"Bring the image back!" An Empirical Study of Visual Summaries in Code Review

Luis F. Gomes
Carnegie Mellon University
Pittsburgh, PA, USA
lfgomes@andrew.cmu.edu

Xin Zhou
Singapore Management University
Singapore
xinzhou.2020@phdcs.smu.edu.sg

Vincent Hellendoorn
Google Deepmind & CMU
Pittsburgh, PA, USA
vhellendoorn@google.com

Jonathan Aldrich
Carnegie Mellon University
Pittsburgh, PA, USA
jonathan.aldrich@cs.cmu.edu

Rui Abreu
Meta & FEUP
Porto, Portugal
rui@computer.org

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

Abstract

AI agents produce more and more code that developers are expected to validate. A critical challenge emerges: developers often accept generated code without sufficient scrutiny. While some defects trigger runtime errors or visibly incorrect behavior, other flaws leave the code executable and even produce plausible outputs. In Data Science, silent flaws such as data leakage or evaluation on improper data can lead to inaccurate decisions or a misinterpretation of results. Inspecting a large volume of code is inherently effortful and time-consuming. Visual representations offer a possible way to reduce this burden. We argue that visuals of generated code can help developers build a mental model of the code without requiring them to inspect it or read long textual explanations. An informal diagram may be more accessible and engaging than a textual description, while still surfacing structural properties of the code. To evaluate this hypothesis, we built JupyterDraw to automatically generate informal diagrams from Jupyter notebooks and conducted a controlled experiment with 26 participants. We investigated whether AI-generated diagrams improve developers' ability to detect such flaws compared to a textual overview alone. Participants reviewed two Jupyter notebooks, each containing exactly two planted methodological flaws, under a counterbalanced Latin-square design that assigns each participant one Visual (V) and one Textual (T) condition. The visual overview significantly improved detection: reviewers caught +0.50 more flaws per task ($d_z = 0.69$, 95% CI $[0.21, 0.79]$; Wilcoxon $p = 0.005$), and the share of reviews that caught *both* flaws (and would therefore deploy a correct model) rose from 29% to 67%. Participants also rated the diagram more helpful than text (median 5 vs. 4, $\Delta = +1.38$ on a 1–6 scale, $d_z = 0.64$, $p = .011$), and credited it as their primary detection channel, reasoning more structurally under it. The flaws reviewers missed in the text were not hiding inside variables and function calls, they lived in the connections between steps, a structure that diagrams expose. As agents write more and we read less of it, bringing images into review keeps developers accountable for what they ship.

CCS Concepts

• **Software and its engineering** → *Documentation*; • **Computing methodologies** → **Multi-agent planning**; • **Human-centered computing** → **Visualization systems and tools**; **Empirical studies in visualization**.

1 Introduction

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 [6, 47].

Prior work on AI-assisted coding has focused on *what* bugs and security issues these tools introduce and how often they occur [28, 37]. 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 [28, 37]. 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 [7, 34]. 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 [2, 30]. 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 [3, 8, 23, 38]. 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* [41] and dual-coding theory explain that visuals engage verbal and visual memory simultaneously, strengthening understanding and recall [25, 26]. 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 [24].

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 paper, 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.

1.1 Contributions

Our work focuses on the data-science domain and Jupyter notebooks, where developers reason about data flows, transformations, and ML components across cells [17, 18]; 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.

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 [32, 33]. Xia *et al.* show that comprehension dominates reviewers' working time [42]. 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" [2]. 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 [6], and is prone to over-trust plausible, executable output [19, 37].

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 [1], and that code visualizations help newcomers orient themselves and locate defects in unfamiliar systems [27]. 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 [16], and eye-tracking evidence shows that layout governs how readers visually search a diagram [46]. 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* [10, 41]. Dual-coding theory explains that visuals engage both verbal and visual memory systems simultaneously, strengthening understanding and recall [25, 26]. Software developers, whose work involves abstract, multi-level structures, are expected to perform such reasoning through text-based interfaces that provide little visual grounding [14]. 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 [20].

Studies on cognitive load in SE confirm that reducing cognitive burden (for instance, by providing high-level abstractions like

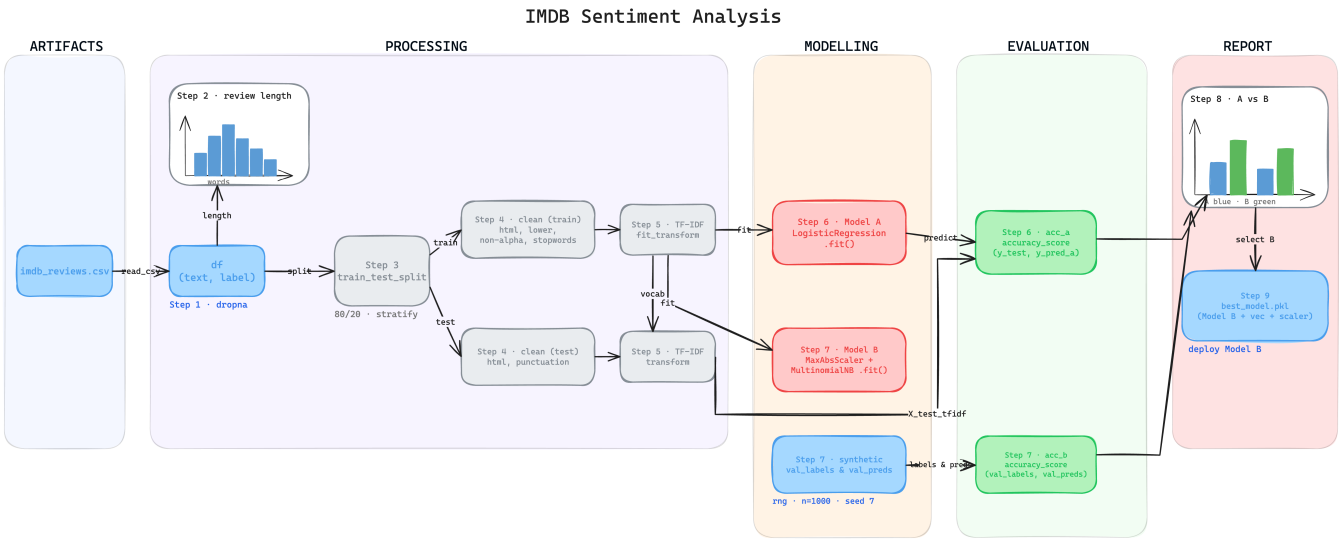

Figure 1: Example of a Jupyter notebook overview generated using JupyterDraw. These are the overviews shown to participants in the visual (V) condition.

software models rather than requiring developers to reconstruct structure from raw code) improves comprehension performance and reduces error rates [12]. 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.* [20] 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.* [23] identified that developers use low-detail abstractions and mix notations when sketching. In the same line, Walny *et al.* [38] 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.* [3] demonstrated that most sketches are informal, abstract, and created on analog media, proposing tools like SketchLink [5] and LivelySketches [4] to bridge analog and digital workflows. Building on Mangano’s observations regarding the frequent use of mixed notations in developer sketches, Gomes *et al.* [11] 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 [43], 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.* [40] generate interactive force-directed graphs that expose cell dependencies and variable flows, supporting self-reflection and collaboration. Ramasamy *et al.* [31] 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 [39] identifies key design trade-offs, including the tension between visualization richness and notebook integration.

A complementary line targets these flaws *automatically*: Yang *et al.* [44] 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 [19]: 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.* [34] 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.* [7] 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.* [13] 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

[29], 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.

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 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) [45]. 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.

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 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, 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

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

Figure 2: Example of the textual condition (T) overview.

different pipelines, and model B is never used for evaluation. In the textual version (Figure 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.

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.

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.

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.

4.3 Tasks

We designed the tasks to simulate a realistic deployment scenario in which a developer is asked to audit commited code. An example of such 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 [21]**: A binary text classification pipeline using TF-IDF and a Naive Bayes classifier, applied to movie reviews.
- **Notebook B - Blower Energy Forecasting [22]**: 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 1 summarizes the four planted flaws.

Table 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.

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 [15]. The two notebooks were designed to be of comparable 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

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.

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 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]

5.1 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)?**

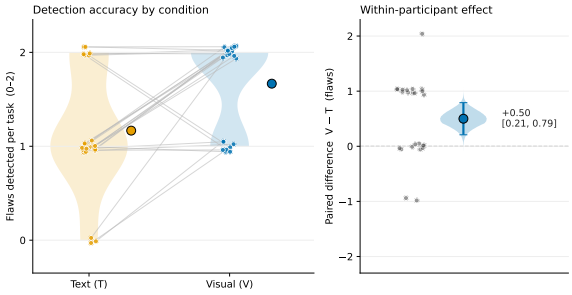

Figure 3: Flaws detected per task by condition ($n = 24$). Left: individual paired observations with bootstrapped condition means; right: the within-participant difference (V − T) with its bootstrap 95% CI.

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 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 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 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 $R^2 = 0.18$), 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.

[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.

Table 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).

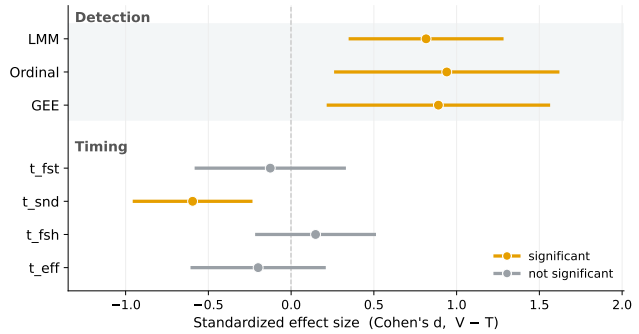

Figure 4: Standardized effect (Cohen's d, V − T) of the visual condition across every regression.

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 4, three of the four effects are small with a 95% CI spanning zero (all 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.*

5.2 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.

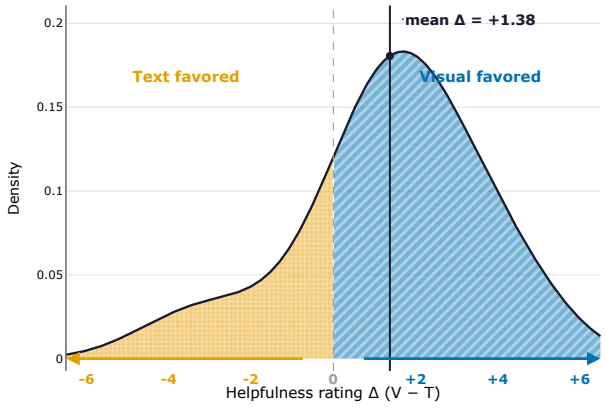

Figure 5: Distribution of the per-participant difference in perceived helpfulness.

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

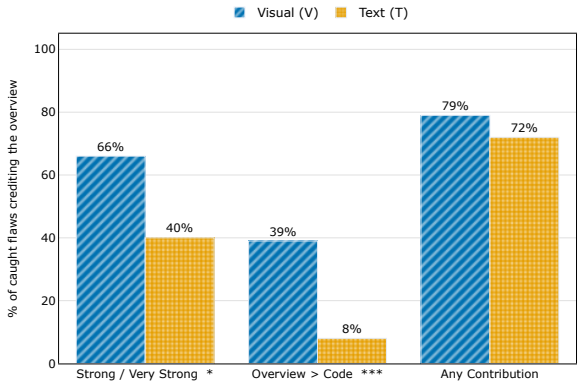

Figure 6: Attribution of caught flaws to each overview (V, T) by criterion, among the flaws participants detected (Fisher exact, $^{*}p < 0.1$, $^{***}p < 0.01$).

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 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) 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.

5.3 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 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.

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 [44], as drivers of the reproducibility crisis [15], and as structural debt that correctness checks miss [35]. LLMs share this blind spot: ~54.5% balanced accuracy on multi-step vulnerability detection [36] and correct code-review judgments only ~64–69% of the time [9]. 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 [17, 18], 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.

7 Conclusion

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://anonymous.4open.science/r/replication_package-CA2F/README.md.

References

[1] Dimitar Asenov, Otmar Hilliges, and Peter Müller. 2016. The Effect of Richer Visualizations on Code Comprehension. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI '16)*. Association for Computing Machinery, New York, NY, USA, 5040–5045. doi:10.1145/2858036.2858372

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

[3] Sebastian Baltes and Stephan Diehl. 2014. Sketches and diagrams in practice. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 530–541. doi:10.1145/2635868.2635891

[4] Sebastian Baltes, Fabrice Hollerich, and Stephan Diehl. 2017. 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)*. IEEE Computer Society, Los Alamitos, CA, USA, 94–98. doi:10.1109/VISSOFT.2017.24

[5] Sebastian Baltes, Peter Schmitz, and Stephan Diehl. 2014. Linking sketches and diagrams to source code artifacts. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 743–746. doi:10.1145/2635868.2661672

[6] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 78 (April 2023), 27 pages. doi:10.1145/3586030

[7] Souti Chattopadhyay, Ishita Prasad, Austin Z. Henley, Anita Sarma, and Titus Barik. 2020. 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* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3313831.3376729

[8] Mauro Cherubini, Gina Venolia, Rob DeLine, and Amy J. Ko. 2007. 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* (San Jose, California, USA) *(CHI '07)*. Association for Computing Machinery, New York, NY, USA, 557–566. doi:10.1145/1240624.1240714

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

[10] Paul Fraisse. 1968. Motor and verbal reaction times to words and drawings. *Psychonomic Science* 12, 6 (June 1968), 235–236. doi:10.3758/bf03331287

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

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

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

[14] 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. 2020. Software engineering whispers: The effect of textual vs. graphical software design descriptions on software design communication. *Empirical Softw. Engg.* 25, 6 (Nov. 2020), 4427–4471. doi:10.1007/s10664-020-09835-6

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

[16] Jinman Kim and Joongjin Kook. 2025. 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* (HABITA79 Pompeii, Pompei, Italy) *(RACS '24)*. Association for Computing Machinery, New York, NY, USA, 199–208. doi:10.1145/3649601.3698733

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

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

[19] 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. 2024. 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* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 2726–2740. doi:10.1145/3658644.3690283

[20] Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China) *(ICSE '06)*. Association for Computing Machinery, New York, NY, USA, 492–501. doi:10.1145/1134285.1134355

[21] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. 2011. Learning Word Vectors for Sentiment Analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Portland, Oregon, USA, 142–150. http://www.aclweb.org/anthology/P11-1015

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

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

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

[25] Allan Paivio. 1990. *Mental Representations: A dual coding approach*. Oxford University Press, Oxford. doi:10.1093/acprof:oso/9780195066661.001.0001

[26] Allan Paivio and Kalman Csapo. 1973. Picture superiority in free recall: Imagery or dual coding? *Cognitive Psychology* 5, 2 (1973), 176–206. doi:10.1016/0010-0285(73)90032-7

[27] Yunrim Park and Carlos Jensen. 2009. 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*. 3–10. doi:10.1109/VISSOF.2009.5336433

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

[29] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. A large-scale study about quality and reproducibility of jupyter notebooks. In *Proceedings of the 16th International Conference on Mining Software Repositories* (Montreal, Quebec, Canada) *(MSR '19)*. IEEE Press, 507–517. doi:10.1109/MSR.2019.00077

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

[31] Dhivyabharathi Ramasamy, Cristina Sarasua, Alberto Bacchelli, and Abraham Bernstein. 2023. Visualising data science workflows to support third-party notebook comprehension: an empirical study. *Empirical Softw. Engg.* 28, 3 (March 2023), 42 pages. doi:10.1007/s10664-023-10289-9

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

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

[34] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. Exploration and Explanation in Computational Notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3173574.3173606

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

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

[37] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. 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* (New Orleans, LA, USA) *(CHI EA '22)*. Association for Computing Machinery, New York, NY, USA, Article 332, 7 pages. doi:10.1145/3491101.3519665

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

[39] Zijie J. Wang, David Munechika, Seongmin Lee, and Duen Horng Chau. 2024. SuperNOVA: Design Strategies and Opportunities for Interactive Visualization in Computational Notebooks. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI EA '24)*. Association for Computing Machinery, New York, NY, USA, Article 304, 17 pages. doi:10.1145/3613905.3650848

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

[41] Andrew J. O. Whitehouse, Murray T. Maybery, and Kevin Durkin. 2006. The development of the picture-superiority effect. *British Journal of Developmental Psychology* 24, 4 (Nov. 2006), 767–773. doi:10.1348/026151005x74153

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

[43] Liwenhan Xie, Chengbo Zheng, Haijun Xia, Huamin Qu, and Chen Zhu-Tian. 2024. 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* (Pittsburgh, PA, USA) *(UIST '24)*. Association for Computing Machinery, New York, NY, USA, Article 119, 14 pages. doi:10.1145/3654777.3676374

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

[45] 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. 2025. SWE-bench Multimodal: Do AI Systems Generalize to Visual Software Domains?. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=riTiq3i21b

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

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