the cells her change will touch, spot inconsistencies between the code and the expected flow, and ask informed questions before digging through every cell. In Figure 4.1 we see an example of a diagram generated by VisDocSketcher, which could be useful for Alice.

Challenges. Developing this system involves several challenges. Firstly, it requires automatically extracting meaningful structural and semantic information from code to generate accurate and readable visualizations. Secondly, assessing the quality of these visualizations requires a robust evaluation framework that can identify whether the underlying high-level code structure is accurately represented by the diagram or not. In the following subsections, we explain in detail how we tackled each of these challenges.

4.2.1 Single Agent Setup

As the initial approach in VisDocSketcher, we implemented a single-agent setup responsible for converting Jupyter notebooks into high-level Mermaid diagrams. In our context, we use Mermaid flowcharts to represent the high-level workflow of Jupyter notebooks, capturing stages

such as data loading, preprocessing, modeling, and evaluation.

In this setup, the agent reads the full content of each Jupyter notebook and executes the following prompt:[2]

(Simplified) Prompt for Single Agent

[Role] You are an expert software visualization system specialized in Python-based data science workflow...
[Task Description] Your task is to generate a high-level informal sketch of a Jupyter Notebook...
[Output Format Control] The diagram should use the Mermaid flowchart syntax and focus on representing major steps such as data loading, preprocessing, visualization, modeling, and evaluation. Use intuitive images and include emojis in node labels...
[Data] Here is the notebook content: ...

This setup has no need to access file reading tools, as the data can be passed directly to the agent. It receives the notebook content and returns a Mermaid diagram as a raw response. The prompt instructs the model to generate a high-level flowchart of the notebook's workflow, highlighting key stages such as data loading, preprocessing, modeling, and evaluation. The response is saved directly as a .mmd file (i.e., a plain text file written in Mermaid syntax that can be rendered into a Mermaid diagram). The single-agent system also includes logic to handle missing files, skipped generations, and consistent output directory structures, enabling scalable and automated diagram generation across large notebook datasets.

4.2.2 Multi Agent Setup

Following successful work in other areas [142], we built a second version of VisDocSketcher as a multi-agent system, where each agent is responsible for a specific task as shown in Figure 4.2. The architecture is implemented using LangGraph [63] and relies on OpenAI's *GPT-4o-mini* model to drive the reasoning capabilities of each agent. The architecture does not depend on this particular model; we report how performance varies across model choices in the robustness analysis. [3] Each agent operates independently, maintaining its own reasoning prompt and toolset. The communication is always made through the supervisor, with intermediate artifacts being saved to the local filesystem.

Supervisor Agent. At the core of the system is the supervisor agent, responsible for interfacing with the user and coordinating all other agents by directing when and how they should operate. It is the only agent that receives a user prompt; all remaining agents are guided exclusively through our system prompts, and by the supervisor prompts during execution. To monitor progress and ensure the correct completion of each step, the supervisor is equipped with a dedicated `file_exists` tool, allowing it to verify the presence of generated artifacts throughout the workflow. All the other agents have two common tools: `write_file` and `read_file`.

[2]The full prompts, including those used for the multi-agent setup, are available in the replication package.

[3]For consistency and robustness tests across different models, please see the comparisons in **Robustness Analysis** of Section 4.4.

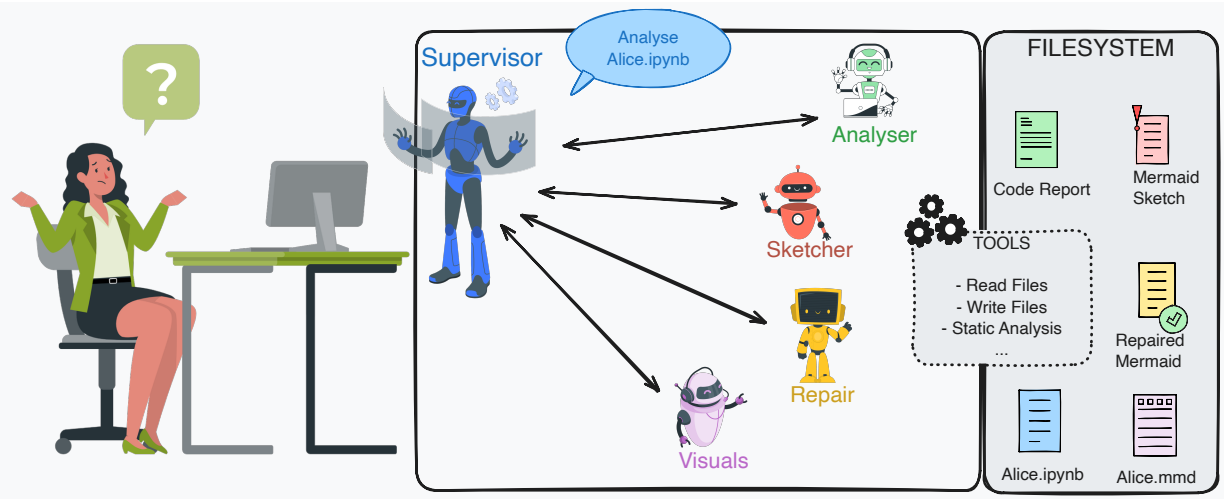


Figure 4.2: Overview of the multi-agent architecture. A supervisor agent receives the user query and orchestrates the workflow by delegating tasks to specialized agents. Each agent communicates exclusively with the supervisor (no inter-agent communication). Agents are equipped with tools to access and manipulate the local filesystem.

Analyzer Agent. This agent processes Jupyter notebooks to extract key structural and semantic elements. It identifies data sources, logical sections of the notebook, and the flow of data variables across cells. In addition, it recognizes ML components such as model definitions, training routines, and evaluation steps. This agent uses a function to extract notebook cells and static analysis approaches, including the construction of data flow graphs and analysis of variable dependencies, to understand the notebook. Then it writes its analysis to an intermediate file for downstream agents. The tools that it can use are `build_dataflow_graph_ast`, `extract_variable_dependencies` and `extract_notebook_cells`.

Sketcher Agent. The sketcher consumes the analysis output from the analyzer agent and generates an initial workflow diagram using Mermaid.js. This diagram focuses on the data flow and logical dependencies inferred from the code. The resulting sketch provides a clear, structural overview of the workflow. It does not have extra tools.

Repair Agent. Given that Mermaid.js syntax can be sensitive and sometimes error-prone, the repair agent is introduced to automatically identify and fix issues in the generated diagrams. This includes handling malformed syntax, escaping special characters, and ensuring that the final diagram is valid according to Mermaid specifications. The agent validates the syntax before saving the corrected version for further enhancement. It has access to *validate_mermaid_from_file* tool that allows the syntax verification of Mermaid.js files.

Visuals Agent. To make the diagrams more intuitive and engaging, the repaired sketches are augmented with visual elements. This includes adding emojis to nodes, applying color schemes based on node types (e.g., data vs. model), and integrating dummy plots or icons that hint at the function of each step. This agent ensures visual consistency and improves readability without compromising the underlying structure. The visual vocabulary follows observations from prior work on developer sketching practices, in which practitioners frequently used iconography and

small plots in their informal sketches [39].

How Agents Collaborate. When the user requests the generation of a new diagram from code, the Supervisor Agent first calls the analyzer Agent which processes the notebook; the Sketcher Agent produces an initial diagram based on this analysis, the Repair Agent corrects any syntactic issues, and the Visuals Agent enriches the output with visual elements; However, the *Supervisor* can decide to repeat any of the steps. Its decisions are based on the textual reports each agent returns through the supervisor channel, combined with `file_exists` checks on the expected intermediate artifacts; for instance, if an expected file is missing or an agent reports remaining issues, it can request the Repair Agent to act again after the Visuals Agent.

The modular structure of the system allows for easy extensibility, enabling researchers to add new agents or modify existing ones with minimal coupling. Similar systems could be adapted to other areas of SE, such as software architecture.

N

4.2.3 Evaluation Framework

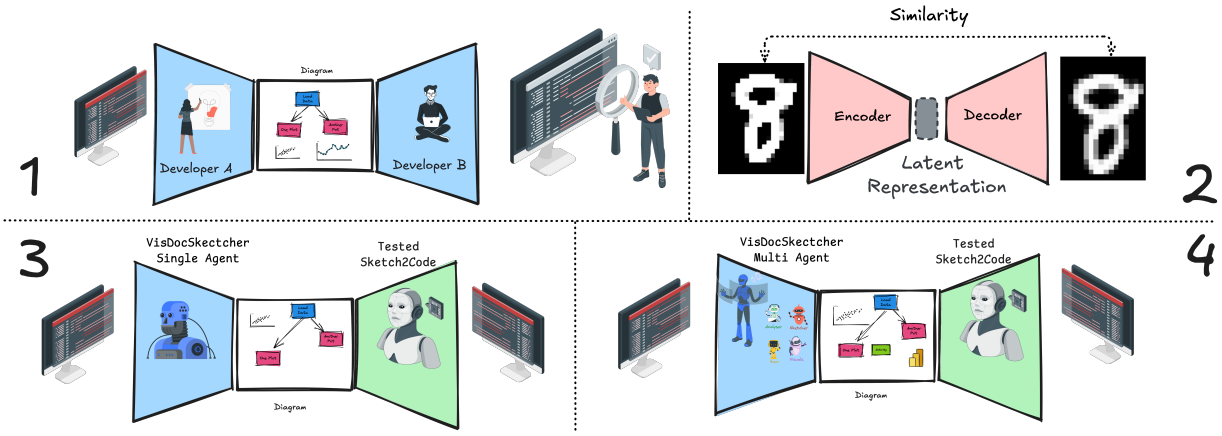


Figure 4.3: **Experimental Setup.** 1) *Human Analogy*: The similarity between the code produced by developer B and the original code depends on how informative the diagram is. 2) *Autoencoder Analogy*: Reconstruction similarity (round-trip correctness) indicates the quality of the latent representation. 3) *Single-Agent Evaluation*: Using a tested Sketch2Code model, semantic similarity reflects the effectiveness of VisDocSketcherin generating code-aligned sketches. 4) *Multi-Agent Evaluation*: Using the same setup, we compare single and multi-agent architectures.

One of the key challenges in evaluating the quality of generated visualizations is their inherent subjectivity: the same concept can be represented in multiple valid ways (e.g., loading a CSV file might be depicted as a box, a text label, or an icon). Additionally, there is a lack of labeled datasets that assess the quality of such sketches at the high-level of code flow and intent.

Inspiration. As depicted in ① of Figure 4.3, suppose we have two human developers: one who reads the code and produces a high-quality sketch (*Code2Sketch*), and another who views only the sketch and create a new program (*Sketch2Code*). If the second developer successfully produces code similar to the initial, it is intuitive to conclude that the sketch is relevant and

faithfully represents the original code. When the same Sketch2Code developer is used throughout, differences between reconstructions can be attributed to the intermediate sketch, and therefore to the capacity of the Code2Sketch developer to represent the code visually; the second developer's own skill still bounds how good any reconstruction can be. In Machine Learning, there is a similar concept, applying an unsupervised methodology that uses reconstruction with unlabeled datasets. *Autoencoders* [66] are neural networks designed to learn efficient data representations by compressing the input data into a lower-dimensional *Latent Space* and then reconstructing it. As depicted in ② of Figure 4.3, it is composed of two parts: the *Encoder*, which reduces the dimensionality of the input data, creating a lower-dimensional representation, while the *Decoder* reconstructs the original input from the compressed representation. The similarity between the input data and the reconstructed data is the *Reconstruction Similarity* [9]. Previous work used the same idea to evaluate the round-trip correctness (RTC) for code LLMs [2], where an LLM is asked to describe the code and subsequently implements the relevant code given that description, measuring the semantic equivalence between the original and reconstructed code samples. Our work adapts this round-trip principle to the visual domain: instead of using natural language as the intermediary representation, we use diagrams.

Translating the human scenario and inspired by the autoencoder concept, we adapt these ideas to visual documentation. Our evaluation framework consists of two main components: a Code2Sketch model, equivalent to the *Encoder*, and a Sketch2Code tool, equivalent to the *Decoder*. The counterpart of the *Latent Space* is the generated visual documentation, which will represent the underlying code. The final score of the framework depends on two factors: the generated visualization, and the ability of the Sketch2Code component to recover code from it. We adopt a Sketch2Code approach from peer-reviewed work [39] and hold it fixed across every condition we compare, so that differences in score can be attributed to the generated visualization and, consequently, to the Encoder's ability to produce an accurate and meaningful sketch from the original code. The Sketch2Code component is itself imperfect, which places a ceiling on the absolute scores the framework reports: a stronger decoder would raise the scores of all conditions, and one that systematically fails on a particular diagram style would understate the sketches written in it.

What reconstruction similarity does and does not measure. A natural objection to round-trip evaluation is that a sketch is meant to be a high-level *abstraction* of the code, not a literal translation of it — under that view, near-perfect reconstruction would signal a diagram that merely transliterates the code. Two properties of our setup limit this risk. First, the intermediate representation is an informal Mermaid flowchart whose nodes name workflow stages rather than statements, so the reconstruction is inherently lossy; in practice, even well-aligned, human-authored sketches yield reconstruction scores far below identity (Section 4.4), indicating that the framework operates in an abstraction regime rather than a copying regime. Second, we emphasize the dataflow component of CodeBLEU, which rewards preserving the structure of the computation (data sources, stages, dependencies) rather than its surface form. Nevertheless, reconstruction similarity measures how *completely* a sketch preserves the information needed to recover the code's structure — a necessary condition for a faithful visualization, but not a sufficient one for usefulness: it does not capture abstraction level, readability, or whether the sketch actually helps developers. We therefore treat AutoSketchEval as an automatic indicator

of code-sketch alignment, and we return to the unmeasured aspects in the Threats to Validity section.

4.3 Evaluation Design

Our work aims to answer three Research Questions (RQs).

- **RQ1: What metrics and methods can contribute to the evaluation of visual code documentation without relying on ground truth sketches?** In RQ1, we assess the validity of our visual documentation evaluation framework, AUTOSKETCHEVAL: whether its scores can separate sketches that are aligned with the code from sketches that are not.
- **RQ2: How does our agentic sketch generation system VISDOCSKETCHER compare to a static-analysis-based baseline?** In RQ2, we compare the sketches generated by VISDOCSKETCHER — in its single-agent and multi-agent variants — against a static-analysis-based baseline.
- **RQ3: How do notebook size and complexity affect the code–sketch alignment of the generated sketches?** In RQ3, we analyze how notebook size and structural complexity impact the alignment scores of sketches generated by VISDOCSKETCHER.

Datasets. Our framework is evaluated in the Data Science domain, which closely aligns with our use case and benefits from publicly available datasets that support comprehensive testing. We required datasets containing complete, high-level workflows rather than isolated snippets.

We used two recently published datasets with complementary characteristics: *Visual Code Assistants Artifacts* [40] and *DistillKaggle* [81]. *Visual Code Assistants Artifacts* is derived from a user study involving 19 data scientists from industry and academia (e.g., Meta, TripAdvisor, CMU, MIT, NYU), and includes both *ground-truth, human-authored diagrams* and the corresponding submitted code. These expert-created code-sketch pairs make this dataset particularly valuable for the validation of our evaluation framework (RQ1) and to make comparisons between our agentic approaches with the baseline (RQ2).

DistillKaggle is a large-scale, curated dataset extracted from Kaggle notebooks, comprising over 500,000 files. It includes metadata such as *Lines of Code*, *Number of Visualizations*, and *Developer Tier*. Although this dataset lacks ground-truth sketches, it is valuable for studying how the code–sketch alignment of generated sketches is influenced by code complexity and author expertise. From the *DistillKaggle* corpus, we cleaned and curated a sample of 1,000 Jupyter notebooks, balanced across key metadata dimensions (Lines of Code, Number of Code Cells); the filtering and sampling scripts are part of the replication package. This curated sample is appropriate for studying complexity effects (RQ3), although we note that even its most complex notebooks remain small compared to production software, so our findings speak to complexity variation within typical notebooks.

For simplicity, we refer to the *Visual Code Assistants Artifacts* dataset as the **ground-truth dataset**, as it contains human-authored visuals, and to *DistillKaggle* as the **large dataset** for its size.

Baseline. Since there are no established baselines for this task in the existing literature, we implemented a static-analysis-based baseline. This approach extracts information from the code

and generates the corresponding diagram to represent the data flow structure. We include this non-LLM method to compare against traditional static analysis techniques. These methods capture all dependencies but provide little abstraction and are inflexible in diagram styles, producing call and dependency graphs that differ from our informal Mermaid diagrams. To enable a fair comparison, we adapted the static analysis output into Mermaid, creating a deterministic template for evaluation within a homogeneous framework (see results in Table 4.3). Although this baseline captures some structural elements, it lacks the semantic richness provided by LLM understanding used in our method. Comparing it against our agent-based approach lets us isolate the contribution of code-aware LLM reasoning over pure structure extraction. Existing notebook visualization tools similarly render deterministic views of a notebook's cell and dependency structure; our baseline can be seen as a representative of this family, adapted to the Mermaid format so that all conditions are compared within a homogeneous framework.

Metrics. In this work, we adopt our proposed novel visualization evaluation framework AUTOSKETCHEVAL (introduced in Section 4.2.3) to assess the generated sketches. In Section 4.4, we present evidence for the validity of this framework: its scores separate code-aligned sketches from mismatched ones.

AUTOSKETCHEVAL uses the similarity between the code reconstructed from a sketch and the original code as a proxy for the sketch's code–sketch alignment. As discussed in Section 4.2.3, this proxy captures how completely a sketch preserves the code's structure, not how useful the sketch is to a developer. To assess the reconstructed code, we follow prior work [99, 150] by comparing it to the original code. Specifically, we adopt a combination of well-established and widely used code similarity metrics in the literature [151]: match-based and embedding-based metrics.

We employ CodeBLEU [99], a code-specific extension of the BLEU metric. CodeBLEU is composed of four sub-metrics: (1) *n-gram match*, which measures token-level overlap; (2) *weighted n-gram match*, which incorporates domain-specific weighting; (3) *syntax match*, which compares syntactic structures using AST-based representations; and (4) *data flow match*, which captures the semantic correctness of variable usage and dependencies. The final CodeBLEU score is a weighted combination of these components, offering a more nuanced similarity assessment than traditional BLEU. In this work, we use both *CodeBLEU-full* (an average of all four components) and *CodeBLEU-dataflow* (using only the data flow component), with the latter only reflecting high-level semantic similarity.

We use CodeBERTScore [150], an embedding-based metric built on top of the CodeBERT model [32]. It computes similarity by embedding the original and reconstructed code snippets and comparing them in vector space. CodeBERTScore accounts for lexical and semantic similarity and includes precision, recall, F1, and F3 sub-metrics. This approach is particularly useful when the reconstructed code differs in surface form but maintains functional equivalence. In this work, we use both *CodeBERTScore-F1* and *CodeBERTScore-Precision*, which are widely used in prior studies. We note that all of these metrics remain largely syntactic: even CodeBERTScore, while more semantic than n-gram matching, compares embeddings of the code's text rather than its behavior. Metrics based on functional equivalence (e.g., executing the reconstructed notebooks and comparing outputs) would be more direct; however, the notebooks in our datasets lack test oracles and depend on external data files, making execution-based comparison impractical in this setting, and we leave it as future work.

4.3.1 Experimental Setup

AutoSketchEval Framework Validation (Setup for RQ1). A minimal requirement for an automatic evaluation framework of subjective artifacts is that it can reproduce clear-cut human judgments. In our case, it should distinguish a sketch that a human annotator would consider *aligned* with the code from an *unaligned* one. Note that this is a necessary rather than a sufficient condition for validity: separating aligned from deliberately mismatched pairs is a coarse test, and passing it does not imply that the framework can rank subtle differences between plausible sketches of the same code. Each visual documentation can *correctly* or *incorrectly* reflect the intentions of the developer and the corresponding code. Consequently, the framework is valid if it is able to separate these two cases accurately.

To evaluate our hypothesis, we pair each drawing from the *ground-truth dataset* with two distinct sets of code notebooks. For the *aligned* reconstruction condition, each drawing is matched to the corresponding solution notebook submitted by the same participant, ensuring a strong alignment between the diagram and the underlying implementation. For the *unaligned* reconstruction condition, we deliberately pair each drawing with a randomly selected notebook, creating a mismatch between the visual representation and the code. Significance and magnitude tests, such as AUC [69], are performed to verify the separation capability of the framework.

VisDocSketcher Evaluation (Setup for RQ2).

We use the high-quality *ground-truth dataset* to compare our proposed tool, VisDocSketcher, against the baseline. Specifically, both VisDocSketcher and the baseline generate sketches for each Jupyter Notebook, and the sketches are assessed using our evaluation framework, AutoSketchEval, whose validity is examined in RQ1.

Impact of Input Complexity (Setup for RQ3). To investigate how code–sketch alignment is influenced by notebook complexity, we conduct experiments on the *large dataset*, which contains notebooks with varying lengths and structural complexity. This dataset is well-suited for analyzing the impact of input complexity. Specifically, we conduct a linear regression analysis using Lines of Code (LOC) and Number of Code Cells (CC) as independent variables, while controlling for developer expertise through the Performance Tier (PT) variable. As the dependent variable we select the *Dataflow* metric as it captures dependency relations among variables [99], which is the most relevant to what we want to see when building a representation of a data science workflow. Among the metrics validated in RQ1 it also separates aligned from unaligned sketches with the smallest margin (Table 4.1), making it the most conservative choice for this analysis: an effect of notebook complexity observed under this metric is unlikely to be an artifact of an easily separated measure.

4.4 Results

RQ1: What metrics and methods can contribute to the evaluation of visual code documentation without relying on ground truth sketches?

In **RQ1**, we assess the validity of our evaluation framework, AutoSketchEval, by testing its ability to distinguish sketches that are aligned with the code from those that are not. As introduced in Section 4.3.1, we used the ground-truth data specifically for this purpose, containing both

aligned sketches paired with their corresponding code and unaligned sketches with unrelated code.

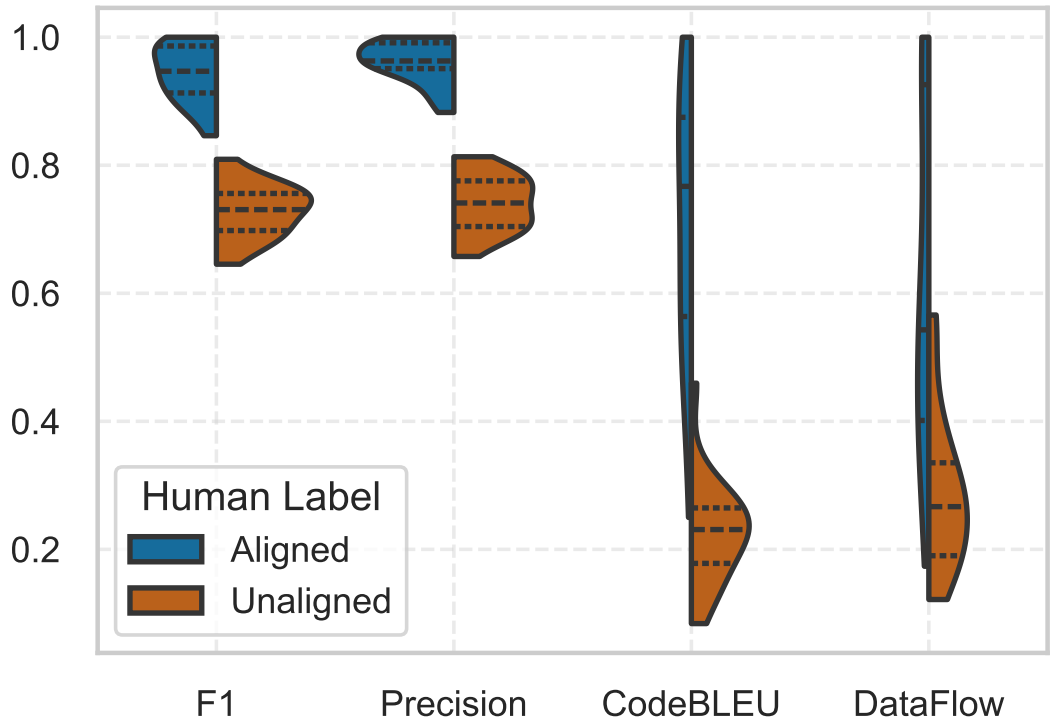


Figure 4.4: AUTOSKETCHEVAL: Framework validation results based on expert-authored ground-truth visuals. *Aligned*: Samples where the diagram and code corresponds to the same task. *Unaligned*: Samples where the diagram and code correspond to different tasks.

Results. Figure 4.4 presents the distribution of evaluation scores measured by CodeBERTScore-F1, CodeBERTScore-Precision, CodeBLEU-full, and CodeBLEU-Dataflow, for code reconstruction in two scenarios: sketches well aligned with the reference code and randomly sampled sketches lacking such alignment. The result confirms that our proposed evaluation framework can effectively distinguish aligned sketches from unaligned ones using code metrics (e.g., CodeBLEU) of the code reconstructed from sketches. However, we can also identify that different metrics present distinct abilities to discriminate between the two groups, i.e., some distributions present more overlap than others.

To evaluate which of the selected metrics more significantly distinguishes the two cases, we applied two complementary non-parametric statistical tests: the *Mann-Whitney U* test [77] and the *Kolmogorov–Smirnov (KS)* test [76]. Both tests show statistical significance for every metric (p-value < 0.001), so all four metrics separate the aligned from the unaligned condition; their detailed results are therefore omitted from the table. Because significance holds throughout, we rank the metrics by the magnitude of the separation they achieve, reported in Table 4.1 as AUC and Cliff's Δ. The embedding-based CodeBERTScore metrics separate the two groups perfectly (F1 and Precision: AUC = 1.000, Cliff's Δ = 1.00), which is consistent with their more

Table 4.1: AUTOSKETCHEVAL – Separation Between Aligned and Unaligned Samples. CodeBERTScore's F1 and Precision achieve perfect discriminative power (AUC = 1.000, Cliff's Δ = 1.00), fully separating aligned from unaligned code-sketch pairs, while CodeBLEU-based metrics show strong but incomplete separation.

Source	Metric	↑ AUC	↑ Cliff's Δ
CodeBERTScore	**F1**	**1.000**	**1.00**
CodeBERTScore	**Precision**	**1.000**	**1.00**
CodeBLEU	CodeBLEU	0.978	0.96
CodeBLEU	Dataflow	0.871	0.74

Note: the Mann–Whitney U and Kolmogorov–Smirnov tests are significant at $p < .001$ for every metric listed.

semantic nature, and they are the most effective of the metrics we consider. CodeBLEU-full follows closely (AUC = 0.978, Δ = 0.96), and CodeBLEU-Dataflow separates the groups with the smallest margin (AUC = 0.871, Δ = 0.74), which still corresponds to a large effect. We note, however, that all metrics considered here remain largely syntactic approximations of code similarity (see Section 4.3).

Separation Magnitude. It is also important to understand the magnitude of the separation between the two cases. The bigger the separation in this binary setup, the better. The Area Under the Curve (AUC) [69] measures how well a metric can distinguish between two classes, in this case, aligned vs. unaligned. An AUC of 1.0 means perfect separation, while 0.5 indicates no better than random chance, given the equal number of good and bad cases in this experiment. Cliff's Delta quantifies the effect size by measuring how often values in one group are higher than in the other. It ranges from -1 to 1, where 0 means no difference, and values above 0.47 are considered large effects. Together, AUC and Cliff's Delta show not just whether the groups differ, but how strongly and consistently they do. The results of group separation are also presented in Table 4.1. The results show that the evaluation framework AUTOSKETCHEVAL achieves consistently high AUC values (≥ 0.87) and Cliff's $\Delta \geq 0.74$, representing a strong ability to discriminate between different classes.

Table 4.2: Robustness analysis: pairwise comparison of LLM model combinations, including same-model pairs and cross-model examples. The *Significant* column indicates whether differences in the distributions between the two setups results are statistically significant. Full results and analysis are available in the replication package.

Model Pair	p-value	Significant
GPT+GPT vs. Gemini+Gemini	0.35	False
GPT+GPT vs. Mistral+Mistral	0.18	False
Gemini+Gemini vs. Mistral+Mistral	0.98	False
GPT+Gemini vs. GPT+Mistral	0.49	False
GPT+Mistral vs. Gemini+GPT	0.62	False
Gemini+GPT vs. Gemini+Mistral	0.88	False

Robustness Analysis. To assess how sensitive the framework is to the choice of LLM, we study the framework's robustness across different LLM combinations. To cover a diverse set of architectures and vendors, we tested three models for both code2sketch and sketch2code: OpenAI's GPT-4o-mini, Mistral's Mistral-small-2503, and Google's Gemini-2.5-flash. The analysis reveals consistent performance across all nine possible model pairings, with average CodeBLEU scores ranging narrowly from 0.324 to 0.370 and dataflow scores from 0.292 to 0.452. We find no evidence that performance differs across model combinations: Kruskal-Wallis tests give $p = 0.869$ for CodeBLEU and $p = 0.083$ for Dataflow, and all 36 pairwise Mann-Whitney U comparisons with Holm correction (table 4.2) yield $p > 0.05$.[4] We note that this does not establish equivalence: a non-significant test is an absence of evidence of a difference, not evidence that no difference exists, and with 36 Holm-corrected comparisons our sample has limited power to detect moderate effects. We therefore report this as consistency across the models we tested, rather than as model-agnosticism in general.

Finding 1: The Framework Separates Aligned from Unaligned Sketches

The evaluation framework separates code-aligned from mismatched sketches, achieving consistently high **AUC values ($\geq$ 0.87) and Cliff's $\Delta \geq$ 0.74** across all metrics, reaching perfect separation with the embedding-based CodeBERTScore-F1 and CodeBERTScore-Precision.
The separation is stable across the three LLMs we tested and requires only code-level metrics, with no ground-truth diagrams. This makes it possible to scale evaluation and automatically flag sketches with low code–sketch alignment.

RQ2: How does our agentic sketch generation system VisDocSketcher compare to a static-analysis-based baseline?

In **RQ2**, we present the performance of the agentic sketch generation system VisDocSketcher, and compare each variation, single- and multi-agent, with the baseline introduced in Section 4.3. For that, we use the validated framework AutoSketchEval to assess the round-trip correctness with the *ground-truth dataset*, replacing the human-authored diagrams with the corresponding generated visuals for each condition. This setup ensures the results are consistent with the validation scenario.

Results. Table 4.3 reports the mean and 5th percentile scores across several metrics, comparing the baseline to the average performance of VisDocSketcher (averaging single-agent and multi-agent variants). The 5th percentile indicates a conservative performance bound: 95% of outputs achieve this score or higher, offering insight into worst-case behavior. Across all metrics, VisDocSketcher outperforms the baseline, with all p-values below 10^{-6}: the aggregate mean score rises from 0.425 to 0.594, and the aggregate 5th percentile from 0.363 to 0.460 (Table 4.3). The improvement in the lower tail indicates fewer weak outputs, i.e., more consistent code–sketch alignment across notebooks.

[4]The full table and analysis can be found in the replication package: `https://github.com/luisgomes24/VisDocSketcher`

Table 4.3: Comparison of baseline and VisDocSketcher performance, reporting the 5th percentile and average values. F1 and precision denote CodeBERTScore components (embedding similarity between original and reconstructed code), not classification metrics.

Metric	Perc. 05		Average	
	Baseline	VisDocS	Baseline	VisDocS
F1	0.633	**0.753**	0.690	**0.836**
precision	0.725	**0.784**	0.774	**0.863**
CodeBLEU	0.067	**0.177**	0.132	**0.333**
dataflow	0.026	**0.125**	0.103	**0.342**
Average	0.363	**0.460**	0.425	**0.594**

Finding 2: Improvement Over Baseline

VisDocSketcher outperforms the static-analysis-based baseline across all evaluated metrics ($p < 10^{-6}$): **the aggregate mean score rises from 0.425 to 0.594, and the 5th percentile from 0.363 to 0.460.** The gains hold in the lower tail as well, indicating more consistent code–sketch alignment across notebooks.

Single Agent v.s. Multi-Agent. We also compare the performance of the two VisDocSketchervariants: the single-agent and multi-agent approaches. Specifically, we compute pairwise differences for each metric i with $\Delta_i = \text{score}^i_{\text{MA}} - \text{score}^i_{\text{SA}}$. As shown in Figure 4.5, the Multi-Agent system outperforms the Single-Agent in **59.3% of cases (p-value = 0.0335)**. This is a modest advantage. Importantly, the two variants differ in more than orchestration: the single-agent setup is a plain LLM call without tools, whereas the multi-agent setup combines orchestration with tool access (static analysis, file operations, syntax validation). The comparison therefore reflects the combined effect of both factors, and the improvement cannot be attributed to multi-agent coordination alone; isolating the contribution of tools (e.g., through a tool-equipped single agent) is left for future work.

However, there is a significant disparity in processing time between the two architectures. On average, the single-agent system completes generation in 8.27 seconds (SD = 7.84), whereas the multi-agent setup takes 93.43 seconds (SD = 29.24), making it 11.3 times slower. This substantial difference reflects the coordination overhead inherent in multi-agent workflows, where specialized agents communicate and validate intermediate outputs. While this design yields higher alignment scores in most paired cases, it introduces a notable computational cost that must be considered when deploying at scale.

Finding 3: Performance and Computation Trade-off

The **tool-equipped multi-agent system outperforms the minimal single-agent system in 59.3% of the cases** ($p = 0.03$) — a modest difference that reflects both orchestration and tool access. It comes at a substantial cost: the multi-agent setup is approximately **11.3 times slower**. These results highlight a trade-off between code–sketch alignment and system efficiency.

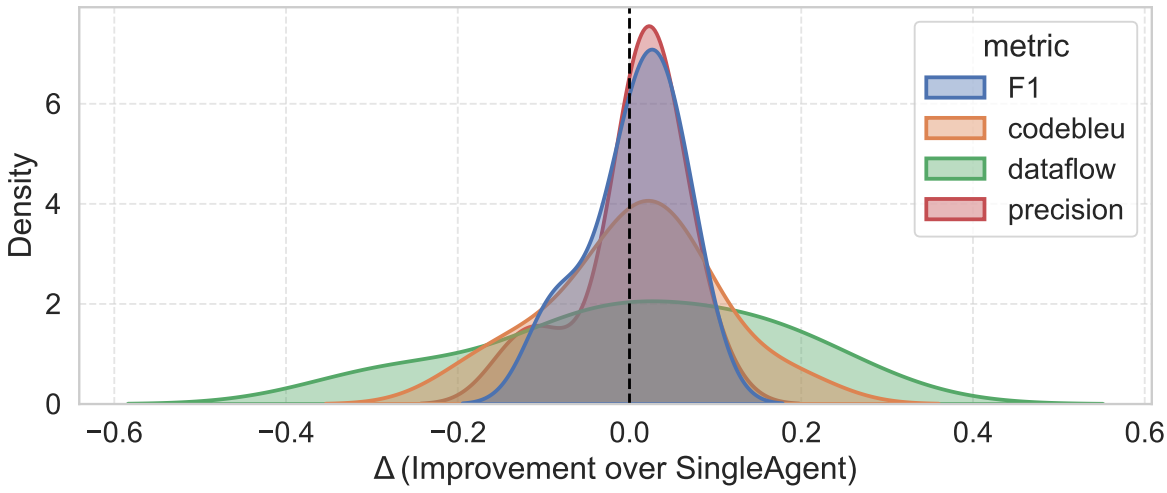


Figure 4.5: Distribution of Pairwise Differences Across Metrics. All metrics show a slight rightward shift (> 0), reflecting consistent but modest improvements of Multi-Agent over Single-Agent architectures.

RQ3: How do notebook size and complexity affect the code–sketch alignment of the generated sketches? In **RQ3**, we study the effects of scale and complexity in code: how the complexity of the code affects the alignment of the sketches VisDocSketcher generates, focusing on two factors: file size in *Number of Lines of Code* (LOC) and code segmentation measured as the *Number of Code Cells* (CC). We control the experiments for the developers' Kaggle Performance Tier (PT) as a proxy for their expertise.

To better understand how notebook complexity affects code–sketch alignment, we first built a general regression model that estimates the main effects of complexity features. We then extended this model with interaction terms to assess whether these effects differ between the single-agent and multi-agent architectures.

Although CodeBLEU scores are often low in absolute terms due to their strict enforcement of structural similarity, our analysis focuses on the relative differences across complexity levels and model variants. We use the *Dataflow* component of CodeBLEU as our dependent variable, since it best captures the semantic structure of a notebook by reflecting how variables and operations are connected. This aligns with our goal of representing the code's high-level workflow. The *CodeBLEU-Dataflow* score shows a strong negative correlation with notebook complexity, -0.46 with lines of code (LOC), and -0.24 with number of code cells (CC), being a first indicator that as notebooks grow more complex, maintaining accurate relationships in the sketches becomes increasingly difficult.

As shown in Table 4.4, both models explain a similar proportion of variance in the CodeBLEU-Dataflow metric (0.35 - 0.37). Given the complexity of factors influencing the generated sketches and the limited set of predictors, explaining over one-third of the variance is a meaningful result.

Results. In both models, the number of lines of code (LOC) and the number of code cells (CC) show statistically significant negative associations with the *CodeBLEU-Dataflow* score. These effects are observed in both architectures, with ($p < 0.001$ for LOC; $p < 0.01$ for CC) providing

Table 4.4: Regression Results: Dataflow and Notebook Complexity.

Variable	General Model			Interaction Model		
	Coef.	**Std. Err.**	***p***	**Coef.**	**Std. Err.**	***p***
Intercept	0.2654	0.0083	***	0.2688	0.0107	***
multi	—	—	—	-0.0074	0.0168	
PT=1	-0.0074	0.0094		0.0041	0.0123	
PT=2	-0.0167	0.0085		0.0007	0.0110	
PT=3	-0.0146	0.0104		-0.0029	0.0135	
PT=4	0.0161	0.0125		0.0128	0.0162	
PT=5	-0.1500	0.0103	***	-0.1550	0.0132	***
PT=1 × multi	—	—	—	-0.0268	0.0188	
PT=2 × multi	—	—	—	-0.0437	0.0171	*
PT=3 × multi	—	—	—	-0.0279	0.0208	
PT=4 × multi	—	—	—	0.0080	0.0252	
PT=5 × multi	—	—	—	0.0124	0.0209	
LOC	-0.0008	0.0000	***	-0.0009	0.0001	***
LOC × multi	—	—	—	0.0001	0.0001	
CC	-0.0012	0.0003	**	-0.0010	0.0004	**
CC × multi	—	—	—	-0.0003	0.0006	
R^2	0.3535			0.3697		
Adjusted R^2	0.3504			0.3631		

Note: $*$ $p < .05$, $**$ $p < .01$, $***$ $p < .001$

strong evidence of their influence. The performance Tier (PT) also exhibits a significant effect, particularly for the highest tier (PT=5), which shows a strong negative coefficient, suggesting that code authored by the most experienced developers is harder for the system to visualize, resulting in a **0.155 decrease** in the dataflow score compared to lower tiers.

Finding 4: Alignment Declines with Notebook Complexity

Code–sketch alignment declines with increasing notebook complexity. **Every additional 100 lines of code is associated with a 0.08 reduction in CodeBLEU-Dataflow, and every 10 additional code cells with a 0.012 reduction** (regression intercept: 0.27). Code by the most experienced developers is significantly harder to visualize, with a 0.155 lower dataflow score.

While the absolute coefficient values may initially appear small, they are meaningful relative to the scale of the score: the regression intercept is 0.27, so an increase of **100 lines of code corresponds to a 0.08 drop** in the dataflow score — roughly 30% of the intercept value. Realistic notebooks easily exceed these size differences, reinforcing the practical relevance of LOC and CC as complexity indicators.

One contributing factor is that longer prompts can degrade the reasoning capabilities of

LLMs. To investigate whether separating concerns can help address this issue, we examine the interaction effects. The interaction terms `LOC×multi` and `CC×multi` are not statistically significant, suggesting that the influence of notebook size and complexity on dataflow accuracy is consistent across both single-agent and multi-agent conditions. This indicates that the multi-agent setup neither amplifies nor mitigates the degradation in code–sketch alignment associated with increased complexity.

Finding 5: Equal Sensitivity Across Architectures

Interaction terms between system type and complexity are not statistically significant (p-value > 0.1). **Notebook complexity has a similar negative effect on code–sketch alignment for both single-agent and multi-agent systems.**

Rendering Statistics. We also examined rendering success rates on the large dataset, as its increased complexity may prevent agentic models from generating valid Mermaid files, leading to rendering errors. We found that the single-agent system successfully generated 999 Mermaid files, of which 849 were rendered without compilation errors (**84.9%**). Unlike a fixed linear pipeline, the multi-agent system operates through flexible stages coordinated by the supervisor agent. This agent decides which intermediate steps to include based on the complexity of the code. For instance, it may skip or repeat some of the stages. While this flexibility enables adaptive behavior, it also introduces variability in the outputs due to both agent failures and supervisor decisions. In the large dataset, the multi-agent system generated 996 JSON reports and 990 initial sketches. Of these, 87.1% (863) were repaired. 920 were further processed to include visual elements. Ultimately, in the multi-agent case, **74.4%** of the sketches were rendered successfully without errors, producing valid visual representations. Although the multi-agent variant renders fewer valid sketches due to its collaborative nature, the sketches it does produce achieved somewhat higher alignment scores in RQ2.

4.5 Discussion

In this section, we present the practical implications of our work in academic and industrial settings.

Research. The evaluation framework opens new directions for studying AI-generated visualizations across domains such as software architecture, data pipelines, and educational tooling. It enables systematic analysis of sketch fidelity and lays the groundwork for cross-domain adaptation, since the evaluation framework could be extended to other structured artifacts (e.g., UML, database schemas). Researchers can also explore collaborative settings where structured round-trip metrics help humans intervene effectively during diagram refinement.

Code Assistants Development. Integrating an automatic evaluation mechanism such as AutoSketchEval can serve two purposes: (1) guiding the assistant itself to iteratively improve the generated visualizations, and (2) providing developers with a confidence score in the UI that reflects how faithfully the sketch represents the underlying code. Furthermore, providing transparency through visual explanations allows users to verify what the AI has inferred about

their code. Displaying fidelity metrics in the UI could improve trust calibration: developers could see when AI-generated diagrams are more trustworthy, promoting safer adoption of LLM-driven tooling.

Maintenance and Software Evolution. As systems evolve, manually updating visual documentation is error-prone and often neglected. Our approach provides a path toward self-updating visual documentation that detects when diagrams fall out of sync with code, proactively signaling when updates are required. This could support long-term maintainability: consistent visual documentation that remains aligned with code could reduce onboarding effort for new developers and serve as a shared understanding artifact for teams. Whether these benefits materialize in practice remains to be validated with users.

Education and Learning. The round-trip framework can also serve as a training signal for learners: diagrams or code that yield lower reconstruction scores may indicate gaps in comprehension or misalignment between a student's mental model and the actual program behavior. This creates opportunities for targeted feedback, where students can iteratively refine the code and the diagram to improve conceptual understanding. Educators and tutoring systems can leverage this signal to monitor progress and reinforce correct mappings between abstract dataflow concepts and concrete implementations, supporting a more active and visualization-driven approach to programming education.

4.6 Threats to Validity

External Validity. Our study is scoped to data-science code in Jupyter notebooks, an important part of Software Engineering [57, 58]. The structure and abstraction of notebooks differ significantly from full-scale software systems, where diagrams often reflect complex architectural layers, components, and long-term evolution; generating a dataflow diagram from a single notebook is fundamentally different from producing architectural views of a distributed system. Although the method itself operates on general code properties (variables, functions, control flow) rather than domain-specific libraries or APIs, we provide no evidence of its effectiveness outside notebook code, and whether the approach transfers to other contexts is an open question for future work.

Internal Validity. The framework's scores depend on the choice of the Sketch2Code model and of the reconstruction metrics: a different decoder or a different metric could rank systems differently. Our robustness analysis across three LLM providers (Section 4.4) reduces this concern for the model choice, but does not eliminate it. The quality of the ground-truth code–diagram pairs also directly affects the validation of the framework; we used expert-authored artifacts from peer-reviewed work [39, 81], but they originate from a single user study and may carry its biases.

Construct Validity. The validation in RQ1 measures one specific property: whether alignment scores are high enough on a diagram paired with its own notebook, and low enough on a diagram paired with a randomly selected notebook, for the two conditions to be told apart. This is a coarse property. The negative cases are drawn from unrelated tasks, which makes them comparatively easy to separate. We do not show that the scores can rank two plausible sketches of the *same* notebook or that a score of a given magnitude corresponds to a diagram a developer would

find trustworthy. Passing our validation is therefore a necessary condition for the framework to be meaningful, and it remains some distance from showing that the alignment it measures is accurate enough to be useful for a concrete purpose, such as driving a confidence indicator in a code assistant or gating the visual documentation attached to a commit. Establishing that requires a study in which developers work with the sketches and their scores, which we leave to future work. Reconstruction-based metrics measure how completely a sketch preserves the code's structure; they do not measure abstraction level, layout clarity, aesthetics, or usefulness to developers (see Section 4.2.3). During development, we informally inspected generated diagrams in the ground-truth dataset and observed that baseline sketches were often oversimplified and missed relevant semantics, whereas VisDocSketcher produced clearer structure; however, this inspection was performed by the authors and is not a substitute for a systematic human evaluation. Whether the generated sketches actually help developers build accurate mental models remains unvalidated.

4.7 Summary

We introduced a novel LLM agentic framework for generating high-level visual documentation of code, with a focus on dataflow reconstructions in the data science domain. Our agentic approach decomposes the generation task across specialized agents, coordinated by a central supervisor, enabling modular and interpretable sketch construction. To address the lack of evaluation standards in assessing sketches, we proposed a novel, automatic evaluation framework grounded in established code-to-sketch similarity metrics. Using this framework, we conducted a comparative analysis between our approach and a template-based baseline. Our approach improves the baseline results by 26.7% in the mean of our metrics (CodeBLEU and CodeBertScore). The multi-agent system outperformed the single-agent system in 59.3% of cases, but at the cost of higher processing time (11.3× slower on average). Quantitative results indicate that sketch quality degrades as code complexity increases, with significant correlations between quality scores and structural notebook features such as lines of code and number of cells. This highlights both the limitations of current generation techniques and the importance of robust evaluation methods. Our work opens promising directions for future research in visual generation for software engineering, including improving scalability, enabling richer forms of sketches, and extending to other domains such as software architecture or scientific workflows.

4.8 Takeaways

The results from this work contribute to the thesis on two levels: they validate the externalization direction of the human–AI interaction loop, and they surface a structural limitation that directly motivates the co-evolution challenge. The AutoSketchEval framework can successfully distinguish code-aligned from non-aligned sketches, without requiring any ground-truth diagrams. This is not merely a methodological result: it establishes that the alignment between code and informal visual artifacts is a measurable, tractable property, which is a prerequisite for any claim made in the remainder of this thesis. VisDocSketcher improves over the static-analysis-based baseline across code-level metrics, suggesting that AI systems can externalize structural and

semantic code reasoning in visual form, supporting the thesis claim that the AI-to-human direction of the interaction loop is feasible. Most consequential, code–sketch alignment degrades with code complexity. This finding reveals a fundamental scalability limit of full-regeneration approaches: as codebases grow, generating an entirely new diagram from scratch becomes both less reliable and less practical. Incremental, localized updates, the core mechanism presented in *code evolution*, are not merely a usability preference but a technical necessity for maintaining visual alignment at scale.

It is important to note, however, that AutoSketchEval measures only one side of the alignment problem: how faithfully a generated sketch reflects the underlying code. What remains open is addressed in the following section. It is the complementary question of *sketch-to-mental-model* alignment: whether the generated visual representations actually help developers build accurate mental models of the code they are reading. Code-alignment is a necessary condition for a useful visualization, but not a sufficient one; a sketch can be structurally faithful yet cognitively ineffective if it does not match the representations developers naturally expect. Closing this gap is what transforms VisDocSketcher from a generation system into a comprehension tool, and it is the focus of the extended evaluation.

Chapter 5

Code Explanation: Diagram Usefulness

The rapid adoption of AI code-generation tools, such as GitHub Copilot or Claude Code, has shifted the primary cognitive bottleneck in software development. Developers shift from active authors to code reviewers responsible for evaluating, validating, and accepting code they did not write [10, 154].

Prior work on AI-assisted coding has focused on *what* bugs and security issues these tools introduce and how often they occur [89, 119]. Less attention has been paid to a complementary problem: how to support developers in *understanding* and accountably reviewing AI-generated code that appears correct and executes without error. A distinct risk of AI-generated code is the case where it is *plausible and executable but methodologically wrong*. Agentic LLMs generally iterate on the code until it executes and passes model-generated tests. As a result, runtime exceptions in their code are rare, but semantic errors remain common. Machine learning workflows in Jupyter notebooks are a prime example of a domain where the code may run without errors while silently using wrong data or processing it in unintended ways, which results in flawed models.

Unlike a crash or a failing test, a methodological flaw produces a valid notebook and plausible results, but misleading downstream conclusions. Empirical studies show that developers can be dangerously overconfident in inaccurate AI-generated code, accepting suggestions in roughly half of observed instances even when they are incorrect [89, 119]. Jupyter notebooks amplify the review challenge. Their cell-by-cell structure fragments program logic across independent execution units and provides no structural overview of the complete pipeline [16, 106]. A reviewer must mentally reconstruct the data flow from raw input to model evaluation before they can meaningfully assess correctness. Reviewers rarely read every line; they work at the pipeline level: which data feeds which step, what is fit to what, and what is evaluated against what [4, 92]. This spatial information is better expressed using two-dimensional representations rather than sequential textual documentation.

Visual artifacts are closely tied to developers' mental models. Practitioners have long used informal sketches and whiteboard diagrams to reason about design, plan implementation, and communicate ideas [6, 19, 75, 120]. Cognitive psychology points out the underlying mechanism that explains this preference: humans process and remember visual information more effectively than text. The *picture superiority effect* [133] and dual-coding theory explain that visuals engage verbal and visual memory simultaneously, strengthening understanding and recall [86, 87]. While

textual explanations may feel overwhelming and make developers overlook important aspects of the pipeline, visual artifacts can facilitate this process by providing an immediate overview of the notebook organization and key operations [80].

While prior work establishes that visuals aid comprehension and that developers reason through sketches, whether auto-generated visual overviews improve silent flaw detection in unfamiliar code remains untested.

In this chapter, we address the question of how to support accountable review of data science code using *informal* visuals. We present JupyterDraw, a tool that automatically generates whiteboard-style diagrams from Jupyter notebooks, and a controlled experiment with 26 developers to directly test whether such diagrams improve developers' ability to detect methodological flaws in the code.

5.1 Contributions

Our work focuses on the data-science domain and Jupyter notebooks, where developers reason about data flows, transformations, and ML components across cells [57, 58]; We conduct a within-subjects controlled experiment in which developers review AI-style ML notebooks containing planted methodological flaws under a textual (T) and visual (V) overview of the pipeline. This work answers the following research questions:

- **RQ1 (Detection).** Do automatically generated visual overviews (V) improve developers' detection of silent methodological flaws in AI-generated ML notebooks, relative to textual summaries (T)?
- **RQ2 (Mechanism).** Through which representation do developers actually detect flaws, and what properties of the overview produce the detection advantage?
- **RQ3 (Perspectives).** How do developers perceive the value, reliability, and future role of visual overviews in AI-assisted code review?

The visual overview significantly improves flaw detection, raising the share of reviews catching both planted flaws from 29% to 67% (RQ1). It becomes participants' primary detection channel, surfacing problematic flow connections not easily seen in text (RQ2) and developers prefer it as a trusted supplementary aid, contingent on fidelity to the code (RQ3).

The contributions of our work are as follows:

- We develop JupyterDraw, an MCP server that provides tools to automatically generate whiteboard-style pipeline diagrams from Jupyter notebooks to support structural review of AI-generated ML code.
- We conduct the first controlled experiment measuring the effect of auto-generated visual summaries on silent flaw detection in data science code, with 26 developers.
- We provide empirical evidence that visual overviews are significantly more helpful than textual ones, eliciting more structural reasoning, and improving overall flaw detection.
- We derive design implications for integrating visual summaries into AI-assisted development workflows, identifying opportunities to improve developer oversight and accountability in code review.

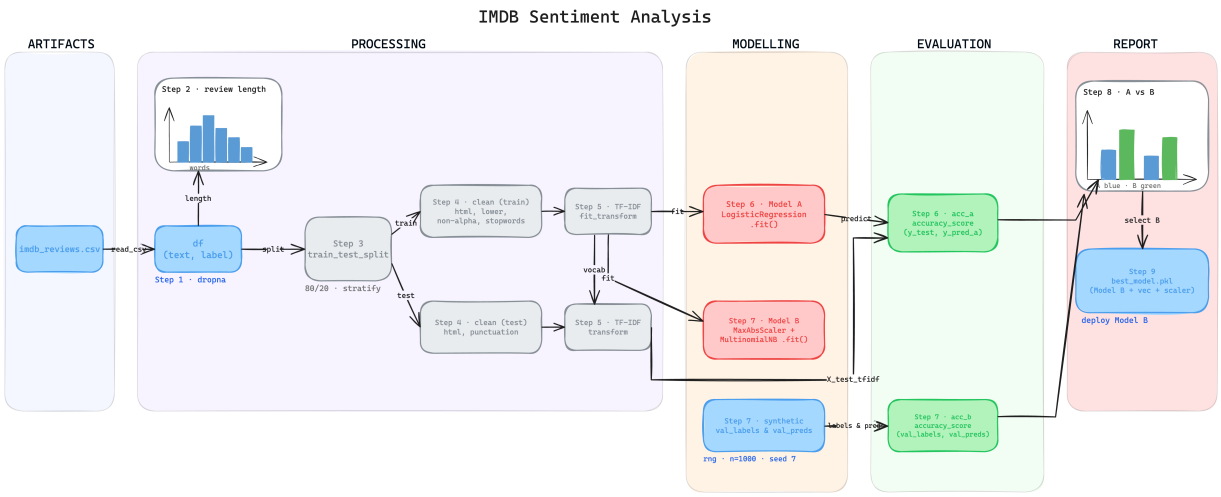


Figure 5.1: Example of a Jupyter notebook overview generated using JupyterDraw. These are the overviews shown to participants in the visual (V) condition.

5.2 Related Work

Understanding code is an important part of software development. Our work builds on three lines of research: how developers *review and comprehend* code they did not write, how *visual artifacts* support that reasoning, and how *tooling* can be used for that end.

Code Comprehension and Review. During code reviews, developers must form accurate mental models of how the code works. Program comprehension of new code is structural: developers use abstractions and comprehend new code at the level of data flow, avoiding reading every line if a higher-level alternative is available [101, 102]. Xia *et al.* show that comprehension dominates reviewers' working time [137]. While finding bugs is the primary motivation for code reviews, Bacchelli and Bird found that the main outcome shifts from defect-finding to comprehension and that "understanding is the key of any review" [4]. This understanding is harder to achieve when the reviewer is seeing the code for the first time and did not write the code themself. This problem also translates to code written by an AI assistant: the developer becomes a reviewer of code they did not author [10], and is prone to over-trust plausible, executable output [59, 119].

Visual representations of code can help bridge this comprehension gap. Controlled studies show that enriching how code is presented reduces the time developers need to answer questions about it, without overloading them [3], and that code visualizations help newcomers orient themselves and locate defects in unfamiliar systems [88]. This benefit is not only about *content* but is also *spatial*: not just what is shown, but also how it is laid out. Kim and Kook, in a controlled experiment on UML class-diagram layout and complexity, found that diagrams measurably improve comprehension [56], and eye-tracking evidence shows that layout governs how readers visually search a diagram [146]. Our work builds on this lesson: *spatial form* shapes understanding. We explore how AI-generated *informal* visual notations affect code comprehension and, consequently, a developer's ability to catch silent flaws, moving toward a future where AI communicates through the sketch-like representations developers already use to reason visually.

Visual Reasoning in SE Research in cognitive psychology demonstrates that people process and remember visual information more effectively than textual information, a phenomenon known as the *picture superiority effect* [34, 133]. Dual-coding theory explains that visuals engage both verbal and visual memory systems simultaneously, strengthening understanding and recall [86, 87]. Software developers, whose work involves abstract, multi-level structures, are expected to perform such reasoning through text-based interfaces that provide little visual grounding [51]. This is particularly challenging because developers must maintain accurate *mental models* of software, internal representations of how components interact, how data flows, and how behavior emerges from code structure [64].

Studies on cognitive load in SE confirm that reducing cognitive burden (for instance, by providing high-level abstractions like software models rather than requiring developers to reconstruct structure from raw code) improves comprehension performance and reduces error rates [42]. Visual explanations are not merely supplementary: they are closer to the representations developers naturally build in their minds, making it a more cognitively efficient medium for communicating code structure than text alone.

Previous work has investigated how developers use informal sketches to support software development. LaToza *et al.* [64] showed that developers construct complex mental models to understand code, emphasizing the need for tools that link informal sketches to code artifacts. Mangano *et al.* [75] identified that developers use low-detail abstractions and mix notations when sketching. In the same line, Walny *et al.* [120] traced how sketches evolve across tasks, showing that informal diagrams serve not only as design tools but also as memory aids and communication devices. Baltes *et al.* [6] demonstrated that most sketches are informal, abstract, and created on analog media, proposing tools like SketchLink [7] and LivelySketches [8] to bridge analog and digital workflows. Building on Mangano's observations regarding the frequent use of mixed notations in developer sketches, Gomes *et al.* [39] explored the potential of LLMs as *Visual Code Assistants*: tools that help express intent visually and generate corresponding code. While this research emphasized translating visual representations into code, the reciprocal capability, explaining code through visuals remained underexplored. Xie *et al.* , in WaitGPT [138], investigated this direction in a study with 12 participants. They demonstrate that visual explanations can improve error detection and their overall confidence in AI-generated code.

Unlike prior visualization-based systems, which embed visualization into a live LLM workflow for steering and monitoring analysis during coding, our work studies whether a generated informal sketch can help developers audit completed notebooks for silent methodological flaws rather than UI interface perspective.

Notebook Tooling and AI-Generated Visuals. Several tools have been proposed to help analysts navigate the complexity of computational notebooks. In Albireo, Wenskovitch *et al.* [128] generate interactive force-directed graphs that expose cell dependencies and variable flows, supporting self-reflection and collaboration. Ramasamy *et al.* [97] propose a graph-based method that visualizes forking paths in data science workflows, demonstrating improved notebook comprehension in a controlled experiment. A systematic survey of 163 notebook visualization tools [125] identifies key design trade-offs, including the tension between visualization richness and notebook integration.

A complementary line targets these flaws *automatically*: Yang *et al.* [141] statically detect data leakage in notebooks, flagging a fixed catalog of known anti-patterns, whereas JupyterDraw

surfaces an informal overview and leaves unanticipated flaws for the human to judge. This human-in-the-loop framing connects to human–AI verification, where developers over-trust AI output during review [59]: JupyterDraw redirects attention rather than issuing a verdict the developer may accept uncritically.

The need for such tooling is motivated by empirical evidence. Rule *et al.* [106] analyzed over one million notebooks and found a persistent tension between exploratory and explanatory use: one in four notebooks contained no explanatory text, leaving structure implicit and opaque to newcomers. Chattopadhyay *et al.* [16] corroborate this in a large-scale need-finding study with data scientists, identifying code navigation and understanding data flow as prominent pain points across all stages of the analytics workflow. At the infrastructure level, Jiang *et al.* [50] present a lightweight algorithm to extract notebook structure as labeled dependency graphs, automatically tagging cells with ML pipeline stages and proposing downstream applications. Large-scale studies have also documented the poor structural quality of notebooks in the wild [93], reinforcing the practical need for these kinds of tools, which can extract and communicate notebook intent at a glance.

In contrast to these tools, which visualize the internal structure of notebooks (cell layout, execution order), JupyterDraw generates diagrams that visualize data science pipelines in a way that is intended to approximate developers' mental models in this domain.

5.3 JupyterDraw: AI-Generated Diagrams

To study our theory, we built JupyterDraw. It can be used inside the AI assistant a developer already uses (any MCP-compatible host, e.g. Claude Desktop, Claude.ai, or Claude Code) rather than as a standalone application or editor extension. This tool takes a Jupyter notebook and produces an informal diagram of its pipeline: data sources, transformations, models, and evaluation. JupyterDraw is an *MCP App*: a Model Context Protocol server that exposes notebook-diagramming tools and renders the diagram as an interactive in-chat widget using Excalidraw.[1] In this section, we describe the prototype and a motivating example. The tool is used to generate the visual overviews shown in the "V" condition of our study.

Interaction. We want participants to focus on *reading and judging* an AI-generated notebook through its diagram, not on operating JupyterDraw. The interaction we seek to study is how a developer can grasp what a notebook does from a single picture, the same way they would skim a colleague's whiteboard sketch before trusting their work. While the user interface and human-AI interaction dynamics is not the main focus of this study, we designed the current system around Excalidraw, an environment that mimics whiteboard-style sketching, leaving this dimension open for future investigation. The choice of this environment is motivated by three considerations: (1) it supports diagram generation through a structured textual format (JSON), enabling integration with LLM-based pipelines, even without image generation capabilities; (2) it exposes the AI agent's spatial reasoning ability, as the agent must determine the positioning of individual elements — text, arrows, boxes, and visualizations — in a way that mirrors human sketching behavior; and (3) it preserves human agency, allowing users to directly manipulate

[1] `https://excalidraw.com/`. We build on top of the Excalidraw MCP: `https://github.com/excalidraw/excalidraw-mcp/`

and refine the generated diagram, creating a foundation for future research on visual human-AI collaboration.

Generation. Generation is driven by the host's language model, *steered* by the MCP server. To generate the diagrams for our study, we need an LLM that is good at understanding Python code and generating structured JSON. For that reason, we chose Claude Opus 4.8, as this model is the best performing in agentic coding tasks (SWE Bench) [143]. JupyterDraw injects concise instructions into every conversation and exposes a `read_me` tool that returns the full Excalidraw element-format reference together with the Jupyter diagramming rules (lanes, role colors, annotations, and layout anti-patterns). The model analyzes the notebook under these rules and emits the diagram as a JSON array of Excalidraw elements, which it passes to the `create_view` tool. For a controlled experiment, we need the generated visuals to be similar for all notebooks. Fixing the visual vocabulary in the instructions, and leaving only content selection to the agent, is what keeps the diagrams consistent across notebooks. The instructions injected into the prototype and used in the user study are the following (simplified prompt):

> You are an Excalidraw diagram assistant specialized in Jupyter notebook diagramming. You are given a data-science notebook. Draw its ML pipeline: one box per meaningful step — datasets, cleaning . . . output plots — colored by type (data/process/model/eval/output) and grouped into subgraphs, connected left-to-right with labeled arrows. Include only what tells the pipeline's story (which data trained which model, on which split, scoring what metric), sketch a rough chart inside every plot box, and skip imports and boilerplate.

Rendering. The `create_view` tool renders the element array as an informal diagram scene: boxes colored by role, arranged in the pipeline lanes, and connected by directed data-flow arrows inside the in-chat widget. It supports an *instant* mode that draws the whole scene at once and an *animated* mode that streams elements with draw-on animations and camera pans, plus a *file* mode that returns a portable `.excalidraw` file. After rendering, the widget captures a screenshot of the diagram and returns it to the model, enabling a self-review loop in which the model inspects its own output and repairs overlaps, lane violations, or occluded labels before presenting the final diagram.

5.3.1 Motivating Example

This study recreates scenarios in which developers must understand and evaluate code they did not write. Specifically, we study this challenge in data science workflows involving Jupyter Notebooks. The most prominent contemporary example is AI-generated code, but the scenario extends naturally to third-party contributions such as intern submissions, student assignments, or externally sourced notebooks. To simulate AI assistants' explanations of code, participants are provided with generated visual or textual overviews, giving them a first-hand understanding of what the code does. Both the overviews and the notebooks are pre-generated to guarantee all the participants have the exact same conditions. We ask each participant to behave as they would when an assistant hands them a draft they intend to ship: look briefly at the output cells and high level code structure, then decide whether it is ready.

Figure 5.1 shows the visual overview JupyterDraw produces for one such notebook. In this example, we present a notebook that loads a dataset, preprocesses the text, vectorizes it, trains classifiers, and reports their performance. The whole code runs with no bugs or warnings,

Artifacts

Inputs

- `imdb_reviews.csv` — raw review dataset with `text` and `label` columns

Outputs

- `best_model.pkl` — serialized Model B bundle (MultinomialNB + TF-IDF vectorizer + MaxAbsScaler), flagged for deployment

Pipeline Steps

Step 1 — Load Data

- From `imdb_reviews.csv` → `read_csv` + `dropna` → `df (text, label)`

Step 2 — Exploratory Analysis

- From `df` → compute `length` of each review → plot **review length** histogram (words)

Step 3 — Train/Test Split

- From `df` → `train_test_split` (80/20 · stratify) → `train`, `test`

Figure 5.2: Example of the textual condition (T) overview.

and the output cells look convincing, reporting good results for model B. However, it contains two critical flaws that undermine the validity of all results: the preprocessing pipeline applies inconsistent transformations to training and test data, and Model B's evaluation uses entirely synthetic (fabricated) metrics instead of real predictions. Note that although flaws of this severity are unlikely to appear in production code given the capabilities of modern LLMs and their access to full code context, they serve a deliberate purpose in this study. They are designed to represent the class of subtle, high-impact errors that are plausible enough to go undetected under casual review, yet consequential enough to fully undermine a notebook's scientific validity. Within a 15-minute code review task, and for participants unfamiliar with the codebase, these flaws approximate the difficulty of real-world silent failures in more extended reviews.

The diagram, however, exposes the structure that the output cells hide. Because causal dependencies are drawn as arrows, the order of operations is visible at a glance: train and test go through different pipelines, and model B is never used for evaluation. In the textual version (Figure 5.2, the same information is shown to participants, removing the visual and spatial component.

In real scenarios, the diagram is useful precisely because developers do not always read the textual explanations an assistant generates and tend to trust the output it shows. A quick, scannable picture of the pipeline lets them catch the structural mistakes that plausible-looking results hide. It saves time by making the design clear enough to discuss at a higher level of abstraction, discussing decisions with the agent, instead of debugging the code cell by cell.

5.4 Methodology

The goal of our study is to investigate the impact of AI-generated visual overviews of code on developers' ability to detect silent flaws and how these visuals affect the overall code review experience. Specifically, we explore whether they reduce cognitive effort, accelerate comprehension, and make the review process more approachable compared to reading source code or textual summaries.

Our population of interest is individuals with background knowledge in Python and ML, independent of their specific level of expertise. Our sample comprises individuals from academia and industry who have previously worked with Jupyter notebooks and understand core ML and data science concepts.

5.4.1 Recruitment

Initial candidates were identified from personal contacts and through a LinkedIn post with an interest form link. To further expand recruitment, we asked enrolled participants to nominate additional candidates who met the eligibility criteria, following a snowball sampling approach. Each participant received a $20 Amazon gift card as compensation for their time, with an extra performance-based bonus of up to $10 (disclosed at the end).

We recruited 26 participants (13 male, 13 female) from different institutions in academia (Carnegie Mellon University, MIT, Brown University, UC Irvine, Instituto Superior Técnico and Univ. of Porto) and industry (Amazon, Apple, Google DeepMind, Cohere, DoorDash, Skilly, Bandora, and Ering). Regarding educational levels, 17 participants were pursuing or had completed a doctorate and nine a master's degree. Regarding their work environment, 22 were in research positions (e.g., research scientists, technical staff at AI companies), and four were in applied software engineering or product roles.

5.4.2 Protocol

Our IRB-approved protocol encompasses four stages after the participant fill in the interest form and demographics survey: 1) Tutorial, 2) Review Session 1, 3) Review Session 2, and 4) Post-Study Survey. The process was designed to take approximately 45 to 60 minutes. The study was conducted in-person or virtually, and in both cases we used in-browser GitHub Codespaces for the tasks.

Demographics Survey. To express interest in participating, candidates complete a Google Forms survey that collects demographic information alongside details about their experience with Python, Jupyter notebooks, and ML. This information provides context for understanding participants' backgrounds, which can influence their review behavior. Upon agreeing to participate, candidates receive a link to the consent form and are asked to schedule a session with the research team to complete the controlled experiment.

Tutorial. Before reviewing the experimental notebooks, participants complete a short training task with researcher guidance that familiarizes them with the review protocol, the note-taking shortcuts (`.b` — `.n` — `.e` VSCode snippets for creating the *begin*, *notes*, and *end* of review timestamps), and the granularity of methodological flaws they should be alert to.

Notebook Review. Each participant reviews two AI-generated Jupyter notebooks and decides for each whether the code is ready for deployment. Alongside the notebook code, participants are provided an *overview* of the pipeline. Depending on the assigned condition, the overview is either: **V (Visual):** an auto-generated diagram rendered by JupyterDraw; **T (Textual):** an auto-generated, structured textual description of the notebook.

To make the comparison fair across conditions, we guarantee that both overviews are presented at the same level of abstraction. The notebook code is identical across conditions; only the overview representation differs.

While reviewing, participants take timestamped notes directly inside the notebook using the provided VSCode snippets. A timer keeps track of each review to inform participants about the 15-minute time limit during the session.

After completing each notebook review, participants fill out a short form for that task, capturing their deployment decision, NASA-TLX workload ratings, and a per-flaw attribution Likert item, rating the contribution of the code, markdown, textual and visual overviews to identifying each potential issue.

Post-Study Survey. After both notebooks are reviewed, participants complete a post-study survey assessing their overall preferences and perceptions. This includes comparative helpfulness ratings of the visual vs. textual overview, perceived accuracy of the auto-generated diagram, an overall condition preference, and open-ended questions on what helped or hindered their review, what was confusing or missing, and how they envision this type of tool fitting into their daily workflow.

5.4.3 Tasks

We designed the tasks to simulate a realistic deployment scenario in which a developer is asked to audit committed code. An example of such a scenario would be a pull request from an intern or a homework submission from a student, who in turn used AI to generate some of the code. Each task is framed as a review process: the participant reviews the notebook and decides whether it is ready for production, identifying possible flaws, and annotating the notebook to be fixed later by the student.

We used two notebooks, each targeting a different ML application domain adapted from Kaggle datasets:

- **Notebook A - IMDB Sentiment Analysis [73]**: A binary text classification pipeline using TF-IDF and a Naive Bayes classifier, applied to movie reviews.
- **Notebook B - Blower Energy Forecasting [74]**: A regression pipeline using Linear Regression and a second model to forecast energy consumption from sensor data.

Each notebook contains **exactly two planted methodological flaws**. These silent errors cause the code to execute cleanly and produce plausible, optimistic metrics, yet represent genuine deployment problems. In both cases, these flaws lead to the deployment of a wrong ML model in the given scenario. Table 5.1 summarizes the four planted flaws.

Flaws are methodological and structural, not runtime bugs, so every cell executes properly. The categories represented are drawn from established taxonomies of leakage and reproducibility failures in ML-based science [52]. The two notebooks were designed to be of comparable

Table 5.1: Planted methodological flaws per notebook.

Notebook	ID	Problem Description
IMDB Sentiment	P1	Inconsistent train/test preprocessing. Different functions are applied to train and test, causing mismatched vocabularies.
IMDB Sentiment	P2	Model B (Naive Bayes) is not evaluated. Synthetic generated labels and predictions are used instead of real metrics.
Energy Forecasting	P1	Training features and target are random noise rather than the real dataset, invalidating all learned patterns.
Energy Forecasting	P2	Models are evaluated on different data; Model A is tested on training data rather than the held-out batch.

complexity and diagram density to avoid confounding the condition effect with task difficulty.

Counterbalancing. To control for order effects and avoid confounding condition with notebook, we used a **within-subjects, 4-group Latin square** counterbalancing design. Participants were assigned to one of four groups in round-robin order. Each group determines which notebook is reviewed first and under which condition, such that every participant reviews one notebook under V and one under T, and each notebook is reviewed equally often under each condition across the sample:

G1: IMDB-T → Energy-V G2: Energy-V → IMDB-T
G3: IMDB-V → Energy-T G4: Energy-T → IMDB-V

5.4.4 Evaluation

We evaluate our research questions with the results from a controlled, within-subjects experiment.

Code fidelity. To validate our results, we first inspect the alignment between code and diagram. Diagrams are subjective artifacts, and their faithfulness to code is not objective. We inspect each diagram manually for correctness. Given the deliberately low complexity of the 15-minute study tasks, establishing structural correctness is straightforward, and we found no errors. To quantify the faithfulness of the generated diagrams to the code, we complement our assessment with an LLM evaluation (Claude Opus 4.8) in which one model generates fine-grained questions and answers grounded in the notebook source code spanning five abstraction levels from low-level implementation details to high-level workflow. A second model, shown *only* the rendered diagram, answers them; scoring is exact-match against the code-derived ground truth. Across the two study notebooks (25 questions each), the diagrams answered 80% (IMDB) and 84% (Energy) of questions correctly, with *zero* incorrect answers: every miss was an explicit abstention where the diagram did not depict enough detail, rather than a misleading depiction. Fidelity was near-perfect at the pipeline-structure, methodology, and high-level workflow levels, and degraded only for the finest implementation details (e.g. exact hyperparameter values), which a structural diagram is not intended to convey. Additionally, we show *only* the diagrams to a third agent that performs the review and is able to identify *all* planted flaws. Together, these prove that the diagram is faithful to the code and that it has enough information for the participants to catch

the flaws. The same analysis is done for the textual overviews.

RQ1: Detection. The main outcome is how many of the two flaws a reviewer catches per task (0, 1, or 2). We compare V against T within each participant and report the average difference with a 95% confidence interval and effect sizes (d_z, rank-biserial r), backed by a Wilcoxon test.

We then confirm the result is stable across three models on the primary sample, each estimating the V−T effect a different way.

The first is a **linear mixed-effects model**, which accounts for the repeated-measures structure of our design. It predicts the number of detected flaws from three fixed effects: the *visual condition* (our variable of interest, which must reach significance to support our hypothesis), and two control variables—which *notebook* was shown and its *session position*—which we expect to be non-significant. To capture between-participant variability, we include a random intercept for each participant, allowing every participant to have their own baseline (i.e., a distinct starting point in how many flaws they detect). The model is specified as: (`n_flaws ~ condition_v + notebook_imdb + order2 + (1|participant)`) For this model we also report Nakagawa & Schielzeth's marginal R^2 (variance explained by the fixed effects alone) and conditional R^2 (fixed effects plus the participant random intercept).

The second is an **ordinal logistic model** (proportional odds), which respects the ordered $\{0, 1, 2\}$ outcome rather than treating it as interval. The third is a **complete-detection model** (GEE logistic), whose outcome is whether the reviewer caught *both* flaws—meaning they identified every issue and would deploy a correct model.

We further test significance under different eligibility cut-offs and when first- and second-task reviews are examined separately. Finally, we test *time* variables: how long reviewers take to reach the first flaw, the second flaw, and to finish—plus an *effective finish* that stops the clock at whichever comes first, the end of the review or the second flaw. This tells us both whether catching more flaws costs extra time and whether the visual overview actually makes reviewers faster.

RQ2: mechanism. We probe *how* the overview helps with three convergent measures: *perception* (helpfulness and accuracy rated 1–6, paired V vs T); *attribution*, for each *found* flaw, the participant rates it a four-level (No/Minor/Strong/Very Strong Contribution) rating of how much the overview or the notebook drove the catch, identifying the route through which detection occurred; and *workload* (NASA-TLX excluding physical demand).

RQ3: perspectives. We code six open-ended post-study themes (preference, where the overview helped, anything confusing/missing, workflow fit, desired changes, free remarks) into a thematic codebook spanning perceived value, trust, future role, and desired improvements, counting each label by distinct participants, and triangulate with in-notebook annotation volume and content.

5.4.5 Data and Artifacts

During the study, we collect both quantitative and qualitative data. Participants annotate the notebooks using timestamped notes (`.n` snippet), which are parsed by `extract_notes.py` into per-session and per-note CSVs. After each interview, the researcher inserts `[[P1]]`/`[[P2]]` markers into a participant's notes where a planted flaw was reported. To check the participants' intent regarding flaw identification, the experimenter discusses this with each participant during

the annotation process. Participants are incentivized to confirm their intent, as the bonus compensation is dependent on the number of flaws correctly identified. These markers are used as the source of detection ground truth. Form responses (NASA-TLX, attribution Likerts, deployment decisions) and survey responses (perception, preference, open-ended) are collected via Google Forms, filled by participants. All anonymized data and analysis scripts are publicly available.

5.5 Results

We report findings from $n = 26$ participants (52 review sessions). As reading a diagram presupposes a schematic mental model of ML workflows, we perform the quantitative analysis (RQ1 and RQ2) with participants who report *some* ML familiarity (greater than *None* = 1/6). RQ3 uses the data of all participants, as opinions from non-AI experts are also valuable for the qualitative analysis. Only two participants self-rated ML familiarity at 1 out of 6 (p05, p06).[2]

5.5.1 RQ1: Detection

A visual overview can only matter for accountable review if it positively impacts developers' understanding of code. We must first establish whether it improves the desired outcome: the number of problems a reviewer detects. We test this directly by comparing detection under the visual and textual conditions with **RQ1: Do automatically generated visual overviews (V) improve developers' detection of silent methodological flaws in AI-generated ML notebooks, relative to textual summaries (T)?**

Review Sessions. Sessions ran a median of 14 minutes (min 6.5, max 16.4), with no meaningful gap between conditions (T mean 12.4, V 12.9). Reviewers annotated as they read, leaving a median of **3 in-notebook notes per session** (mean 3.3, range 0–14), again comparable across conditions (T 3.12, V 3.50). Contrary to the identical time effort, detection diverged. Under T, reviewers caught on average **1.17** of the two flaws (88% caught at least one, but only **29%** caught both). Under V they caught **1.67**, with **100%** of sessions catching at least one and **67%** catching both. Of the 24 participants, 13 caught more flaws with the visual overview, 9 tied, and only 2 did worse Figure 5.3, left). We explore if the flaw gap is significant and whether it survives adjustment for notebook and task order (4 balanced groups, 6 participants each).

Paired Comparison. The paired difference is significant (Wilcoxon signed-rank $p = 0.005$) and large (Cohen's $d_z = 0.69$, matched-pairs rank-biserial $r = 0.75$), and its bootstrap 95% CI $[0.21, 0.79]$ lies entirely above zero (Figure 5.3, right).

Regression models. A linear mixed model with a per-participant random intercept estimates a V effect of +0.50 flaws ($p < 0.001$) (Table 5.2). This means that participants detected, on average, half a flaw more per session under the visual condition. The estimate is unchanged when we adjust for which notebook was shown and which task came first. Both covariates are negligible (notebook $\hat{\beta} = 0.00$, $p = 1.00$; order $\hat{\beta} = 0.08$, $p = 0.56$), confirming that the condition alone drives the difference. The fixed effects account for 18% of the variance in detection (marginal

[2]Interestingly, both rated the textual overview as more helpful (p05: Visual helpfulness 3 vs. Text 5; p06: rated both equally low (2). Without enough ML knowledge, diagram nodes and edges are arbitrary symbols rather than meaningful representations and so participants prefer code and text.

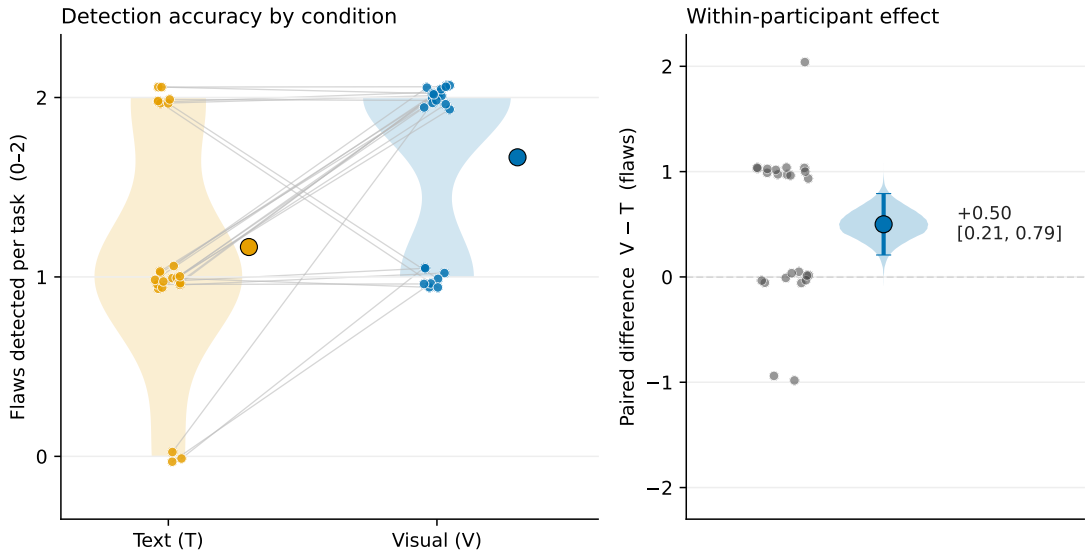


Figure 5.3: Flaws detected per task by condition (n = 24). Left: individual paired observations with bootstrapped condition means; right: the within-participant difference (V − T) with its bootstrap 95% CI.

R^2 = 0.18), rising to 33% once individual differences in reviewer ability are absorbed by the participant random intercept (conditional R^2 = 0.33). Two alternative specifications confirm the estimate: an ordinal proportional-odds model returns an odds ratio of 5.51 (95% CI [1.63, 18.59], p = 0.006), meaning the visual condition increases the odds of catching at least one flaw by a factor of five; a GEE logistic model of *complete detection* yields an odds ratio of 5.02 (95% CI [1.50, 16.79], p = 0.009), meaning the visual condition quintuples the odds of catching all flaws.

Finding 1 - Visuals Improve Detection

Developers detect more methodological flaws with an AI-generated visual overview than with a textual one (+0.50 flaws per task, d_z = 0.69, 95% CI [0.21, 0.79]; Wilcoxon p = 0.005). The share of reviews that caught both planted flaws rose from 29% to 67% (**+38 percentage points**).

Robustness. Across all ML-familiarity thresholds, the visual-condition effect remains positive and statistically significant. Refitting the mixed model on the full sample (n = 26) and on progressively stricter subsamples ≥ 2, ≥ 3, and ≥ 4v (n = 24, 22, 21) always yields a V coefficient between 0.39 and 0.50 with $p \leq 0.010$. No matter where we place the eligibility threshold, the visual overview consistently improves flaw detection.

Time. To test performance, we use the same model (`(outcome ~ condition_v + notebook_imdb + order2 + (1|participant))`) with four time-related variables: time to the first flaw, time to the second flaw, time to finish, and an *effective finish* (the earlier of session end or catching the second flaw).

As shown in Figure 5.4, three of the four effects are small with a 95% CI spanning zero (all

Table 5.2: Linear mixed model predicting the number of detected flaws (0–2) per task, with a random intercept per participant ($n = 24$). The visual condition increases detection net of notebook and task order.

Predictor	**Coeff.**	**95% CI**	**p-Value**
Intercept	1.125	[0.83, 1.42]	<0.001***
Condition (V)	0.500	[0.22, 0.78]	<0.001***
Notebook (IMDB)	0.000	[−0.28, 0.28]	1.000
Order (2nd task)	0.083	[−0.20, 0.36]	0.561
Marginal R^2 (fixed)	0.178		
Cond. R^2 (fixed+random)	0.333		

P-Values: (*) $p < 0.1$; (**) $p < 0.05$; (***) $p < 0.01$
R^2: marginal (fixed effects) / conditional (incl. participant random intercept).

p-values > 0.3). The one timing outcome that does reach significance runs in V's favour. Among reviewers who caught both flaws, V reached the second one sooner ($d = -0.60$, $p = 0.001$), a time *saving* of approx. 1min 57 s ($d \times \text{SD} = -0.60 \times 3.27 \approx -1.95$ min).

Finding 2 - Time Benefits

The one timing effect that reaches significance favors the visual condition. Among reviewers who caught both flaws, V reached the second one ≈1 min 57 s sooner. **The visual overview increases flaw detection without slowing reviewers down.** Across time to the first flaw, time to finish, and time to complete the review, no timing measure differs significantly between the two conditions.

Establishing that the visual overview improves detection leaves open the question of where the advantage comes from. A higher flaw count tells us the diagram helps, but not *how*: whether reviewers caught flaws through the diagram itself *or merely read code more carefully when one was present*.

5.5.2 RQ2: Mechanism

In RQ2 we ask: **Through which representation do developers actually detect flaws, and what properties of the overview produce the detection advantage?**

If the diagram is genuinely the mechanism, two properties should follow: reviewers should perceive it as accurate and more helpful than text, and they should use it (over the code) as the route by which they find flaws. We analyze the perceived helpfulness and accuracy from the post-study survey and attribution from the review forms, where participants identified the contribution of the code and overview for each task.

Perceived Helpfulness and Accuracy. Participants rated the visual overview higher than the textual one on both dimensions (paired Wilcoxon signed-rank). They found it more helpful for understanding the notebook (median 5 vs. 4 on a 1–6 scale; $d_z = 0.64$, $p = 0.011$) and a more accurate representation of the code (median 5 vs. 4; $d_z = 0.47$, $p = 0.037$). Figure 5.5 shows the

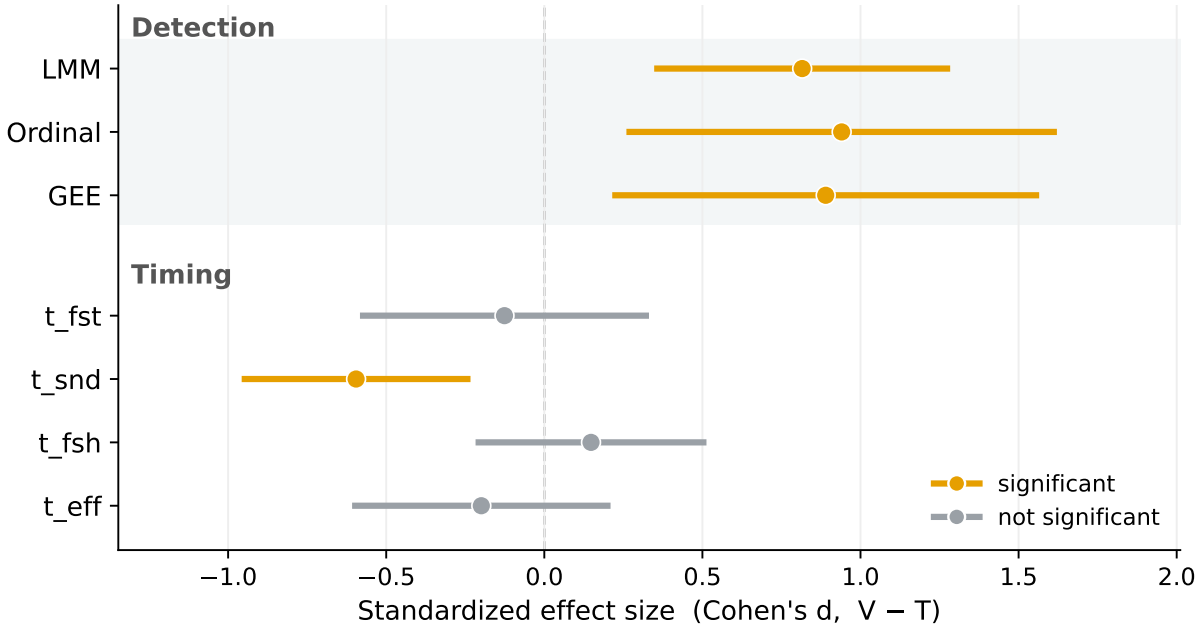


Figure 5.4: Standardized effect (Cohen's d, V – T) of the visual condition across every regression.

per-participant pairwise helpfulness difference (Δ = V – T). The distribution leans to the right, favoring V with a positive average difference (Δ = +1.38).

Detection Route. We next ask, among the flaws participants *actually caught*, how they credited the overview (V and T) relative to the code. For every reported flaw, participants rated the contribution of the code and of each overview on a four-level scale (*No*, *Minor*, *Strong*, *Very Strong*; encoded 0–3). Figure 5.6 reports three criteria (Fisher exact, V vs. T).

The visual overview was rated *above the code*, the primary detection route, about five times as often as the textual one (39% vs. 8%, OR = 7.50, p = 0.008). It was also credited as a *strong* contributor (Strong or Very Strong) more often, though in this sample the gap is only marginal (66% vs. 40%, OR = 2.88, p = 0.069). The one criterion that does *not* differ is whether the overview made *any* contribution at all (79% vs. 72%, p = 0.56): reviewers give both overviews at least minor credit, so the visual advantage lies not in being consulted but in its *primacy* as a detection route. In T, the code remains the dominant route; in V, the diagram competes with it on roughly even terms.

Finding 3 - Reviewers Use the Visual Overview as a Detection Route

Reviewers find the visual overview more helpful (d_z = 0.64, p = 0.011) than the textual one, and they credit it as the primary detection route (rated above the code) about five times as often (39% vs. 8%, p = 0.008). Both overviews (V and T) receive at least minor credit (79% vs. 72%), so the gain is in the diagram's value as a detection route, not in being consulted or not.

Workload. The NASA-TLX composite (five items, Performance reverse-scored, 1–10 scale)

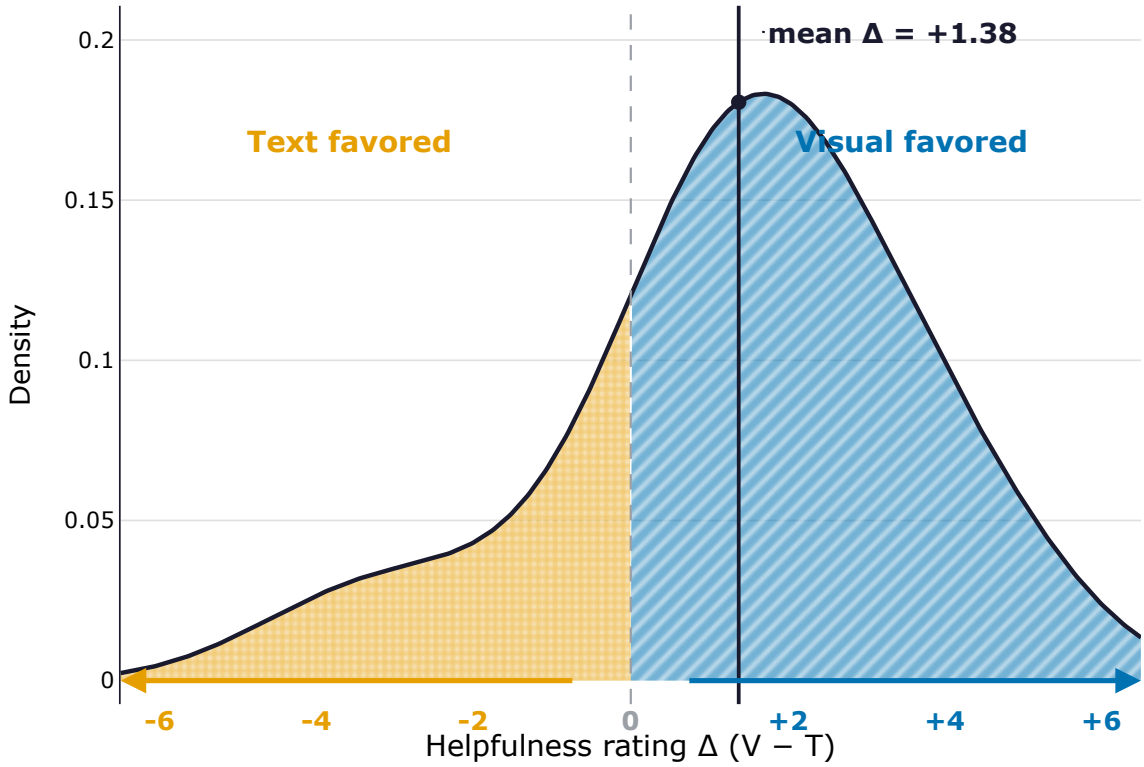


Figure 5.5: Distribution of the per-participant difference in perceived helpfulness.

did not differ between conditions (V 5.18 vs. T 5.55; paired $n = 21$, Wilcoxon $p = 0.31$), consistent with the absence time benefits cost in RQ1. Taken together, the three self-reports (perceived helpfulness, perceived accuracy, and attributed detection route) converge with the observed detection gain of RQ1: the diagram is not merely *available* under V; it is *perceived as more helpful* and *used as a primary detection route*.

A representation that improves detection can still be distrusted, ignored, or judged impractical. To understand how the visual overview fits into real review practice, we must turn from observed behavior to developers' perception.

5.5.3 RQ3: Perspectives

We explore developers' views through **RQ3: How do developers perceive the value, reliability, and future role of visual overviews in AI-assisted code review?**

We perform thematic analysis using open-ended questions from the post-user survey. We coded the responses into five themes: *value*, *reliability and trust*, *future role*, *emotions*, and *desired improvements*.

Value. Asked which documentation they preferred, **18 of 24 (75%)** chose the visual (V) overview. Eleven participants credited it for **quick comprehension**: a model of the code before

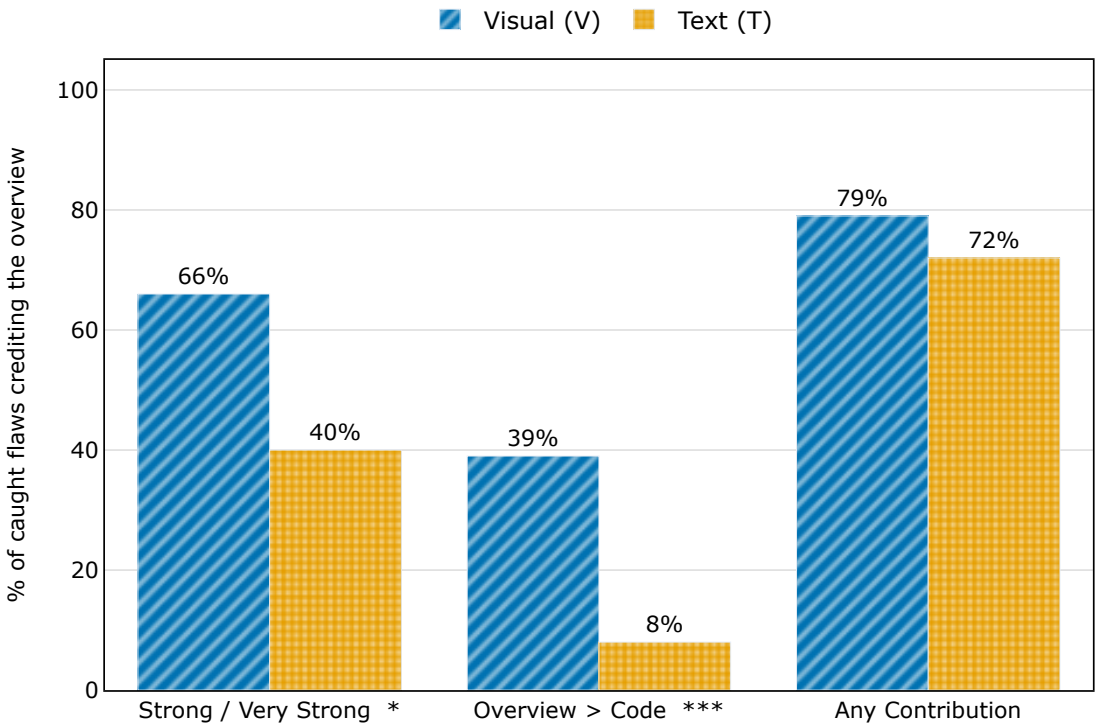


Figure 5.6: Attribution of caught flaws to each overview (V, T) by criterion, among the flaws participants detected (Fisher exact, $^{*}p < 0.1$, $^{***}p < 0.01$).

any detailed reading. P15 found it "*quicker grasping overall pipeline and where each step falls*", and P10 that "*Visual was better in terms of understanding the big picture and getting a holistic view*". Participants explain how some flaws were found: "*the diagram was really helpful to visualize that the test data was being used for one of the models but not the other*" (P23). Six reviewers framed the benefit as **reduced effort**, as P1 put it: "*Reading large amounts of code is already tiring. . . The visual aid relieved this*".

Reliability. The value is conditional on the diagram being *faithful to the code*. Eight said the overview is useful only as long as it can be trusted, and that they still verify it against the code: "*my only fear* [*being*] *the visual representation not being a faithful representation of the actual thing*" (P23).

Five participants preferred text over visual representations: "Textual. It gives more information about what is being done." (P25). This preference was also expressed by the participants without ML familiarity: P6 noted that they *"basically only read the code as it's the actual source of truth"* and argued that *"AI-generated documentation tends to be inaccurate. . . it's outdated the day you make it, so the code is more important to me."* Notably, trust could also be *earned* over time: P4 described an arc from doubt to reliance, explaining that *"At first. . . subconsciously, I couldn't trust it. Over time. . . it was essential for finding the second mistake."*

Complementing self-reports, we took informal notes during the sessions, and participants reacted most sharply when the diagram was gone. Moving into the textual (T) condition, several complained without prompting: "Damn it, where's the image?" (P02), "No pictures?" (P18), "I want the drawings again" (P23) and "It's annoying, bring the image back!" (P19).

Finding 4 - Value and Reliability

80% of participants mention their preference for visual overviews over text. They see clear value: quick comprehension, data-flow tracing, and lower effort. Their endorsement is conditional on **fidelity**: the overview is trusted only if it matches the code, and an inaccurate diagram misleads rather than helps.

Future role. Developers want visuals in their workflow: eleven said they would adopt it for real work (P17 called it "*a really handy way for doing code reviews*"; P4, that it "*will be helpful and I will use daily*"). They position it as a *complement, not a replacement*. P15 would use it "*as a supplementary to existing forms of documentation*", while P18 stressed that "*looking at the code itself is important*".

Several explicitly tied their value to the rise of AI-generated code; P2 was emphatic that "*. . . this seems very useful. . . especially with coding agents*". The recurring open question is *scalability*: five participants wondered whether the benefit survives large or complex codebases, as P12 worried that "*in more complex projects, the visual documentation will inevitably be more complex and may not have the same effect*", a concern P19 shared while still expecting that "*having a visual overview for the large projects will be very helpful*".

Improvements. Participants were concrete about what could improve. The most common request (five participants) was **code-linking and interactivity**: a live connection between the diagram and the code. Eight participants flagged **layout and legibility** (overlapping arrows and text, alignment, size), and others asked for an explicit key to the symbols and a mapping from each box to the code cells it represents (P10: "*know which code blocks are being referenced in different blocks in the diagram*"; P26: "*Indicate what arrows mean and what boxes represent*"). These are precisely the affordances that would let reviewers verify fidelity cheaply — the requirement Finding 4 identifies.

Finding 5 - Interactive Complement

Developers frame visual overviews as a **scalable complement to code review, well-suited to AI-assisted coding and reviewing.** The most requested improvements (code-linking, clearer visuals, explicit code mapping) all serve the same end: making it easy to confirm the diagram is faithful to the code. Scalability to large codebases is the key open question for future work.

5.6 Threats to Validity

Construct validity. Asking participants to review code invites more scrutiny than a typical over-confident developer applies. We mitigated this by framing each task as a review of code that

may or may not contain problems, eliciting a deploy/reject decision rather than a "find the bugs" prompt. Any residual bias is conservative: a less careful reviewer extracts even less from text but can still glance at a diagram, making the visual advantage *larger*, favoring our results.

A related concern is that the textual overview may be too weak to do a fair comparison. We controlled for this by deriving the textual overview from the same JupyterDraw diagram, so both encode identical entities, dataflow, and steps at the same level of abstraction; the textual variant removes only the mechanism under analysis: visual-spatial encoding (color, arrows, boxes, layout). We confirmed via LLM probing that the planted flaws are derivable from either representation alone, and the notebook code is identical across conditions.

External validity. There is a threat that our study under-represents a real scenario, since developers would use AI for the review. We design the experiments without AI because our questions concern whether a visual overview helps a *human* catch silent errors, not whether *agents* can detect them (frontier models find them immediately). The overviews we use (T and V) are meant to simulate the summarization that an agent would provide to explain the code during an AI-assisted review. We mitigate this risk by adapting both the flaws and the overviews to the 15-minute time budget. The planted flaw examples we used are documented as pervasive in real ML code and hard to detect manually [141], as drivers of the reproducibility crisis [52], and as structural debt that correctness checks miss [109]. LLMs share this blind spot: ~54.5% balanced accuracy on multi-step vulnerability detection [114] and correct code-review judgments only ~64–69% of the time [22]. The flaws and overviews are realistic proxies in two ways: first, we tested smaller models (Claude 3 Opus, GPT-4o-mini, GPT-4.1-nano) which uniformly miss both flaws in both notebooks, indicating that even subtler, less-documented variants of these flaws would likely evade detection by larger models as well, consistent with the literature above; and second, we have participants in all possible outcomes (from zero, one and two flaws found), which shows the tasks were not too easy or too difficult for the time budget.

A second threat is that our study is evaluated on data science code. While being an important part of Software Engineering [57, 58], it may not generalize to other software fields. The evaluation targets the intended audience for these notebook-based AI-generated artifacts, namely Python/Jupyter developers with ML background, so the sample is appropriate for the research question even though representativeness beyond this population is limited.

Internal validity. Detection ground truth is the post-interview marker annotated against the pre-defined criteria, so a note counts only when explicitly marked. There is a risk of annotation bias and missed annotations. We mitigate this by having the ground truth pre-specified (`solutions/summary.md`). Participants were incentivized with a performance bonus of up to $10, payable only for correctly detected flaws, creating a contractual commitment to correct scoring. The annotation criteria and coding rules are released with the replication package to enable independent verification. To further check that the coding is reproducible, we use an LLM coder that coded all 176 notes blindly from the same pre-specified criteria, never seeing our labels. Agreement was substantial (Cohen's ($\kappa = 0.77$) at the note level and 0.79 at the participant-level, detecting the flaw counts aggregate, over 86% raw agreement). Order and learning effects could also be a concern. We counterbalanced notebooks and conditions with a Latin square. The mixed model finds task order negligible ((+0.05) flaws, (p=0.76)) and notebook identity similarly; the visual advantage holds within both first and second tasks.

5.7 Summary

Developers are increasingly asked to review for code they did not write. Silent defects are even more dangerous: the code runs cleanly, reports plausible metrics, and is nonetheless methodologically wrong. We show that an automatically generated, informal picture of the ML workflows lets a reviewer judge such code without reading it line by line. For that we built JupyterDraw to conduct a within-subjects controlled experiment with 26 developers. The AI-generated visual overview behaved as the picture-superiority account predicts: reviewers caught a mean of +0.5 more flaws per task ($d_z = 0.69$, 95% CI $[0.21, 0.79]$; Wilcoxon $p = 0.005$), and more than doubled the rate of fully correct reviews from 29% to 67% (RQ1); they credited the diagram as a primary route to detection rather than a decoration (RQ2); and they preferred it as a supplementary aid whose adoption rests on trusting its fidelity to the code (RQ3).

These results position auto-generated structural summaries as a practical, low-cost layer of human oversight for AI-assisted development: a complement to code reading that makes relational flaws visible and that integrates into existing agentic workflows through MCP. The flaws our reviewers missed in the text were not hidden inside any function call; they lived in the connections *between* steps, a structure that a diagram makes easy to identify. As agents write more code and we read less of it, bringing pictures into the review loop offers a concrete way to keep developers accountable for what they ship.

Data Availability. All artifacts are available at `https://github.com/code-review-study/replication_package/`.

Chapter 6

Code Evolution: A Preliminary Study

Chapters 3 and 4 established the two directions of the interaction loop separately: that a multimodal model can recover executable structure from a developer's whiteboard sketch, and that it can render source code back into an informal diagram faithful enough to be measured. Both were studied as single-shot translations. Two findings from those chapters, arriving from opposite ends of the loop, point past that framing to the same missing capability.

The first is behavioural. In the sketch-to-code study, 47.4% of participants returned to their sketch while programming, without being asked or prompted to do so, treating the drawing as a persistent reference rather than as a one-shot specification that had been consumed. The artifact they had drawn minutes earlier was still doing work, but nothing in the tooling kept it true: the sketch described the program as it had been imagined, while the code moved on without it. The second is a measurement. In the code-to-sketch study, diagram faithfulness degraded systematically with the size of the input, losing roughly 8% of dataflow score for every additional 100 lines of code. Regenerating a whole diagram is therefore not merely wasteful as a system grows; it is progressively less accurate. Taken together, developers already behave as though sketch and code are a single, living artifact, and the regenerate-from-scratch strategy that served Chapters 3 and 4 cannot sustain that illusion at realistic scale.

What is missing is not a third translation direction but a different relationship between the two artifacts. Co-evolution requires something neither prior system maintains: persistent correspondences between regions of a visual abstraction and segments of code, durable enough that an edit to either side can be propagated to the affected part of the other while everything else is left untouched. This is the property that separates a collaborator from a generator, and Section 2.4 showed that no existing family of tools provides it. Derived visualizations are read-only, low-code platforms constrain expression to a fixed vocabulary, sketch-linking tools attach drawings to code without propagating anything, and formal bidirectional transformation presupposes the structure that informal sketches refuse to have.

This chapter takes the first step toward that capability. We present a design and a proof-of-concept prototype with an initial evaluation. The chapter presents the design of *Visual Loop*, the engineering decisions taken to make incremental propagation possible, a prototype demonstrating sketch-to-code interpretation, and an evaluation of how reliably that prototype reads structural intent from freehand annotation. The completed system and user study are left to future work, and what would be required to complete it is set out in Chapter 9.

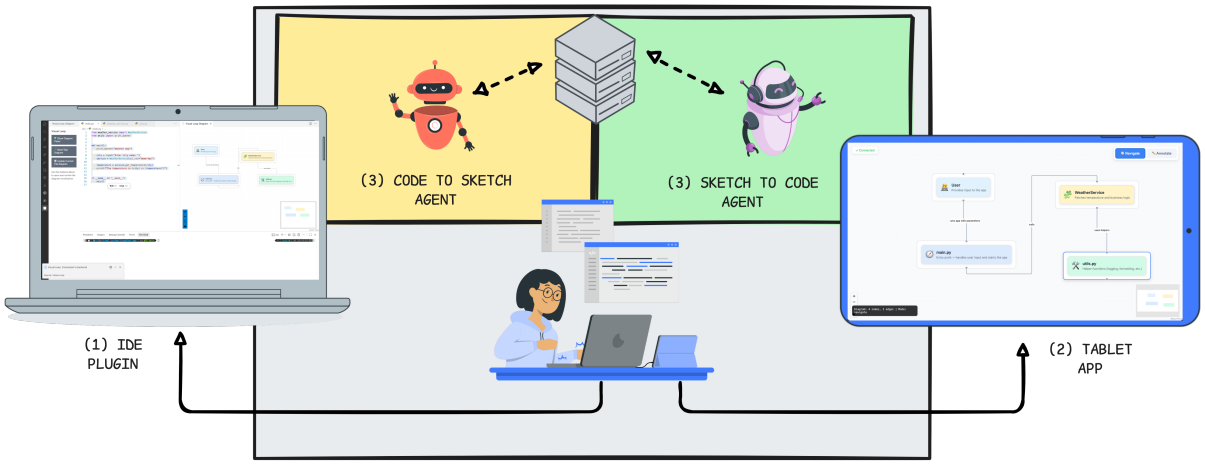


Figure 6.1: Visual Loop: a continuous visual development environment where code and sketches co-evolve through AI synchronization. Our prototype connects the IDE with a tablet interface, enabling developers to visualize, navigate, and modify systems through informal sketches: (1) desktop *IDE plugin* with static analysis and visualization hooks, (2) a tablet-based *Visualization Engine* for sketching and interaction, and (3) an *AI Synchronization Layer* powered by foundation models that maintain consistency between code and visuals in real time.

6.1 Visual Loop: The Vision

In *Visual Loop*, developers work with a tablet companion to their code editor, maintaining their mental model of the system as a living, manipulable artifact. They can express design intentions or modifications through freehand sketches, much as they would on a whiteboard, and the AI interprets these sketches contextually, reasoning about how they relate to the existing codebase and proposing corresponding changes. Unlike traditional visual languages, *Visual Loop* does not require a predefined syntax. Instead, the AI understands how to interpret developers' informal visuals in the same intuitive way that other humans would. This shared understanding transforms the development process from a sequence of textual commands into a continuous visual conversation between human cognition and machine reasoning.

6.1.1 Motivating Examples

This section motivates *Visual Loop* with examples of situations commonly encountered by developers.

Example 1: Onboarding Challenge. Alice joins a team working on an e-commerce platform. On her first day, she is asked to fix a bug in the checkout service. She spends hours searching through all files to understand the service architecture, drawing her own diagram on paper to map component relationships, and asking teammates. Her hand-drawn diagram becomes *outdated* after a refactoring PR is merged.

Problem. Alice must reconstruct the system's mental model from scratch using only text. The architectural knowledge exists only in senior developers' heads.

With *Visual Loop*

Alice opens the project in *Visual Loop*. The tablet immediately shows an auto-generated architecture diagram with all microservices.

She taps the "Payment Service" component on the tablet, and VS Code jumps to the relevant file. She circles three related components with her stylus and writes *"How to validate?"*

AI Assistant: *"These three services coordinate through the ValidationQueue: (1) CheckoutService initiates, (2) PaymentGateway processes, (3) FraudDetector validates. See highlighted code."*

When the refactoring PR merges, the diagram updates automatically. Alice's annotations remain, now correctly aligned with the new structure.

Result. Alice saves times by understanding the system visually, navigates through diagrams, and maintains an always-current mental model.

Example 2: Design Discussion. A team gathers for an architecture review. The tech lead explains a proposed refactoring on the whiteboard, sketching both the current and new architectures while team members discuss and suggest changes. Someone takes a photo of the board before erasing it. Later, a developer manually translates the sketch into code but discovers inconsistencies during implementation. The photo shared in Slack becomes the only record, quickly outdated and not synced with the evolving code.

Problem. Rich visual design discussions rely on manual translation to code, leading to implementation drift and lost design rationale.

With *Visual Loop*

The team connects a tablet running *Visual Loop* to the projector, displaying the current architecture diagram. The tech lead sketches directly on the diagram—adding a new `CacheLayer`, drawing arrows from the API Gateway and Database, and crossing out the old direct connection. Next to the cache, she writes: *"Redis? What size?"*

AI Assistant: "Proposed changes detected: add Redis-based CacheLayer, route API Gateway through CacheLayer, remove direct database calls. Cache configuration unresolved."

As the team debates alternatives, *Visual Loop* displays both options side by side for comparison. Once the team agrees, the AI applies the chosen design: it updates routing logic, adds TODO comments for configuration, and commits the synchronized architecture to the shared documentation.

Result. The design discussion directly becomes implementation. Less manual translation and no lost knowledge.

Example 3: Bug Hunt. Bob is debugging a race condition. He sets breakpoints in several files, reruns the program, and sketches the execution flow on paper. He discovers a timing issue between three asynchronous operations. The sketch helps him reason about the bug but stays informal and hard to share with teammates.

Problem. Debugging demands understanding dynamic behavior, and existing tools show only static code, leaving developers' mental models of execution unshared and disconnected from their environment.

With *Visual Loop*

It generates a live diagram of the payment flow, updating in real time as Bob steps through the debugger. When he notices three operations running in parallel, he circles them and writes "Race condition here?" on the tablet.

> **AI Assistant:** "Detected unsynchronized async calls: `UpdateInventory`, `ChargeCard`, and `SendConfirmation`. Suggest adding `await` on `ChargeCard` before `SendConfirmation`."

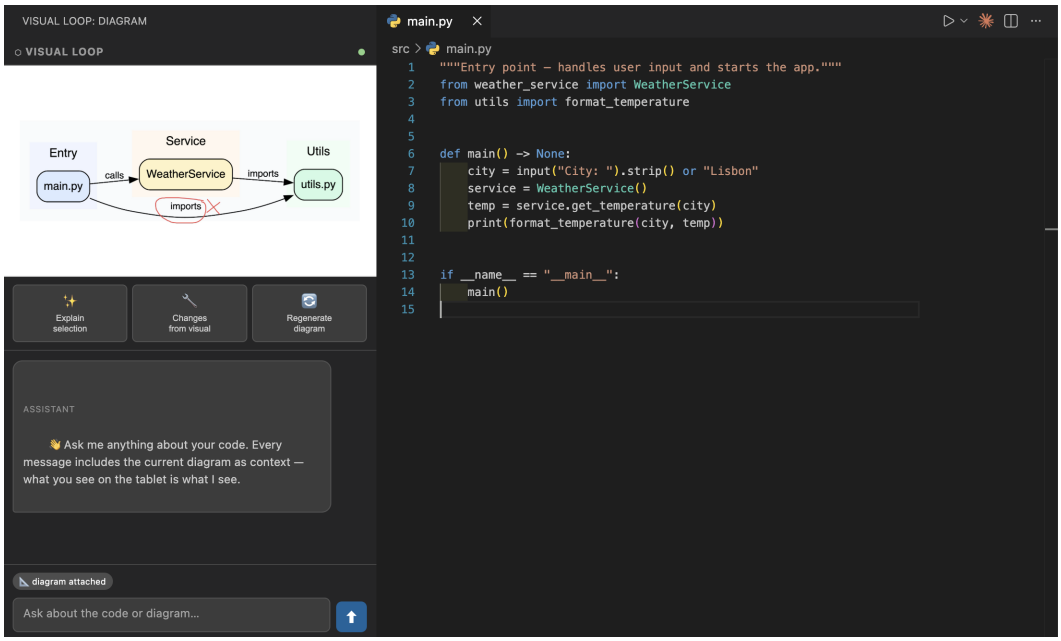
Bob draws an arrow from `ChargeCard` to `SendConfirmation` to enforce the correct order. The AI updates the code and synchronizes the diagram, showing the fixed flow. He attaches the annotated visualization to the pull request, illustrating the bug and its resolution.
Result. Complex asynchronous debugging becomes intuitive and collaborative. Bob solves the issue visually before coding, and his team learns from the automatically generated documentation.

6.1.2 Prototype

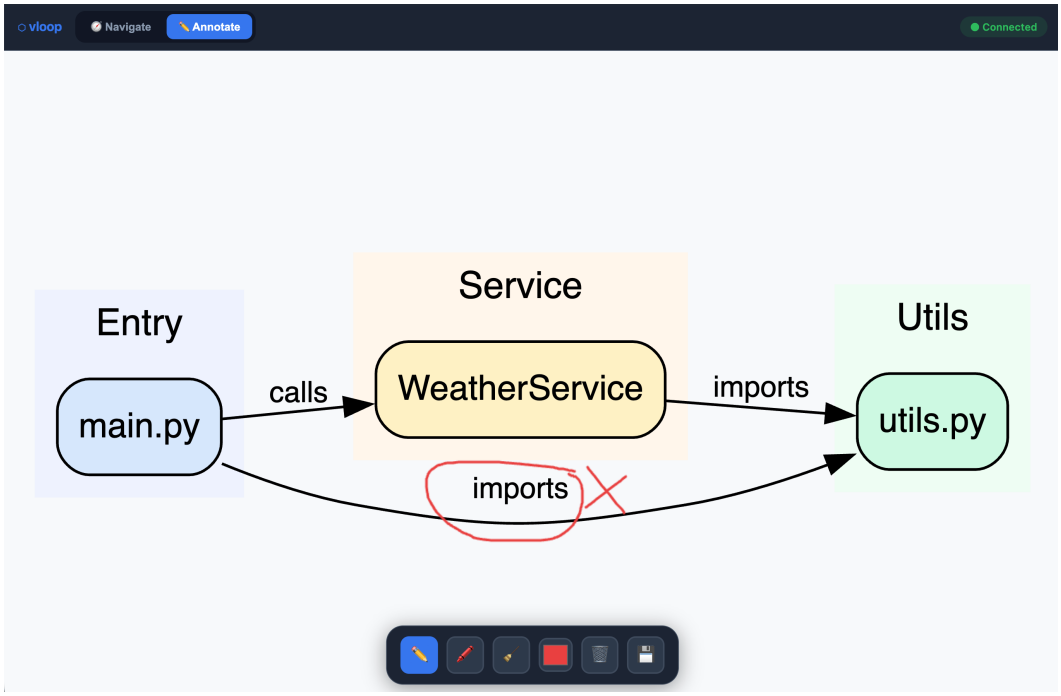
The *Visual Loop* prototype exemplifies a new interaction direction where foundation models bridge the gap between code and human *visual* reasoning. Built on top of multi-modal LLMs, the system connects the developer's coding workspace with an intelligent visualization environment. Our prototype integrates three main components (Figure 6.1):

1. **Code Editor (VSCode extension).** A standard development environment augmented with static analysis and contextual extraction. It feeds structured information—such as function hierarchies, data flow, and dependencies—into the foundation model, grounding its visual reasoning in the actual code semantics (Figure 6.2 - *a*).
2. **Visualization Engine (Tablet app).** A companion workspace for sketch-based interaction, real-time diagram updates, and code navigation. Developers can freely draw and annotate while the underlying model interprets these gestures and symbols within the code context. This interface transforms the tablet into an intelligent visual processor that understands intent and syntax (Figure 6.2 - *b*).
3. **AI Synchronization (LLM).** This layer orchestrates two complementary sets of AI agents: *Code to Sketch* and *Sketch to Code*. These agents leverage the multi-modal reasoning (code, text, and images) to maintain semantic alignment between visuals and source code. Code2Sketch updates diagrams that reflect the evolving program structure, while Sketch2Code interprets developer visual annotations to propose code edits or refactorings.

Three engineering decisions were taken specifically to keep incremental updates reachable, even though the prototype does not yet implement them fully. Node identity is content-addressed with a unique key, so a node keeps its identity across independent regenerations, which is the precondition for a structural diff and for reattaching a developer's annotation to the element it was



(a) Desktop editor.



(b) Tablet companion, annotation mode.

Figure 6.2: *Visual Loop* interfaces, showing the corresponding codebase in the code editor and tablet. **(a)** The vscode panel, containing: the diagram with the developer's annotation on it, quick actions for code-diagram evolution, and an assistant that uses the diagram as attached context. **(b)** On the tablet the diagram fills the canvas and is marked up directly with the stylus. Because the two devices hold one shared diagram rather than two copies, the stroke drawn in (b) is the stroke the editor reasons about in (a).

drawn on; correspondences are anchored to symbol names instead of line ranges, with the concrete range resolved on demand through the editor's language server, so the mapping survives any edit that shifts line numbers, avoiding the fragility that reduced SketchLink and LivelySketches to archival links once the code moved on [7, 8]; and the underlying representation of the diagram is a textual graph rather than relying only on a rendered image, so that two versions can be diffed and merged with the same techniques already used for text.

6.2 Proof-of-Concept: Weather App Demonstration

To illustrate *Visual Loop*, we implemented a minimal prototype using a VS Code extension and a tablet app (Figure 6.1). For the proof-of-concept, we use a simple weather application composed of three files: `main.py`, `utils.py`, and `weather_service.py`. The WeatherService class fetches and processes data using helper functions in `utils.py`.

In this proof-of-concept example, the goal is to remove the *WeatherService* class, identified as a source of redundant logic and complexity, and refactor the application to call `utils.py` directly. It follows the workflow depicted in Figure6.3:

1. Initialization The developer opens the project in *Visual Loop*. The desktop editor displays the auto-generated diagram, which can be freely rearranged with drag-and-drop to match the developer's mental model.

2. Synchronization The connected tablet mirrors this updated layout in real time, maintaining consistent spatial organization between devices.

3. Exploration Using the tablet in `Navigation Mode`, we can tap on nodes, automatically highlighting the corresponding code segments in VS Code. This live linkage bridges high-level structure and implementation details.

4. Annotation Switching to `Drawing Mode`, the developer sketches arrows and notes directly on the diagram. For example, cross out WeatherService, draw new links from `main.py` to `utils.py`, and write "direct call" as an annotation.

5. Interpretation Upon pressing Save, the hand-drawn modifications are sent to the LLM-based reasoning module.

The AI interprets the visual intent and summarizes it.[1]

> **AI Answer:** The user wants to make two main changes based on the hand-drawn annotations: 1. **Remove the "WeatherService" Component**. 2. **Add a direct call to utils.py**.

The summary is presented to the developer for confirmation. In future iterations, these AI inferences can be applied directly as candidate code edits.

The prototype was evaluated across 30 drawing interactions of four types. The model correctly interpreted most annotations, achieving the following accuracies: (1) Remove components: 4/5 cases (80%), (2) Add components: 6/7 (85.7%), (3) Remove calls: 6/8 (75%), and (4) Add calls: 9/10 (90%).[2]

[1]The model used in the prototype is OpenAI's `gpt-4o-mini`.

[2]Some interactions are implicit; for instance, removing a node also implies removing its associated calls. Full experimental data are available at: `https://github.com/luisgomes24/Visual-Loop`.

6.3 Discussion

Visual Loop proposes a paradigm shift rather: a move toward continuous, visual–textual co-evolution of code and visuals. This transition raises conceptual, technical, and practical questions worth discussing.

Visual Loop is not a replacement for programming languages. It does not aim to remove code but to expand how intent is expressed. Visual interaction is most beneficial when it mirrors how developers reason. Text remains superior for precision and completeness, but visuals provide an intuitive medium for reasoning and for communicating structural changes. The two modalities are not competing; they form a complementary loop that aligns with human cognition.

Foundation models play a central role in this vision. Traditional visualization systems relied on predefined grammars or rule-based mappings between code and diagrams [68, 98, 100]. In contrast, foundation models bring generalization: they can infer structure from code and interpret informal sketches without relying on fixed visual vocabularies [39, 138]. This opens a new research path where models learn to ground abstract visual reasoning in executable semantics. However, this flexibility also introduces challenges in controllability, explainability, and reproducibility that must be addressed.

The approach raises questions about authorship and collaboration. When AI agents interpret or evolve diagrams and code, who is responsible for the resulting changes? In collaborative environments, multiple developers might annotate or sketch simultaneously, requiring version control mechanisms for text but also for visual artifacts and AI inferences.

The integration of visual reasoning introduces evaluation challenges. Unlike traditional IDE extensions or visualization tools, the quality of *Visual Loop* interactions depends not only on output correctness but also on how well the system aligns with human cognitive processes. Empirical studies on usability, and cognitive load are crucial to determine when visual co-development improves or degrades problem-solving.

6.4 Summary

Software development has assumed that programming should be primarily textual, although humans think visually. The emergence of multimodal foundation models challenges this assumption, enabling tools that align with human cognition rather than computer syntax. In this chapter, I introduced *Visual Loop*, taking a first step toward this vision: a development environment where code and visuals co-evolve. AI interprets freehand sketches as developer intent, allowing the transition between visual reasoning and code. This is not a replacement for coding but a cognitively aligned interface where text and visuals show synchronized views of the system.

6.5 Takeaways

Where *Visual Code Assistants* and *VisDocSketcher* establish the two directions of the human–AI interaction loop independently (sketch-to-code and code-to-sketch) *Visual Loop* sets out to close the loop by making the two artifacts co-evolve. This is not an incremental extension but a

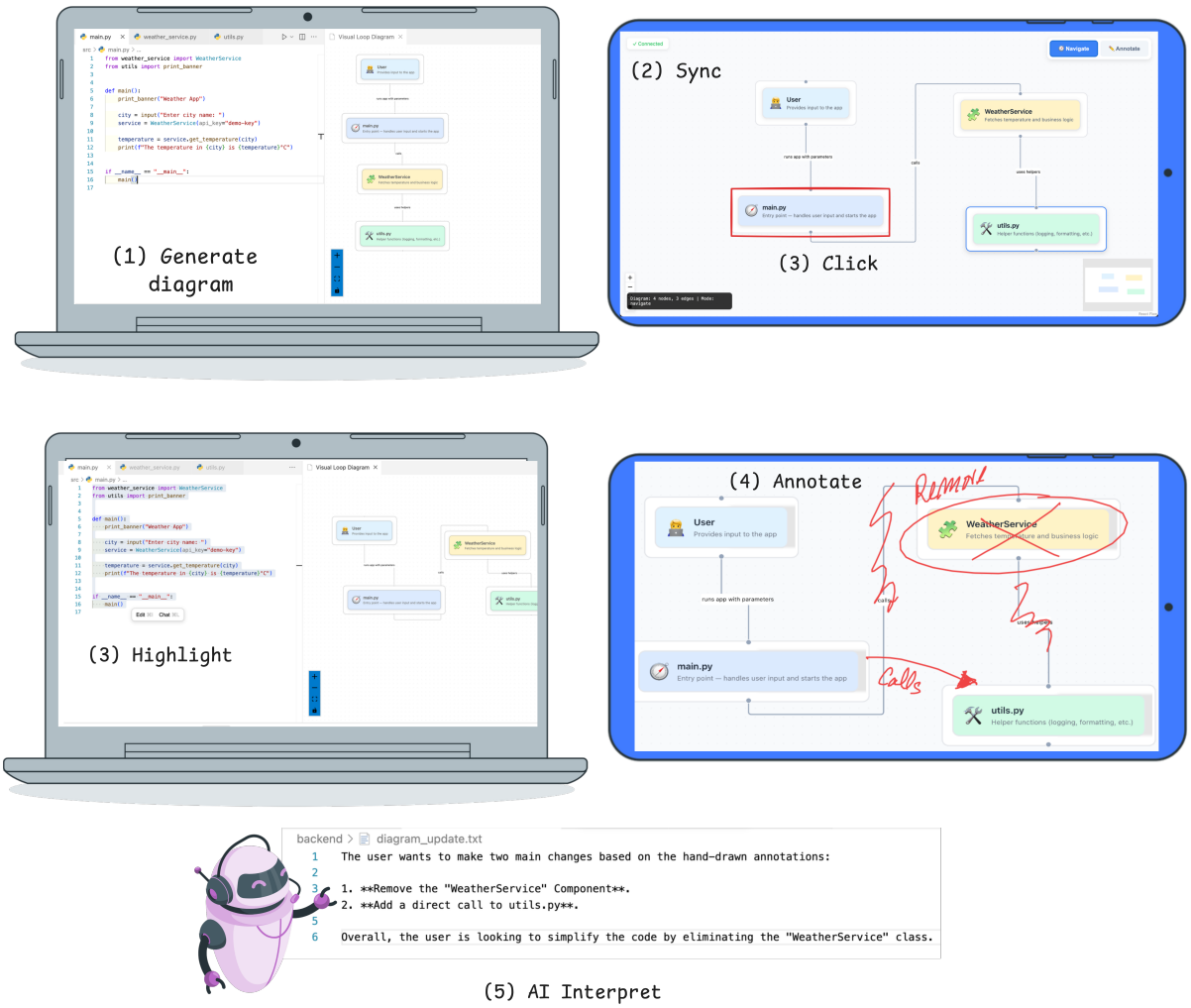


Figure 6.3: Proof-of-Concept Example: exploring code and rearranging diagrams to sketch changes, interpreting them through AI, and synchronizing updates between code and visuals.

qualitative shift in the interaction model: the prior two systems treat each translation as a fresh, stateless generation task, whereas co-evolution requires the system to maintain persistent, fine-grained correspondences between visual regions and code segments across time. The thesis claim that AI can function as a cognitive collaborator—rather than a transactional generator—depends on this property: a collaborator does not redraw the entire whiteboard every time a function changes.

Across 30 freehand editing interactions spanning four types, the system read the developer's structural intent correctly in 75–90% of cases, and it did so without a predefined visual vocabulary: the annotations were crossings-out, arrows, and words written beside nodes, of the kind Section 2.1 showed developers already produce unprompted. That establishes something the preceding chapters did not test. An informal edit to a diagram can serve as a legible specification of a *change*, not merely as a legible description of a system.

The evaluation covers a single small application, so it establishes that the interpretation is possible rather than how it behaves across developers or at scale. This chapter therefore shows the interaction to be feasible while leaving open whether it is sustainable, which is the question Process 3 ultimately has to answer. Chapter 9 sets out what answering it requires.

Chapter 7

Summary of Contributions

This thesis advances the vision of AI systems that participate in software development through the same multimodal, informal artifacts that developers already use. The contributions span system design, empirical evaluation, and methodology, organized across three challenges and one cross-cutting concern.

- **Visual Code Assistant: concept and prototype.** We introduce the concept of a *Visual Code Assistant* and implement the first in-IDE prototype that converts informal, hand-drawn sketches of ML workflows into executable Jupyter Notebooks. This is the first system to treat structural, intent-bearing sketches as first-class specifications for code generation. [39]
- **Benchmark for sketch-to-code evaluation.** We conduct a user study evaluating code generation from whiteboard sketches with 19 data scientists, producing a dataset of paired sketches and notebooks, an LLM-as-a-Judge evaluation methodology validated against human grading, and an empirical characterization of ML developer sketching practices. [39]
- **VisDocSketcher: agentic system for visual documentation generation.** We design and implement a system that automatically generates informal, high-level visual documentation directly from source code, combining static analysis with a multi-agent LLM architecture that produces sketch-style flowcharts at the abstraction level developers already reason about. [41]
- **AutoSketchEval: reference-free evaluation framework for visual documentation.** We propose a novel evaluation framework in which sketch quality is assessed through the fidelity of code reconstructed from the sketch via an independent Sketch2Code system, requiring no ground-truth diagrams. [41]
- **JupyterDraw: empirical evidence that visual overviews improve review of AI-generated code.** We build JupyterDraw, a tool that renders whiteboard-style overviews of Jupyter Notebooks, and conduct the first controlled within-subjects experiment with 26 developers on whether automatically generated visual summaries help reviewers detect silent methodological flaws in AI-generated notebooks. Visual overviews raised complete flaw detection at no cost in time or workload and were also credited as the primary detection route nearly five times as often as textual summaries.
- **Visual Loop: a design and proof-of-concept for co-evolving code and visual artifacts.**

We design a system in which informal visual sketches and source code co-evolve, anchored on correspondences between visual regions and code symbols, and implement a prototype that demonstrates the sketch-to-code direction. An evaluation with 30 freehand editing interactions shows the system interprets structural edit intent with 75–90% accuracy across four interaction types. [33]

Chapter 8

Discussion

The transformation that motivated this thesis is no longer speculative. Reported numbers indicate that by 2025 more than 30% of new code at Google and between 20% and 30% at Microsoft was generated by AI [15, 29]. However, trust in the accuracy of these tools fell to nearly 30%, and for the first time, developer distrust in AI-generated outputs outweighs trust [113]. This is not only a matter of self-reported sentiment, but it was also reported in controlled studies. Sabouri *et al.* showed that comprehensibility and perceived correctness were the factors most frequently used to evaluate the trustworthiness of code suggestions, shaping how developers actually verify what they ship [107]. The industry largely reads lack of trust as a model-quality problem, one that the next generation of models will likely close. This dissertation suggests a different reading: a substantial part of the gap is a *representational* problem. The interface between agents, code, and human judgment remains a wall of text. Our results show that shifting that interface without changing the underlying model measurably changes what developers understand and trust. In this chapter, I discuss what this means in practice for the software engineering community.

Visual specifications as input. The clearest industry signal of 2025 was the rise of spec-driven development: GitHub's Spec Kit, Amazon's Kiro, and similar tools now build AI-assisted work around specification documents written before code generation [117]. But these tools primarily accept text. This overlooks an important aspect of how developers reason, since developers do not hold structural intent solely in prose: they hold it spatially and often draw it [6]. Our results show sketches capture system structure with 70–80% fidelity, while text is better for constraints and parameters. Design meetings and whiteboard sessions should count as real specification artifacts, not throwaway scribbles to be transcribed later. Concretely, spec-driven tools and AI coding platforms (e.g., Claude Code, Copilot, Cursor) should adapt their interfaces to accept sketches and images as first-class inputs alongside text. This also opens specification to people who cannot write structured specs or code. Our participants pointed to cross-functional meetings where non-programmers described pipeline intent by drawing. Visual specification could give those people a real way to specify systems, democratizing coding without needing a formal vocabulary, something low-code platforms promised but never fully delivered.

The bottleneck has moved, from generation to understating. Code and documentation generation are no longer the biggest constraint on AI-assisted development; human comprehen-

sion is. Industry telemetry reports that teams adopting AI agents merge nearly twice as many pull requests while review time grows by roughly 90% [78]. Industry telemetry and controlled studies converge on the same paradox: AI coding agents increase throughput but slow down review and closure cycles [21, 78]. Even when practitioners perceive the tools as useful and feel faster, empirical measurements show the opposite—time saved on generation is offset by time spent verifying AI-generated output [12]. Large-scale analyses show code duplication multiplying since AI assistants became widespread [36], which is what merging faster than one can understand looks like at scale. The industry's dominant response has been to automate the review itself, but delegating oversight to the same class of system that produced the code relocates the accountability question instead of answering it [59]. Additionally, developers with LLM-based review tools reported faulty reviews, unnecessary corrections, and irrelevant comments: the automated reviewer added a verification burden of its own [21]. Our experiments isolate a mechanism that the field has barely touched. Changing only the *representation* from textual to visual more than doubled the rate of complete flaw detection. Comprehension throughput, in other words, is not fixed by human cognitive limits; it is fixed by representations that ignore them. Building effective code assistants requires aligning with the ways developers naturally reason and communicate. Although developers frequently rely on whiteboards, sketches, and visual artifacts to communicate and understand coding ideas, existing assistants remain constrained by text-centric interactions. Next-generation code assistants should leverage these overlooked visual channels to explain code and better support developer cognition.

Automatic evaluation to solve the trust gap and the verification-cost problem. Participants in our studies suggest one factor which dominates their trust relationship with generated overviews: the verification cost. They used the visual overview to the extent that they could check it against the code and asked for element-to-code linking as their top improvement. Closing the trust gap does not require making AI output more persuasive; it requires making AI claims cheaper and easier to verify. Automatic evaluation and verification becomes an important part of the generation process. Round-trip methods like AutoSketchEval show that the fidelity of an explanation artifact can be measured without ground truth [41], which means platforms could attach a fidelity check to generated overviews the way continuous integration attaches tests to commits. The same logic argues for ensemble judging over single-judge evaluation when correctness itself must be assessed automatically [152]. Following the same psychology principles that make visuals more effective, we need to be cautious when presenting visual overviews to developers: the picture superiority effect [133] does not distinguish faithful pictures from unfaithful ones, meaning that a wrong diagram may mislead with more authority than a wrong paragraph. Visual explanation and fidelity verification are not two separate features. They are two sides of the same coin, and shipping the first without the second would leave developers worse off than text.

Visual documentation can become a build artifact. Software teams have never disputed the value of diagrams; they have disputed who updates them. The diagrams-as-code movement, with Mermaid rendered natively on GitHub since 2022 [37], solved rendering and versioning but not upkeep. Diagrams, like other manually maintained artifacts, drift out of correspondence with the implementation as it evolves, through lack of documentation practice, knowledge gaps, or simple neglect. The results of this thesis change the economics of the whole pipeline rather

than one stage of it: generation is automatic, fidelity is measurable without reference diagrams, and the correspondence tracking prototyped in *Visual Loop* makes updates incremental rather than regenerative. Together, these ideas make it possible to automatically keep diagrams up to date. If the code and diagram don't match, the change is blocked until they do. This also makes code reviews easier: instead of starting with hundreds of changed lines, reviewers can first see a simple diagram of what changed. Since people in our study relied on these diagrams once they had them, this could make reviews faster and easier.

Software tools were built to help computers understand code that humans wrote. Now AI writes much of the code, and humans spend more time trying to understand it. This dissertation argues that the biggest improvement is not better AI models, but better ways to explain what the code is doing. Simple diagrams can help people plan software, catch problems that are easy to miss in code or text, and stay accurate by automatically checking them against the code. Every result in this dissertation used today's widely available AI models. As AI keeps improving, the tools that matter most will be the ones that help people understand code, not just produce it.

Chapter 9

Future Work

The contributions of this thesis establish that informal visual artifacts can specify code, explain it, and co-evolve with it. Each of these results also exposes what remains to be done. Some directions are consolidations, reproducing existing findings with better generation technology and broader domains, while the largest one is the concretization of the vision that motivates the thesis: a co-evolution system that learns how each developer, team, and organization visualizes software.

Regenerating visual documentation with JupyterDraw. The VisDocSketcher results were obtained with Mermaid diagrams, whose vocabulary is restricted to nodes and connections laid out by a deterministic renderer. Our later work showed that giving the model full control over space, color, and shape produces diagrams that developers value and that improve flaw detection, so the natural next step is to reproduce the VisDocSketcher study using JupyterDraw as the generation engine. This requires adapting JupyterDraw from a single steered agent into a multi-agent architecture mirroring the one we validated: an *analyzer* agent equipped with the code analysis tools (dataflow graph construction, variable dependency extraction), a *planner* agent that drafts the logical structure of the diagram in Mermaid, which serves as a compact intermediate representation, dedicated tools to convert the Mermaid plan into Excalidraw elements, and a *visualizer* agent with explicit instructions to optimize the spatial layout. The visualizer stage responds directly to our study participants' most common complaints, since eight of them flagged overlapping arrows, misalignment, and occluded labels as the main obstacles to legibility. Beyond the engineering effort, this reproduction tests a hypothesis the thesis leaves open: whether the quality gains we measured for multi-agent generation (better sketches in 59.3% of cases) carry over from constrained to free-form visual vocabularies.

Beyond notebooks and data science. Our generation and review studies target Jupyter notebooks, a domain chosen deliberately for its visual, pipeline-shaped structure and for the availability of ground-truth sketches. The techniques themselves are not notebook-specific: the analyzer operates on general code properties, and the round-trip evaluation only assumes that code can be reconstructed from the visual. Future work should test how far this generalizes by studying other abstraction levels that developers already draw, such as architecture diagrams of multi-component systems, representations of features and their interactions across the codebase, and execution or debugging flows. Our complexity results suggest this will not be a matter of simply pointing the tools at larger inputs, since sketch quality dropped 8% for every additional 100

lines of code; hierarchical views that decompose a system into linked, per-component sketches are the most promising path.

From prototype to a co-evolution system. The final goal of this thesis is to turn the *Visual Loop* proof-of-concept into a working development environment, and this is where the vision is concretized. The current prototype demonstrates the loop on a simple application, interpreting 75–90% of hand-drawn annotations correctly, but it interprets each interaction in isolation. A full system must maintain persistent, fine-grained correspondences between visual regions and code segments across the development lifecycle, propagate localized changes in both directions, and support the different types of visualizations, from pipeline overviews to architecture diagrams, within the same environment. The most important open problem is adaptation. Developers do not share a single visual language: our first study documented how widely sketching styles vary across individuals [39], and teams and organizations layer their own conventions on top. A co-evolution system should adapt at different levels (e.g. to users, teams, organizations) and it should do so not only through written rules and configuration but by learning from interactions: every rearranged diagram, corrected interpretation, and rejected proposal is a signal about how that developer visualizes software. Over time, the system should converge on each user's notation the same way a new team member does. This raises research questions the thesis opens, including how to represent and version learned visual preferences, how to reconcile conflicting conventions within a team, and how to keep adapted diagrams verifiable, since an overview that adapts to its reader must still be faithful to the code, the requirement our review study identified as non-negotiable.

Understanding the human side of co-evolution. Like the two completed systems, *Visual Loop* needs a user study before its claims can be considered established. The studies in this thesis measured single-shot interactions, one sketch generating code or one diagram supporting one review. Co-evolution should be a dialogue, and its costs and benefits may only appear over time. Similar to our previous studies, the system should be tested with professional developers measuring how continuous visual-textual synchronization affects comprehension, task efficiency, and collaboration, together with qualitative analysis of how developers perceive AI agents that edit their diagrams and code. Interesting directions include whether they trust the propagated changes and how they recover when an interpretation is wrong.

Consolidating the empirical base. Several smaller threads from the individual studies deserve follow-up. The relationship between sketching effort and generation quality was estimated on 19 participants and its key coefficient fell just short of significance, so a larger sample is needed to estimate it reliably [39]. Our review study used planted flaws in controlled notebooks; replications with naturally occurring flaws and with tasks closer to production complexity would strengthen external validity. The code-linking feature our participants requested most, tapping a diagram element to reach the code it represents and an explicit legend of the notation, are implemented in the *Visual Loop* prototype but their effect on review performance has not been isolated experimentally. Finally, on the evaluation side, the ensemble judging strategies of SE-Jury [152] have been validated for code artifacts and could be extended to visual ones, complementing round-trip metrics with judge-based assessments of properties reconstruction cannot capture, such as legibility and appropriate abstraction.

These directions follow the original vision: from generating and evaluating individual visual artifacts, which this thesis has shown to be feasible and beneficial, toward development environments where visuals are a persistent, personal, and trustworthy interface between developers and

AI.

Chapter 10

Conclusion

Software engineering has always been a multimodal activity. Developers sketch architectures on whiteboards, trace dataflows with arrows, and negotiate design decisions through drawings long before, and long after, any code is written. AI assistants now writing a substantial fraction of the world's code communicate through a single channel, text, forcing developers to translate spatial, structural intent into prose on the way in and to reconstruct structure from walls of generated text on the way out. This mismatch is not a usability detail but a core limitation of the current paradigm: by embedding AI systems within multimodal, iterative interaction loops that mirror human-human collaboration, we can improve mental model alignment and move AI tools from code generators toward cognitive collaborators.

My research evaluated this claim across the three processes that close the human-AI interaction loop. For *code generation* (Process 1), the Visual Code Assistants study showed that informal, hand-drawn sketches function as genuine specifications: multimodal LLMs recovered 70–80% of the intended code outline from whiteboard sketches alone, each additional minute of sketching improved accuracy by 2.5 percentage points, and sketching reduced the code participants had to write by 49% on average [39]. Nearly half of the participants spontaneously returned to their sketches while coding, evidence that developers already treat drawings as persistent, shared artifacts rather than one-shot prompts.

For *code explanation*, VisDocSketcher demonstrated that informal, developer-style visual documentation can be generated automatically from source code, and AutoSketchEval showed that the fidelity of such visuals can be measured without any ground-truth diagrams, distinguishing code-aligned from non-aligned sketches with an AUC above 0.87 [41]. The JupyterDraw review study then established that these visuals matter where it counts: with the model, the code, and the time budget held constant, handing reviewers an AI-generated visual overview instead of a textual summary more than doubled the rate of fully correct reviews, from 29% to 67%, quintupling the odds of catching every planted flaw. Representation alone, with no change to the underlying model, measurably changed what developers understood.

For *code evolution*, the *Visual Loop* prototype closed the loop, maintaining correspondences between visual regions and code segments and interpreting 75-90% of hand-drawn annotations correctly, showing that code and informal visuals can co-evolve through localized, bidirectional updates instead of regeneration.

Taken together, these results support my thesis statement. Sketches can specify code, code can

be explained through the informal visuals developers already draw, the two can stay synchronized as both evolve, and every step of that loop can be evaluated automatically. The broader lesson is that the gap between developers and AI assistants is to a significant degree a *representation* problem rather than purely a model-quality problem: the largest comprehension gains came from changing what the developer sees and how they express their intents.

As AI systems take over more of the writing of software, the bottleneck in development is shifting from the production of code to the human understanding of it. The contributions of my thesis outline one path to protecting that resource: AI tools that meet developers at the whiteboard, interpret what they draw, explain themselves in the same visual language, and keep that shared picture faithful as the system changes. An assistant that participates in this loop is no longer a chat bot that emits text and code; it is a collaborator that helps build, and continuously maintains, the shared mental model on which trustworthy software development depends.

Bibliography

[1] Md Abdullah Al Alamin, Sanjay Malakar, Gias Uddin, Sadia Afroz, Tameem Bin Haider, and Anindya Iqbal. An Empirical Study of Developer Discussions on Low-Code Software Development Challenges. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, page 46–57, Madrid, may 2021. IEEE. doi: 10.1109/msr52588.2021.00018. URL http://dx.doi.org/10.1109/MSR52588.2021.00018.

[2] Miltiadis Allamanis, Sheena Panthaplackel, and Pengcheng Yin. Unsupervised evaluation of code LLMs with round-trip correctness. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

[3] Dimitar Asenov, Otmar Hilliges, and Peter Müller. The Effect of Richer Visualizations on Code Comprehension. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, page 5040–5045, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450333627. doi: 10.1145/2858036.2858372. URL https://doi.org/10.1145/2858036.2858372.

[4] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721, 2013. doi: 10.1109/ICSE.2013.6606617.

[5] Jason Baldridge, Jakob Bauer, Mukul Bhutani, Nicole Brichtova, Andrew Bunner, Lluis Castrejon, Kelvin Chan, Yichang Chen, Sander Dieleman, Yuqing Du, et al. Imagen 3. *arXiv preprint arXiv:2408.07009*, 2024.

[6] Sebastian Baltes and Stephan Diehl. Sketches and diagrams in practice. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 530–541, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330565. doi: 10.1145/2635868.2635891. URL https://doi.org/10.1145/2635868.2635891.

[7] Sebastian Baltes, Peter Schmitz, and Stephan Diehl. Linking sketches and diagrams to source code artifacts. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 743–746, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330565. doi: 10.1145/2635868.2661672. URL https://doi.org/10.1145/2635868.2661672.

[8] Sebastian Baltes, Fabrice Hollerich, and Stephan Diehl. Round-Trip Sketches: Supporting the Lifecycle of Software Development Sketches from Analog to Digital and Back . In *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 94–98, Los

Alamitos, CA, USA, sep 2017. IEEE Computer Society. doi: 10.1109/VISSOFT.2017.24. URL https://doi.ieeecomputersociety.org/10.1109/VISSOFT.2017.24.

[9] Dor Bank, Noam Koenigstein, and Raja Giryes. Autoencoders. *CoRR*, abs/2003.05991, 2020. URL https://arxiv.org/abs/2003.05991.

[10] Shraddha Barke, Michael B. James, and Nadia Polikarpova. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proc. ACM Program. Lang.*, 7 (OOPSLA1), April 2023. doi: 10.1145/3586030. URL https://doi.org/10.1145/3586030.

[11] S. Bassil and R.K. Keller. Software visualization tools: survey and analysis. In *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*, pages 7–17, 2001. doi: 10.1109/WPC.2001.921708.

[12] Joel Becker, Nate Rush, Beth Barnes, and David Rein. Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity. *arXiv preprint arXiv:2507.09089*, 2025. URL https://arxiv.org/abs/2507.09089.

[13] Tony Beltramelli. pix2code: Generating Code from a Graphical User Interface Screenshot. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450358972. doi: 10.1145/3220134.3220135. URL https://doi.org/10.1145/3220134.3220135.

[14] Abir Bouraffa, Gian-Luca Fuhrmann, and Walid Maalej. Developers' Visuo-Spatial Mental Model and Program Comprehension. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 1920–1932, Melbourne, Victoria, Australia, 2023. IEEE Press. ISBN 9781665457019. doi: 10.1109/ICSE48619.2023.00163. URL https://doi.org/10.1109/ICSE48619.2023.00163.

[15] Satish Chandra. AI in Software Engineering at Google: Progress and the Path Ahead (Invited Talk). In *Proceedings of the 1st ACM International Conference on AI-Powered Software*, AIware 2024, page 182, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706851. doi: 10.1145/3664646.3676277. URL https://doi.org/10.1145/3664646.3676277.

[16] Souti Chattopadhyay, Ishita Prasad, Austin Z. Henley, Anita Sarma, and Titus Barik. What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, page 1–12, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367080. doi: 10.1145/3313831.3376729. URL https://doi.org/10.1145/3313831.3376729.

[17] Jingye Chen, Yupan Huang, Tengchao Lv, Lei Cui, Qifeng Chen, and Furu Wei. TextDiffuser: diffusion models as text painters. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

[18] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde 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, Josh 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. Evaluating Large Language Models Trained on Code, 2021.

[19] Mauro Cherubini, Gina Venolia, Rob DeLine, and Amy J. Ko. Let's go to the whiteboard: how and why software developers use drawings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, page 557–566, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595935939. doi: 10.1145/1240624.1240714. URL https://doi.org/10.1145/1240624.1240714.

[20] Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios N. Angelopoulos, Tianle Li, Dacheng Li, Banghua Zhu, Hao Zhang, Michael I. Jordan, Joseph E. Gonzalez, and Ion Stoica. Chatbot arena: an open platform for evaluating LLMs by human preference. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

[21] Umut Cihan, Vahid Haratian, Arda İçöz, Mert Kaan Gül, Ömercan Devran, Emircan Furkan Bayendur, Baykal Mehmet Uçar, and Eray Tüzün. Automated Code Review in Practice. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 425–436, 2025. doi: 10.1109/ICSE-SEIP66354.2025.00043.

[22] Umut Cihan, Ali İçöz, Vahid Haratian, and Eray Tüzün. Evaluating Large Language Models for Code Review. *arXiv preprint arXiv:2505.20206*, 2025.

[23] OpenCompass Contributors. OpenCompass: A Universal Evaluation Platform for Foundation Models. https://github.com/open-compass/opencompass, 2023.

[24] Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo Li, and Zhi Jin. Codescore: Evaluating code generation by learning code execution. *ACM Transactions on Software Engineering and Methodology*, 34(3):1–22, 2025.

[25] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Evaluating Large Language Models in Class-Level Code Generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3639219. URL https://doi.org/10.1145/3597503.3639219.

[26] Marcelo d'Amorim, Rui Abreu, and Carlos Mello. Visual Sketching: From Image Sketches to Code. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 101–104, 2020.

[27] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. Selecting empirical methods for software engineering research. *Guide to advanced empirical software engineering*, pages 285–311, 2008.

[28] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Joshua B. Tenenbaum. Learning to infer graphics programs from hand-drawn images. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 6062–6071, Red Hook, NY, USA, 2018. Curran Associates Inc.

[29] Entrepreneur. AI Is Taking Over Coding at Microsoft, Google, and Meta. https://www.entrepreneur.com/business-news/ai-is-taking-over-coding-at-microsoft-google-and-meta/490896, 2025. Accessed: 2026-07-11.

[30] K. Anders Ericsson and Herbert A. Simon. *Protocol Analysis: Verbal Reports as Data*. The MIT Press, apr 1993. ISBN 9780262272391. doi: 10.7551/mitpress/5657.001.0001. URL http://dx.doi.org/10.7551/mitpress/5657.001.0001.

[31] Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. Out of the BLEU: how should we assess quality of the code generation models? *J. Syst. Softw.*, 203: 111741, 2023. doi: 10.1016/J.JSS.2023.111741. URL https://doi.org/10.1016/j.jss.2023.111741.

[32] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics, 2020.

[33] Luis Filipe Fernandes Gomes, Xin Zhou, David Lo, and Rui Abreu. Visual Loop: Bridging the Cognitive Gap in Software Development Through Visual-AI Collaboration. In *Proceedings of the 2026 IEEE/ACM Third International Conference on AI Foundation Models and Software Engineering*, FORGE '26, page 195–200, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400724770. doi: 10.1145/3793655.3793729. URL https://doi.org/10.1145/3793655.3793729.

[34] Paul Fraisse. Motor and verbal reaction times to words and drawings. *Psychonomic Science*, 12(6):235–236, jun 1968. ISSN 0033-3131. doi: 10.3758/bf03331287. URL http://dx.doi.org/10.3758/bf03331287.

[35] Robert France and Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering (FOSE '07)*, pages 37–54, 2007. doi: 10.1109/FOSE.2007.14.

[36] GitClear. AI Copilot Code Quality: 2025 Research. https://www.gitclear.com/ai_assistant_code_quality_2025_research, 2025. Accessed: 2026-07-11.

[37] GitHub. Include Diagrams in Your Markdown Files with Mermaid. https://github.blog/developer-skills/github/include-diagrams-markdown-files-mermaid/, 2022. Accessed: 2026-07-11.

[38] Luís Gomes. Transforming Ideas into Code: Visual Sketching for ML Development. In *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH 2023, page 10–12, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703843. doi: 10.1145/3618305.3623588. URL https://doi.org/10.1145/3618305.3623588.

[39] Luis F. Gomes, Vincent J. Hellendoorn, Jonathan Aldrich, and Rui Abreu. An Exploratory Study of ML Sketches and Visual Code Assistants . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1653–1664, Los Alamitos, CA, USA, May 2025. IEEE Computer Society. doi: 10.1109/ICSE55347.2025.00124. URL https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00124.

[40] Luís F. Gomes, Vincent Hellendoorn, Jonathan Aldrich, and Rui Abreu. An Exploratory Study of ML Sketches and Visual Code Assistants - Artifacts, may 2025. URL https://doi.org/10.5281/zenodo.15118203.

[41] Luís F. Gomes, Xin Zhou, David Lo, and Rui Abreu. VisDocSketcher: Towards Scalable Visual Documentation with Agentic Systems, 2025. URL https://arxiv.org/abs/2509.11942.

[42] Lucian Gonçales, Kleinner Farias, Bruno da Silva, and Jonathan Fessler. Measuring the Cognitive Load of Software Developers: A Systematic Mapping Study. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 42–52, 2019. doi: 10.1109/ICPC.2019.00018.

[43] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence, 2024. URL https://arxiv.org/abs/2401.14196.

[44] Junda He, Christoph Treude, and David Lo. LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road Ahead. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–30, 2025.

[45] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, page 1–12, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450359702. doi: 10.1145/3290605.3300500. URL https://doi.org/10.1145/3290605.3300500.

[46] Martin Hirzel. Low-Code Programming Models. *Commun. ACM*, 66(10):76–85, September 2023. ISSN 0001-0782. doi: 10.1145/3587691. URL https://doi.org/10.1145/3587691.

[47] Hui Huang, Yingqi Qu, Hongli Zhou, Jing Liu, Muyun Yang, Bing Xu, and Tiejun Zhao. On the Limitations of Fine-tuned Judge Models for LLM Evaluation, 2024. URL https://arxiv.org/abs/2403.02839.

[48] Ruanqianqian (Lisa) Huang, Savitha Ravi, Michael He, Boyu Tian, Sorin Lerner, and Michael Coblenz. How Scientists Use Jupyter Notebooks: Goals, Quality Attributes, and

Opportunities. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25, page 1243–1255. IEEE Press, 2025. ISBN 9798331505691. doi: 10.1109/ICSE55347.2025.00232. URL https://doi.org/10.1109/ICSE55347.2025.00232.

[49] John Hutchinson, Mark Rouncefield, and Jon Whittle. Model-driven engineering practices in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 633–642, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304450. doi: 10.1145/1985793.1985882. URL https://doi.org/10.1145/1985793.1985882.

[50] Yuan Jiang, Christian Kästner, and Shurui Zhou. Elevating Jupyter Notebook Maintenance Tooling by Identifying and Extracting Notebook Structures. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 399–403, 2022. doi: 10.1109/ICSME55016.2022.00047.

[51] Rodi Jolak, Maxime Savary-Leblanc, Manuela Dalibor, Andreas Wortmann, Regina Hebig, Juraj Vincur, Ivan Polasek, Xavier Le Pallec, Sébastien Gérard, and Michel R. V. Chaudron. Software engineering whispers: The effect of textual vs. graphical software design descriptions on software design communication. *Empirical Softw. Engg.*, 25(6): 4427–4471, nov 2020. ISSN 1382-3256. doi: 10.1007/s10664-020-09835-6. URL https://doi.org/10.1007/s10664-020-09835-6.

[52] Sayash Kapoor and Arvind Narayanan. Leakage and the Reproducibility Crisis in Machine-Learning-Based Science. *Patterns*, 4(9):100804, 2023.

[53] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, page 1–11, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356206. doi: 10.1145/3173574.3173748. URL https://doi.org/10.1145/3173574.3173748.

[54] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, UIST '20, page 140–151, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375146. doi: 10.1145/3379337.3415842. URL https://doi.org/10.1145/3379337.3415842.

[55] Daneul Kim, Jaeah Lee, and Jaesik Park. Improving Editability in Image Generation with Layer-wise Memory. In *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7889–7898, 2025. doi: 10.1109/CVPR52734.2025.00739.

[56] Jinman Kim and Joongjin Kook. A Research on the Impact of a Class Diagram Layout and Complexity on System Model Understandability. In *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, RACS '24, page 199–208, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400706066. doi: 10.1145/3649601.3698733. URL https://doi.org/10.1145/3649601.3698733.

[57] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. The emerging role of data scientists on software development teams. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 96–107, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450339001. doi: 10.1145/2884781.2884783. URL https://doi.org/10.1145/2884781.2884783.

[58] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. Data scientists in software teams: state of the art and challenges. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 585, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356381. doi: 10.1145/3180155.3182515. URL https://doi.org/10.1145/3180155.3182515.

[59] Jan H. Klemmer, Stefan Albert Horstmann, Nikhil Patnaik, Cordelia Ludden, Cordell Burton, Carson Powers, Fabio Massacci, Akond Rahman, Daniel Votipka, Heather Richter Lipford, Awais Rashid, Alena Naiakshina, and Sascha Fahl. Using AI Assistants in Software Development: A Qualitative Study on Security Practices and Concerns. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 2726–2740, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706363. doi: 10.1145/3658644.3690283. URL https://doi.org/10.1145/3658644.3690283.

[60] Jan H. Klemmer, Stefan Albert Horstmann, Nikhil Patnaik, Cordelia Ludden, Cordell Burton, Carson Powers, Fabio Massacci, Akond Rahman, Daniel Votipka, Heather Richter Lipford, Awais Rashid, Alena Naiakshina, and Sascha Fahl. Using AI Assistants in Software Development: A Qualitative Study on Security Practices and Concerns. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 2726–2740, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706363. doi: 10.1145/3658644.3690283. URL https://doi.org/10.1145/3658644.3690283.

[61] Amy J. Ko and Brad A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, page 151–158, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581137028. doi: 10.1145/985692.985712. URL https://doi.org/10.1145/985692.985712.

[62] Mario Krenn, Lorenzo Buffoni, Bruno Coutinho, Sagi Eppel, Jacob Gates Foster, Andrew Gritsevskiy, Harlin Lee, Yichao Lu, João P. Moutinho, Nima Sanjabi, Rishi Sonthalia, Ngoc Mai Tran, Francisco Valente, Yangxinyu Xie, Rose Yu, and Michael Kopp. Forecasting the future of artificial intelligence with machine learning-based link prediction in an exponentially growing knowledge network. *Nature Machine Intelligence*, 5 (11):1326–1335, oct 2023. ISSN 2522-5839. doi: 10.1038/s42256-023-00735-0. URL http://dx.doi.org/10.1038/s42256-023-00735-0.

[63] LangGraph. LangGraph. https://www.langchain.com/langgraph, 2025. Accessed: 2025-07.

[64] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th International Conference on*

Software Engineering, ICSE '06, page 492–501, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933751. doi: 10.1145/1134285.1134355. URL https://doi.org/10.1145/1134285.1134355.

[65] Kaixin Li, Yuchen Tian, Qisheng Hu, Ziyang Luo, and Jing Ma. MMCode: Evaluating Multi-Modal Code Large Language Models with Visually Rich Programming Problems, 2024. URL https://arxiv.org/abs/2404.09486.

[66] Pengzhi Li, Yan Pei, and Jianqiang Li. A comprehensive survey on design and application of autoencoder in deep learning. *Applied Soft Computing*, 138:110176, 2023.

[67] Xingjun Li, Yuanxin Wang, Hong Wang, Yang Wang, and Jian Zhao. NBSearch: Semantic Search and Visual Exploration of Computational Notebooks. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450380966. doi: 10.1145/3411764.3445048. URL https://doi.org/10.1145/3411764.3445048.

[68] Zheng Li, Aidan McGowan, Yan Liu, and Abdelwahab Hamou-Lhadj. UML Crisis! An Educational Perspective. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, FSE Companion '25, page 895–900, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400712760. doi: 10.1145/3696630.3727246. URL https://doi.org/10.1145/3696630.3727246.

[69] Charles X. Ling, Jin Huang, and Harry Zhang. AUC: a statistically consistent and more discriminating measure than accuracy. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, IJCAI'03, page 519–524, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

[70] Pan Lu, Hritik Bansal, Tony Xia, Jiacheng Liu, Chunyuan Li, Hannaneh Hajishirzi, Hao Cheng, Kai-Wei Chang, Michel Galley, and Jianfeng Gao. MathVista: Evaluating Mathematical Reasoning of Foundation Models in Visual Contexts, 2024. URL https://arxiv.org/abs/2310.02255.

[71] Yujie Lu, Dongfu Jiang, Wenhu Chen, William Wang, Yejin Choi, and Bill Yuchen Lin. WildVision Arena: Benchmarking Multimodal LLMs in the Wild, February 2024. URL https://huggingface.co/spaces/WildVision/vision-arena/.

[72] Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, Xiaoyin Che, Zhiyuan Liu, and Maosong Sun. RepoAgent: An LLM-Powered Open-Source Framework for Repository-level Code Documentation Generation. In Delia Irazu Hernandez Farias, Tom Hope, and Manling Li, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 436–464, Miami, Florida, USA, nov 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-demo.46. URL https://aclanthology.org/2024.emnlp-demo.46/.

[73] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning Word Vectors for Sentiment Analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. Association for

Computational Linguistics. URL http://www.aclweb.org/anthology/P11-1015.

[74] Vitthal Madane. Energy Consumption Time Series Dataset. https://www.kaggle.com/datasets/vitthalmadane/energy-consumption-time-series-dataset/data, 2022.

[75] Nicolas Mangano, Alex Baker, Mitch Dempsey, Emily Navarro, and André van der Hoek. Software design sketching with calico. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, page 23–32, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450301169. doi: 10.1145/1858996.1859003. URL https://doi.org/10.1145/1858996.1859003.

[76] Frank J. Massey. The Kolmogorov-Smirnov Test for Goodness of Fit. *Journal of the American Statistical Association*, 46(253):68–78, 1951. ISSN 01621459, 1537274X. URL http://www.jstor.org/stable/2280095.

[77] Patrick E. McKnight and Julius Najab. *Mann-Whitney U Test*, pages 1–1. John Wiley & Sons, Ltd, 2010. ISBN 9780470479216. doi: https://doi.org/10.1002/9780470479216.corpsy0524.

[78] MetaCTO. The Code Review Bottleneck in AI-Driven Development. https://www.metacto.com/blogs/code-review-bottleneck-ai-development, 2025. Accessed: 2026-07-11.

[79] Soumik Mohian and Christoph Csallner. PSDoodle: fast app screen search via partial screen doodle. In *Proceedings of the 9th IEEE/ACM International Conference on Mobile Software Engineering and Systems*, MOBILESoft '22, page 89–99, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393010. doi: 10.1145/3524613.3527816. URL https://doi.org/10.1145/3524613.3527816.

[80] Daniel Moody. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, 35(6):756–779, 2009. doi: 10.1109/TSE.2009.67.

[81] Mojtaba Mostafavi Ghahfarokhi, Arash Asgari, Mohammad Abolnejadian, and Abbas Heydarnoori. DistilKaggle: A Distilled Dataset of Kaggle Jupyter Notebooks. In *Proceedings of the 21st International Conference on Mining Software Repositories*, MSR '24, page 647–651, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705878. doi: 10.1145/3643991.3644882. URL https://doi.org/10.1145/3643991.3644882.

[82] OpenAI. GPT-4 Technical Report, 2023.

[83] OpenAI. DALL·E 2. https://openai.com/index/dall-e-2/, 2025. Accessed: 2025-07.

[84] OutSystems. OutSystems AI-Powered Low-Code Platform. Available at: https://www.outsystems.com/, 2025.

[85] Fernando Padoan, Ronnie De Souza Santos, and Rodrigo Pessoa Medeiros. Charting a Path to Efficient Onboarding: The Role of Software Visualization. In *Proceedings of the 2024 IEEE/ACM 17th International Conference on Cooperative and Human Aspects of Software*

Engineering, CHASE '24, page 133–143, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705335. doi: 10.1145/3641822.3641873. URL https://doi.org/10.1145/3641822.3641873.

[86] Allan Paivio. *Mental Representations: A dual coding approach*. Oxford University Press, Oxford, 09 1990. ISBN 9780195066661. doi: 10.1093/acprof:oso/9780195066661.001.0001. URL https://doi.org/10.1093/acprof:oso/9780195066661.001.0001.

[87] Allan Paivio and Kalman Csapo. Picture superiority in free recall: Imagery or dual coding? *Cognitive Psychology*, 5(2):176–206, 1973. ISSN 0010-0285. doi: https://doi.org/10.1016/0010-0285(73)90032-7. URL https://www.sciencedirect.com/science/article/pii/0010028573900327.

[88] Yunrim Park and Carlos Jensen. Beyond pretty pictures: Examining the benefits of code visualization for Open Source newcomers. In *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 3–10, 2009. doi: 10.1109/VISSOF.2009.5336433.

[89] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. *Commun. ACM*, 68(2):96–105, jan 2025. ISSN 0001-0782. doi: 10.1145/3610721. URL https://doi.org/10.1145/3610721.

[90] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. Program Comprehension and Code Complexity Metrics: An fMRI Study. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, page 524–536, Madrid, Spain, 2021. IEEE Press. ISBN 9781450390859. doi: 10.1109/ICSE43902.2021.00056. URL https://doi.org/10.1109/ICSE43902.2021.00056.

[91] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. Do Users Write More Insecure Code with AI Assistants? In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 2785–2799, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700507. doi: 10.1145/3576915.3623157. URL https://doi.org/10.1145/3576915.3623157.

[92] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR)*, pages 507–517, 2019.

[93] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 507–517, 2019. doi: 10.1109/MSR.2019.00077.

[94] Victor-Alexandru Pădurean and Adish Singla. Benchmarking Generative Models on Computational Thinking Tests in Elementary Visual Programming, 2024. URL https://arxiv.org/abs/2406.09891.

[95] Ketai Qiu, Niccolò Puccinelli, Matteo Ciniselli, and Luca Di Grazia. From Today's Code to Tomorrow's Symphony: The AI Transformation of Developer's Routine by 2030. *ACM*

Trans. Softw. Eng. Methodol., 34(5), may 2025. ISSN 1049-331X. doi: 10.1145/3709353. URL https://doi.org/10.1145/3709353.

[96] Sawan Rai, Ramesh Chandra Belwal, and Atul Gupta. A Review on Source Code Documentation. *ACM Trans. Intell. Syst. Technol.*, 13(5), jun 2022. ISSN 2157-6904. doi: 10.1145/3519312. URL https://doi.org/10.1145/3519312.

[97] Dhivyabharathi Ramasamy, Cristina Sarasua, Alberto Bacchelli, and Abraham Bernstein. Visualising data science workflows to support third-party notebook comprehension: an empirical study. *Empirical Softw. Engg.*, 28(3), mar 2023. ISSN 1382-3256. doi: 10.1007/s10664-023-10289-9. URL https://doi.org/10.1007/s10664-023-10289-9.

[98] Nikitha Rao, Jason Tsay, Kiran Kate, Vincent Hellendoorn, and Martin Hirzel. AI for Low-Code for AI. In *Proceedings of the 29th International Conference on Intelligent User Interfaces*, IUI '24, page 837–852, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705083. doi: 10.1145/3640543.3645203. URL https://doi.org/10.1145/3640543.3645203.

[99] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. CodeBLEU: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

[100] Rebecca Reuter, Theresa Stark, Yvonne Sedelmaier, Dieter Landes, Jurgen Mottok, and Christian Wolff. Insights in Students' Problems during UML Modeling. In *2020 IEEE Global Engineering Education Conference (EDUCON)*, page 592–600, Porto, Portugal, apr 2020. IEEE. doi: 10.1109/educon45650.2020.9125110. URL http://dx.doi.org/10.1109/EDUCON45650.2020.9125110.

[101] M.P. Robillard, W. Coelho, and G.C. Murphy. How effective developers investigate source code: an exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889–903, 2004. doi: 10.1109/TSE.2004.101.

[102] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. How do professional developers comprehend software? In *2012 34th International Conference on Software Engineering (ICSE)*, pages 255–265, 2012. doi: 10.1109/ICSE.2012.6227188.

[103] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. How do professional developers comprehend software? In *2012 34th International Conference on Software Engineering (ICSE)*, pages 255–265, 2012. doi: 10.1109/ICSE.2012.6227188.

[104] Joseph Romeo, Marco Raglianti, Csaba Nagy, and Michele Lanza. UML is Back. Or is it? Investigating the Past, Present, and Future of UML in Open Source Software . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 2342–2354, Los Alamitos, CA, USA, May 2025. IEEE Computer Society. doi: 10.1109/ICSE55347.2025.00155. URL https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00155.

[105] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron,

Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open Foundation Models for Code, 2024. URL https://arxiv.org/abs/2308.12950.

[106] Adam Rule, Aurélien Tabard, and James D. Hollan. Exploration and Explanation in Computational Notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, page 1–12, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356206. doi: 10.1145/3173574.3173606. URL https://doi.org/10.1145/3173574.3173606.

[107] Sadra Sabouri, Philipp Eibl, Xinyi Zhou, Morteza Ziyadi, Nenad Medvidovic, Lars Lindemann, and Souti Chattopadhyay. Trust Dynamics in AI-Assisted Development: Definitions, Factors, and Implications. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25, page 1678–1690. IEEE Press, 2025. ISBN 9798331505691. doi: 10.1109/ICSE55347.2025.00199. URL https://doi.org/10.1109/ICSE55347.2025.00199.

[108] Eduardo Santana de Almeida, Iftekhar Ahmed, and Andre van der Hoek. Let's Go to the Whiteboard (Again): Perceptions From Software Architects on Whiteboard Architecture Meetings . *IEEE Transactions on Software Engineering*, 49(10): 4773–4795, oct 2023. ISSN 1939-3520. doi: 10.1109/TSE.2023.3314410. URL https://doi.ieeecomputersociety.org/10.1109/TSE.2023.3314410.

[109] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. Hidden Technical Debt in Machine Learning Systems. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 28, pages 2503–2511, 2015.

[110] Akshay Sethi, Anush Sankaran, Naveen Panwar, Shreya Khare, and Senthil Mani. DLPaper2Code: Auto-Generation of Code From Deep Learning Research Papers. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), apr 2018. ISSN 2159-5399. doi: 10.1609/aaai.v32i1.12326. URL http://dx.doi.org/10.1609/aaai.v32i1.12326.

[111] Chenglei Si, Yanzhe Zhang, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. Design2Code: How Far Are We From Automating Front-End Engineering?, 2024. URL https://arxiv.org/abs/2403.03163.

[112] Janice Singer, Susan E Sim, and Timothy C Lethbridge. Software engineering data collection for field studies. *Guide to advanced empirical software engineering*, pages 9–34, 2008.

[113] Stack Overflow. 2025 Stack Overflow Developer Survey: AI. https://survey.stackoverflow.co/2025/ai/, 2025. Accessed: 2026-07-11.

[114] Benjamin Steenhoek, Md Mahbubur Rahman, Monoshi Kumar Roy, Mirza Sanjida Alam, Hengbo Tong, Swarna Das, Earl T. Barr, and Wei Le. To Err is Machine: Vulnerability Detection Challenges LLM Reasoning. *arXiv preprint arXiv:2403.17218*, 2024.

[115] M.-A. Storey. Theories, methods and tools in program comprehension: past, present and future. In *13th International Workshop on Program Comprehension (IWPC'05)*, pages 181–191, 2005. doi: 10.1109/WPC.2005.38.

[116] Gemini Team et al. Gemini: A Family of Highly Capable Multimodal Models, 2024.

[117] Thoughtworks. Spec-Driven Development: Unpacking 2025's New Engineering Practices. https://www.thoughtworks.com/en-us/insights/blog/agile-engineering-practices/spec-driven-development-unpacking-2025-new-engineering-practices, 2025. Accessed: 2026-07-11.

[118] Weixi Tong and Tianyi Zhang. CodeJudge: Evaluating Code Generation with Large Language Models, 2024. URL https://arxiv.org/abs/2410.02184.

[119] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI EA '22, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391566. doi: 10.1145/3491101.3519665. URL https://doi.org/10.1145/3491101.3519665.

[120] Jagoda Walny, Sheelagh Carpendale, Nathalie Henry Riche, Gina Venolia, and Philip Fawcett. Visual Thinking In Action: Visualizations As Used On Whiteboards. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2508–2517, 2011. doi: 10.1109/TVCG.2011.251.

[121] Jagoda Walny, Jonathan Haber, Marian Dörk, Jonathan Sillito, and Sheelagh Carpendale. Follow that sketch: Lifecycles of diagrams and sketches in software development. In *2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 1–8, 2011. doi: 10.1109/VISSOF.2011.6069462.

[122] April Yi Wang, Dakuo Wang, Jaimie Drozdal, Michael Muller, Soya Park, Justin D. Weisz, Xuye Liu, Lingfei Wu, and Casey Dugan. Documentation Matters: Human-Centered AI System to Assist Data Science Code Documentation in Computational Notebooks. *ACM Trans. Comput.-Hum. Interact.*, 29(2), January 2022. ISSN 1073-0516. doi: 10.1145/3489465. URL https://doi.org/10.1145/3489465.

[123] Ruiqi Wang, Jiyu Guo, Cuiyun Gao, Guodong Fan, Chun Yong Chong, and Xin Xia. Can LLMs Replace Human Evaluators? An Empirical Study of LLM-as-a-Judge in Software Engineering. *arXiv preprint arXiv:2502.06193*, 2025.

[124] Zijie J. Wang, Katie Dai, and W. Keith Edwards. StickyLand: Breaking the Linear Presentation of Computational Notebooks. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI EA '22, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391566. doi: 10.1145/3491101.3519653. URL https://doi.org/10.1145/3491101.3519653.

[125] Zijie J. Wang, David Munechika, Seongmin Lee, and Duen Horng Chau. SuperNOVA: Design Strategies and Opportunities for Interactive Visualization in Computational Notebooks. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, CHI EA '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703317. doi: 10.1145/3613905.3650848. URL https://doi.org/10.1145/3613905.3650848.

[126] Jingxuan Wei, Cheng Tan, Qi Chen, Gaowei Wu, Siyuan Li, Zhangyang Gao, Linzhuang Sun, Bihui Yu, and Ruifeng Guo. From Words to Structured Visuals: A Benchmark and Framework for Text-to-Diagram Generation and Editing. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 13315–13325, 2025.

[127] Nathaniel Weinman, Steven M. Drucker, Titus Barik, and Robert DeLine. Fork It: Supporting Stateful Alternatives in Computational Notebooks. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450380966. doi: 10.1145/3411764.3445527. URL https://doi.org/10.1145/3411764.3445527.

[128] John Wenskovitch, Jian Zhao, Scott Carter, Matthew Cooper, and Chris North. Albireo: An Interactive Tool for Visually Summarizing Computational Notebook Structure. In *2019 IEEE Visualization in Data Science (VDS)*, pages 1–10, 2019. doi: 10.1109/VDS48975.2019.8973385.

[129] Richard Wettel and Michele Lanza. Visualizing Software Systems as Cities. In *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 92–99, Banff, AB, Canada, 2007. IEEE. doi: 10.1109/VISSOF.2007.4290706.

[130] Richard Wettel and Michele Lanza. CodeCity: 3D visualization of large-scale software. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, page 921–922, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580791. doi: 10.1145/1370175.1370188. URL https://doi.org/10.1145/1370175.1370188.

[131] Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities: a controlled experiment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 551–560, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304450. doi: 10.1145/1985793.1985868. URL https://doi.org/10.1145/1985793.1985868.

[132] Martin Weyssow, Aton Kamanda, Xin Zhou, and Houari Sahraoui. CodeUltraFeedback: An LLM-as-a-Judge Dataset for Aligning Large Language Models to Coding Preferences. *ACM Trans. Softw. Eng. Methodol.*, 35(3), February 2026. ISSN 1049-331X. doi: 10.1145/3736407. URL https://doi.org/10.1145/3736407.

[133] Andrew J. O. Whitehouse, Murray T. Maybery, and Kevin Durkin. The development of the picture-superiority effect. *British Journal of Developmental Psychology*, 24(4): 767–773, nov 2006. ISSN 2044-835X. doi: 10.1348/026151005x74153. URL http://dx.doi.org/10.1348/026151005X74153.

[134] Laurie Williams and Robert R Kessler. The collaborative software process. In *International Conference on Software Engineering 2000*, 2000.

[135] Chengyue Wu, Yixiao Ge, Qiushan Guo, Jiahao Wang, Zhixuan Liang, Zeyu Lu, Ying Shan, and Ping Luo. Plot2Code: A Comprehensive Benchmark for Evaluating Multi-modal Large Language Models in Code Generation from Scientific Plots, 2024. URL https://arxiv.org/abs/2405.07990.

[136] Yifan Wu, Joseph M. Hellerstein, and Arvind Satyanarayan. B2: Bridging Code and Inter-

active Visualization in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, UIST '20, page 152–165, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375146. doi: 10.1145/3379337.3415851. URL https://doi.org/10.1145/3379337.3415851.

[137] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. Measuring Program Comprehension: A Large-Scale Field Study with Professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2018. doi: 10.1109/TSE.2017.2734091.

[138] Liwenhan Xie, Chengbo Zheng, Haijun Xia, Huamin Qu, and Chen Zhu-Tian. WaitGPT: Monitoring and Steering Conversational LLM Agent in Data Analysis with On-the-Fly Code Visualization. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, UIST '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706288. doi: 10.1145/3654777.3676374. URL https://doi.org/10.1145/3654777.3676374.

[139] Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. In-IDE Code Generation from Natural Language: Promise and Challenges. *ACM Trans. Softw. Eng. Methodol.*, 31(2), mar 2022. ISSN 1049-331X. doi: 10.1145/3487569. URL https://doi.org/10.1145/3487569.

[140] Tao Xu, Pengchuan Zhang, Qiuyuan Huang, Han Zhang, Zhe Gan, Xiaolei Huang, and Xiaodong He. Attngan: Fine-grained text to image generation with attentional generative adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1316–1324, 2018.

[141] Chenyang Yang, Rachel A. Brower-Sinning, Grace A. Lewis, and Christian Kästner. Data Leakage in Notebooks: Static Detection and Better Processes. In *Proc. 37th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE)*, 2022. doi: 10.1145/3551349.3556918.

[142] Dayu Yang, Antoine Simoulin, Xin Qian, Xiaoyi Liu, Yuwei Cao, Zhaopu Teng, and Grey Yang. DocAgent: A Multi-Agent System for Automated Code Documentation Generation, 2025.

[143] 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 Wang, and Ofir Press. SWE-bench Multimodal: Do AI Systems Generalize to Visual Software Domains? In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=riTiq3i21b.

[144] Yukang Yang, Dongnan Gui, Yuhui Yuan, Weicong Liang, Haisong Ding, Han Hu, and Kai Chen. GlyphControl: glyph conditional control for visual text generation. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

[145] Xiang Yue, Yuansheng Ni, Kai Zhang, Tianyu Zheng, Ruoqi Liu, Ge Zhang, Samuel Stevens, Dongfu Jiang, Weiming Ren, Yuxuan Sun, Cong Wei, Botao Yu, Ruibin Yuan, Renliang Sun, Ming Yin, Boyuan Zheng, Zhenzhu Yang, Yibo Liu, Wenhao Huang, Huan Sun, Yu Su, and Wenhu Chen. MMMU: A Massive Multi-discipline Multimodal

Understanding and Reasoning Benchmark for Expert AGI, 2024. URL https://arxiv.org/abs/2311.16502.

[146] Shehnaaz Yusuf, Huzefa Kagdi, and Jonathan I. Maletic. Assessing the Comprehension of UML Class Diagrams via Eye Tracking. In *15th IEEE International Conference on Program Comprehension (ICPC '07)*, pages 113–122, 2007. doi: 10.1109/ICPC.2007.10.

[147] Han Zhang, Tao Xu, Hongsheng Li, Shaoting Zhang, Xiaogang Wang, Xiaolei Huang, and Dimitris N Metaxas. Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 5907–5915, 2017.

[148] Junyi Zhang, Jiaqi Guo, Shizhao Sun, Jian-Guang Lou, and Dongmei Zhang. LayoutDiffusion: Improving Graphic Layout Generation by Discrete Diffusion Probabilistic Models. In *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 7192–7202, 2023. doi: 10.1109/ICCV51070.2023.00664.

[149] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena, 2023. URL https://arxiv.org/abs/2306.05685.

[150] Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. Codebertscore: Evaluating code generation with pretrained models of code. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 13921–13937. Association for Computational Linguistics, 2023. doi: 10.18653/V1/2023.EMNLP-MAIN.859. URL https://doi.org/10.18653/v1/2023.emnlp-main.859.

[151] Xin Zhou, Kisub Kim, Ting Zhang, Martin Weyssow, Luis F. Gomes, Guang Yang, and David Lo. An LLM-as-Judge Metric for Bridging the Gap with Human Evaluation in SE Tasks, 2025. URL https://arxiv.org/abs/2505.20854.

[152] Xin Zhou, Kisub Kim, Ting Zhang, Martin Weyssow, Luís F. Gomes, Guang Yang, Kui Liu, Xin Xia, and David Lo. SE-Jury: An LLM-as-Ensemble-Judge Metric for Narrowing the Gap with Human Evaluation in SE. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 2606–2618, 2025. doi: 10.1109/ASE63991.2025.00214.

[153] Terry Yue Zhuo. ICE-Score: Instructing Large Language Models to Evaluate Code. In Yvette Graham and Matthew Purver, editors, *Findings of the Association for Computational Linguistics: EACL 2024, St. Julian's, Malta, March 17-22, 2024*, pages 2232–2242. Association for Computational Linguistics, 2024. URL https://aclanthology.org/2024.findings-eacl.148.

[154] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2022, page 21–29, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392730. doi: 10.1145/3520312.3534864. URL

https://doi.org/10.1145/3520312.3534864.