Understanding CI/CD Workflow Runs Through Interactive and Animated Visualizations

Pablo Valenzuela-Toledo
University of Bern
Switzerland
Universidad de La Frontera
Chile

Timo Kehrer
University of Bern
Switzerland

Sebastiano Panichella
University of Bern
Switzerland
AI4I
Italy

Abstract

Workflow runs record the executions of continuous integration and delivery specifications. They serve as the primary monitoring mechanism, enabling failure detection and integration reliability assessment. However, understanding run intricacies is far from trivial. Each run involves multiple interdependent execution units (*e.g.*, attempts, jobs, and steps) whose relationships are not immediately visible. Moreover, current tools provide limited support for detecting patterns across runs, particularly in projects with high workflow execution frequency. As a result, run monitoring remains constrained to fragmented views that are cumbersome to use.

We present a novel approach that supports workflow run comprehension through interactive and animated visualizations. Our approach enables developers to inspect end-to-end execution traces and to identify structural and behavioral patterns. We implemented a GitHub Actions prototype and analyzed runs from public repositories to illustrate its potential. Preliminary insights suggest that the approach is feasible and provides a starting point for workflow run comprehension analysis.

CCS Concepts

• **Software and its engineering → Continuous integration**; *Software visualization*; *Program comprehension*.

Keywords

GitHub Actions, CI/CD, workflow runs, visualization

1 Introduction

Workflow runs record the executions of continuous integration and delivery (CI/CD) specifications and serve as the primary monitoring mechanism [13, 30]. Each run encapsulates execution units, such as attempts, jobs, and steps, within a shared execution context. These runs provide developers with evidence to assess process reliability and the impact of changes on integration behavior.

However, understanding the intricacies of runs is far from trivial. Jobs may execute in parallel across distributed environments, and steps may vary according to matrix configuration strategies [16], complicating the reconstruction of how the overall execution unfolds [14]. Inferring the execution logic of such interleaved traces is inherently difficult and time-consuming when information from intermediate paths (*e.g.*, logs) is missing or incomplete [18]. In projects with thousands of runs, intertwined execution paths and incomplete traces obscure recurring behaviors and anomalous patterns [30].

Existing tools typically provide fragmented and predominantly textual representations of runs, which are difficult to inspect across different levels of granularity. Fine-grained views expose only partial information of individual executions. They offer limited interactivity, lack animation, and do not support end-to-end run tracing [13]. Coarse-grained views often provide summarized execution listings but hide the contextual details required to understand historical behavior [15]. As a result, run data remains dispersed, limiting developers' ability to achieve a comprehensive understanding.

Prior research has identified several limitations in how developers understand runs. Diagnosing failures remains difficult, as developers still lack efficient means to uncover their underlying causes [37, 38]. Resource usage is also hard to assess, leading to inefficient allocation of time and computational resources [7, 39]. Furthermore, failures across histories are challenging to analyze, since available views are too fragmented to reveal patterns or support effective troubleshooting [19, 42]. In addition, workflows evolve continuously through debugging, refactoring, and reconfiguration [35], entailing significant maintenance costs that are difficult to monitor without appropriate tooling [30, 34].

We introduce the idea of reifying runs as first-class visual entities. Our approach models each execution unit within a unified Workflow Run Domain Model that captures its full context. From this model, we create interactive and animated visualizations that reveal how executions unfold [24]. To illustrate this vision, we implemented an early prototype named *Run-Visualizer* and used it to explore workflow runs from GitHub Actions (GA), a widely used CI/CD platform [11]. The data, detailed approach, scripts, and videos of the paper are publicly available in the replication package [1]. DOI 10.5281/zenodo.18272174

2 Related Work

Prior work on workflow runs focuses on build and test failures, not visual comprehension. Rausch *et al.* analyzed runs in open-source projects, linking build outcomes to task complexity and tooling [27]. Pinto *et al.* examined cultural and behavioral causes of failures, such as overconfidence and speed pressure [26]. Zhang *et al.* [41] analyzed millions of builds to identify recurring compiler error patterns. Other works explored automated and predictive techniques for failure diagnosis, understanding and prevention [4, 21, 28, 29, 34, 36, 40, 43]. Valenzuela-Toledo *et al.* studied how large language models explain GA run failures [37].

Research on visual representations of workflow runs is still limited. Existing platforms such as GitHub provide coarse and mostly static execution views that summarize run outcomes but omit how execution units compose execution traceability [9]. The most related approach, *μPrintGen* [2], focuses on visually locate conclusion states within log files rather than visualize the workflow run units.

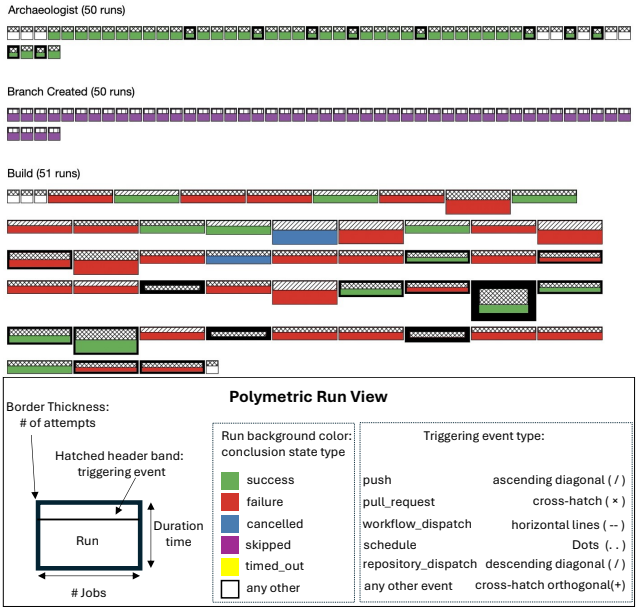

Figure 1: Polymetric Run View. https://bit.ly/4pddOWO

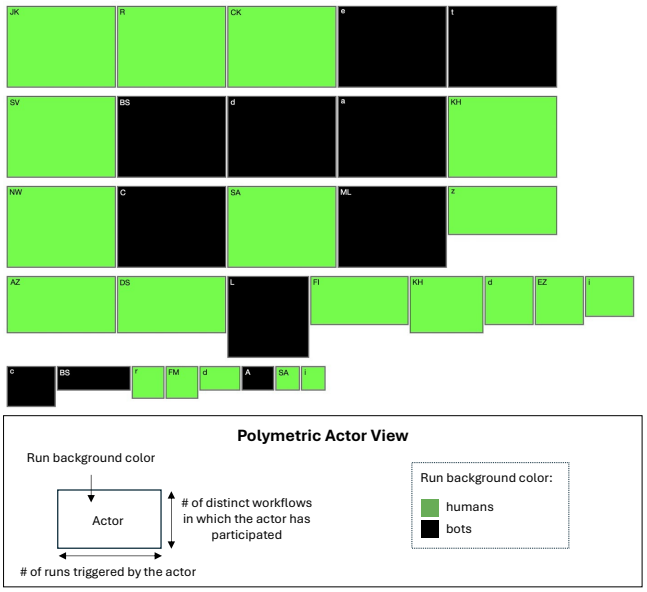

Figure 2: Polymetric Actor View. https://bit.ly/4p76tIu

Our vision takes a different direction. It reifies workflow runs into interactive and animated visualizations, enabling developers to explore them at multiple levels of detail and reconstruct complete execution traces.

3 Approach

Our approach consists of three stages: (i) data extraction, (ii) model population, and (iii) visualization generation.

3.1 Data Extraction

We mine structured and unstructured run data from GitHub using its REST API. Execution metadata files in JSON format describe the run units (*e.g.*, jobs, steps). Log files record execution traces, including command outputs, timestamps, and status messages.

3.2 Workflow Run Domain Model Population

The domain model represents the internal structure of a GA run. It includes two main groups of entities: *execution* and *context*.

Execution entities describe how the execution unfolds. A run consists of one or more attempts, corresponding to individual executions of a workflow. Each attempt groups several jobs, which may execute multiple steps concurrently or sequentially. Each step defines the commands to be executed. Log files record the execution of steps in text form. Logs are divided into sequential text fragments, or chunks, linked to one or more conclusion states (*e.g.*, success or failure).

Contextual entities capture detailed information about the circumstances of a run failure. A *failure-context* object connects a run failure to its related execution units, including the *workflow*, *attempt*, *job*, *step*, *log*, *commit*, and *author*. It forms a cross-referential structure that encapsulates the full trace of the failure run. A *commit* represents the code change that triggered the run, while the *commit author* identifies the developer responsible for that change. Several failure-context are grouped into a *failure-context collection*.

3.3 Visualization Generation

We developed *Run-Visualizer*, a domain-specific tool for visualizing GitHub Actions runs in the *Glamorous Toolkit* [17], supporting *coarse-* and *fine-grained* views.

3.3.1 Coarse-grained views. These views present two polymetric visualizations [20] of a run: *polymetric runs* and *polymetric actors*.

The *polymetric runs* view provides an overview of execution activity across workflows (Figure 1). Each run is represented as a rectangle arranged chronologically. Runs of the same workflow appear in continuous rows to preserve grouping and temporal order. The width encodes the number of jobs. The height represents the execution time, normalized across the dataset. The background color denotes the outcome (*e.g.*, green for success, red for failure), and the border thickness indicates the number of attempts. A hatched header band marks the triggering event (*e.g.*, push or pull request).

The *polymetric actors* view shifts the focus from executions to the actors behind them (Figure 2). Each rectangle represents an *actor* (*i.e.*, the committer). Actors are identified by the user name and email extracted from the head commit of each run. The width encodes the number of runs triggered by the actor. The height represents the count of different workflows the actor has triggered at least once. The color differentiates bots from human actors (black for bots, green for humans). Actor initials appear within each rectangle for quick identification. Rectangles are arranged in rows sorted by area, computed as the product of runs and workflows. This layout emphasizes actors with broader participation.

3.3.2 Fine-grained views. These views provide detail of the workflow run execution units: *run-anatomy* and *run-failure-context collection*.

The *run-anatomy* view describes the internal structure of a single run (Figure 3). It is composed of five coordinated panels: *Attempts*, *Jobs*, *Steps*, *Log*, and *Commit*. The *Attempts* panel summarizes the main execution units. Each attempt is displayed as a rectangle whose

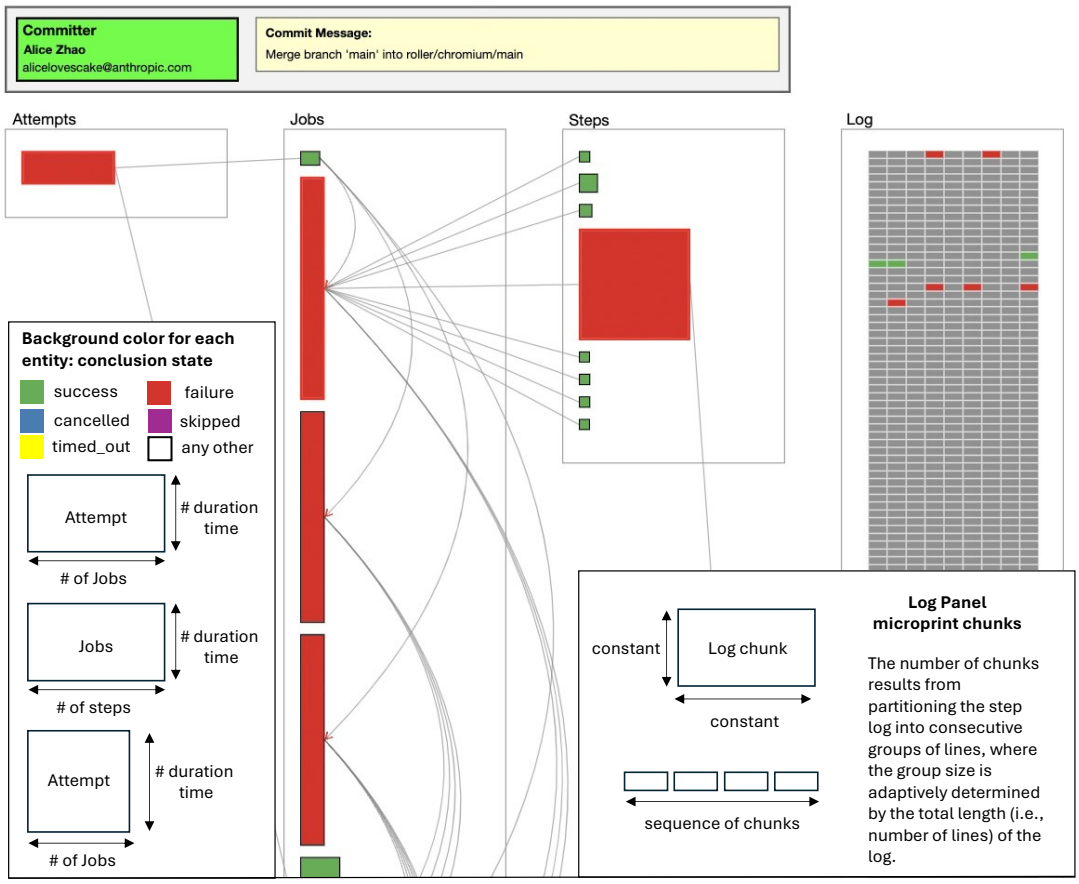

Figure 3: Run Anatomy View. https://bit.ly/49Uu8Hx

height represents total duration and width reflects the number of jobs it contains. The background color encodes the conclusion state (*e.g.*, success, failure) and is used consistently across entities. Within each attempt, jobs are grouped according to the workflow dependency structure and arranged in execution order. Each job is shown as a rectangle where height corresponds to execution time and width indicates the number of steps it includes. Linking edges depict dependencies between jobs and their constituent steps. The *Steps* panel decomposes each job into smaller rectangles aligned along a shared axis. Step height and width represent step duration.

The *Log* panel adopts a microprint-based approach [12]. This design enables a log overview without requiring developers to read their full textual content. It displays the textual output of the run as uniformly sized *chunks*. The number of chunks results from partitioning the log at step level. Chunks are displayed as small fixed-size rectangles, sorted in the greed by temporal progression.

The *commit* panel links each run to the specific change and developer that initiated it. The color distinguishes bots (black) from human actors (green), and the commit message is displayed in yellow for differentiation purposes.

The *run-anatomy* view provides animated transitions to illustrate the temporal flow of execution. Selecting an attempt reveals its jobs, and expanding a job displays its steps in execution order.

The *run-failure-context collection* view supports the exploration of multiple failure-context objects (Figure 4). It integrates four perspectives: *failure-context collection*, *author–workflow matrix*, *failures by step*, and *failures by commit*.

The *failure-context collection* view allows comparing failures-context objects. The *author–workflow matrix* links committers to the workflows in which their failures occurred, showing both the frequency of failures and the most common steps associated with each of them. This relation connects developer activity to the workflow components where instability is concentrated. The *failures by step* view lists steps sorted by their failure frequency, allowing developers to identify the most error-prone steps across different workflows. Finally, the *failures by commit* view links failures to the commits that triggered them. It shows the author and commit message to reveal which code changes are most often associated with failures.

4 Exploratory Showcase

We present an exploratory showcase using runs extracted from the *electron*[1] and *vercel*[2] repositories. We used *electron* for views requiring diverse execution histories and *vercel* for the *failure-context*

[1] https://github.com/electron/electron/
[2] https://github.com/vercel/vercel/

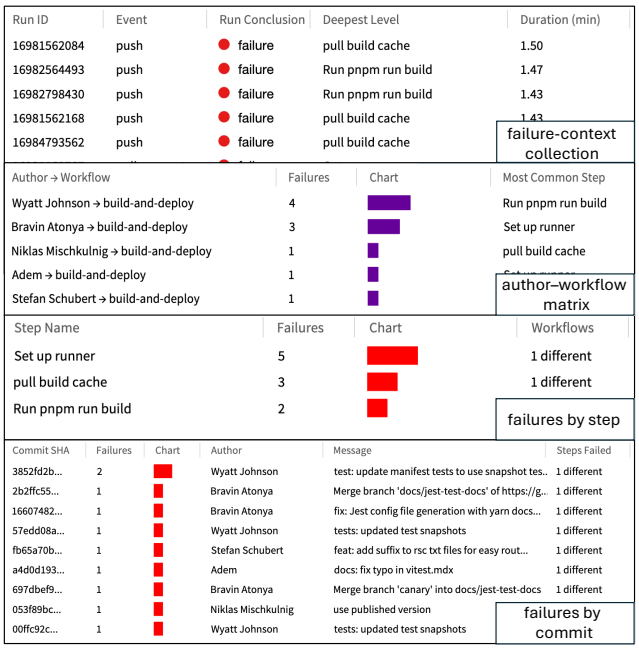

Figure 4: Failure-Context View. https://bit.ly/47KRpuj

collection view due to its recurring failure patterns. Since interactions and animations are difficult to convey in textual form, we provide videos for each case.

The *polymetric runs* view (Figure 1) illustrates the diversity of execution behaviors across workflows. Although the repository contains twenty-two workflows, this excerpt focuses on three representative ones: *Archaeologist*, *Branch Created*, and *Build*. From these visual traces, we can discern distinct stability and failure dynamics. *Archaeologist* appears predominantly green, reflecting sustained success with only a few retries. In contrast, *Branch Created* displays a uniform sequence of purple rectangles, revealing its regular scheduled executions. For the *Build* workflow, we observe alternating red and green rectangles grouped in dense clusters, interspersed with hatched headers that denote heterogeneous triggering events. These clusters expose bursts of recurrent failures followed by short recovery periods. Thicker black borders around some runs indicate multiple attempts.

The *polymetric actors* view (Figure 2) shows that a few contributors dominate run activity in the *electron* repository, visible as the largest rectangles. Surrounding them, numerous smaller rectangles represent occasional participants with limited engagement. Several large black rectangles correspond to bot accounts, indicating substantial automated execution. This pattern reveals a strong reliance on automation to sustain repository activity.

Using the *run-anatomy* view (Figure 3), we observe the trace of a run's execution. Because the full trace involves numerous execution units, the figure shows only a representative portion. The view depicts a run (ID 18023989912) associated with the *Build* workflow, triggered by a *pull_request* event. The run took about seven minutes and concluded with a failure after a single attempt. At the top, the *Commit* panel links the execution to the specific code change and the developer who initiated it. Below, the remaining panels show how the execution unfolded across jobs, steps, and logs. Most jobs terminated unsuccessfully. In the *Steps* panel, a single failure step dominates the execution time. The *Log* panel shows a dense grid of gray *chunks*, indicating large portions without explicit conclusions.

In the *failure-context collection* view (Figure 4), two clusters emerge: *push* runs failing during cache or build steps lasting about one and a half minutes, and *pull_request* runs failing almost instantly during runner setup. The contrast suggests distinct failure causes between automated and contributor-triggered executions. In the *author–workflow matrix* view, failures concentrate around two contributors (original user names anonymized): Bob and Alice. Bob's runs mostly fail at the *Run pnpm run build* step. Alice's runs are dominated by *Set up runner* errors. The *Failures by Step* view shows that most errors originate from *Set up runner*, followed by *pull build cache* and *Run pnpm run build*, indicating initialization and caching as primary sources of instability. Finally, the *Failures by Commit* view links many failures to test updates or documentation merges. Minor maintenance changes can repeatedly trigger failures in the *build-and-deploy* workflow.

5 Discussion

Our visualizations show that reifying workflow runs as visual entities is feasible to support comprehensive analysis. Processing all runs and logs took minutes on standard hardware, and view switching remained responsive. The visualizations revealed execution behaviors hard to observe in the standard GitHub Actions interface. For example, failures cannot be traced end-to-end in a single view, as jobs and logs are shown separately. They also revealed unanticipated patterns, such as short-lived workflow instabilities, visible only through visual encoding. These observations align with prior software visualization findings and indicate that visual representations support the detection of unexpected patterns and structural relationships [8, 22, 23].

Animated transitions in the *run-anatomy* view support execution-trace reconstruction, aligning with prior research [3, 5, 25, 31–33].

6 Conclusion and Future Work

We introduced an approach to understanding CI/CD workflow runs through interactive and animated visualizations. Initial results suggest that visual encodings can convey execution behavior that is difficult to grasp in existing CI interfaces. Building on these observations, the findings will inform an empirical evaluation with active GitHub Actions workflow maintainers. Participants will perform diagnostic tasks focused on failure localization and workflow stability. The study will contrast our approach with standard GitHub views [6, 10].

Acknowledgments. The authors would like to thank the Swiss (CH) Group for Original and Outside-the-box Software Engineering (CHOOSE) for providing financial support for the conference travel.

References

[1] 2025. *Replication Package – GitHub Actions Run Analysis*. doi:10.5281/zenodo.17635512
[2] Sebastian Alfaro, Alexandre Bergel, and Jocelyn Simmonds. 2023. mu PrintGen: Supporting Workflow Logs Analysis Through Visual Microprint. In *2023 IEEE Working Conference on Software Visualization*. IEEE, 45–49.
[3] Carmen Armenti and Michele Lanza. 2024. Using animations to understand commits. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 660–665.
[4] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, my tests broke the build: An explorative analysis of travis ci with github. In *2017 IEEE/ACM 14th International conference on mining software repositories (MSR)*. IEEE, 356–367.

[5] Sandra Berney and Mireille Bétrancourt. 2016. Does animation enhance learning? A meta-analysis. *Computers & Education* 101 (2016), 150–167.
[6] Alison Fernandez Blanco, Alexandre Bergel, Juan Pablo Sandoval Alcocer, and Araceli Queirolo Córdova. 2022. Visualizing memory consumption with Vismep. In *2022 Working Conference on Software Visualization (VISSOFT)*. IEEE, 108–118.
[7] Islem Bouzenia and Michael Pradel. 2024. Resource Usage and Optimization Opportunities in Workflows of GitHub Actions. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.
[8] Noptanit Chotisarn, Leonel Merino, Xu Zheng, Supaporn Lonapalawong, Tianye Zhang, Mingliang Xu, and Wei Chen. 2020. A systematic literature review of modern software visualization. *Journal of Visualization* 23, 4 (2020), 539–558.
[9] Electron Contributors. 2025. Electron GitHub Actions Run 19210705401. https://github.com/electron/electron/actions/runs/19210705401. Accedido: February 6, 2026.
[10] Andreina Cota Vidaurre, Evelyn Cusi Lopez, Juan Pablo Sandoval Alcocer, and Alexandre Bergel. 2022. TestEvoViz: visualizing genetically-based test coverage evolution. *Empirical Software Engineering* 27, 7 (2022), 184.
[11] Alexandre Decan, Tom Mens, Pooya Rostami Mazrae, and Mehdi Golzadeh. 2022. On the use of github actions in software development repositories. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 235–245.
[12] Stéphane Ducasse, Michele Lanza, and Romain Robbes. 2005. Multi-level method understanding using microprints. In *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, 1–6.
[13] GitHub. 2024. Use the Visualization Graph. https://docs.github.com/en/actions/how-tos/monitor-workflows/use-the-visualization-graph GitHub Documentation.
[14] GitHub. 2025. Running variations of jobs in a workflow. https://docs.github.com/en/actions/how-tos/write-workflows/choose-what-workflows-do/run-job-variations. GitHub Actions Documentation.
[15] GitHub. 2025. Viewing workflow run history. https://docs.github.com/en/actions/how-tos/monitor-workflows/view-workflow-run-history. GitHub Actions Documentation.
[16] Inc. GitHub. 2025. *Run job variations*. GitHub Documentation. https://docs.github.com/en/actions/how-tos/write-workflows/choose-what-workflows-do/run-job-variations
[17] Glamorous Toolkit Team. 2025. Glamorous Toolkit. https://gtoolkit.com/. Accessed: 2025-10-1.
[18] Shilin He, Xu Zhang, Pinjia He, Yong Xu, Liqun Li, Yu Kang, Minghua Ma, Yining Wei, Yingnong Dang, Saravanakumar Rajmohan, et al. 2022. An empirical study of log analysis at Microsoft. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1465–1476.
[19] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. 2014. Why do automated builds break? an empirical study. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 41–50.
[20] Michele Lanza. 2004. CodeCrawler-polymetric views in action. In *Proceedings. 19th International Conference on Automated Software Engineering, 2004.* IEEE, 394–395.
[21] Yiling Lou, Zhenpeng Chen, Yanbin Cao, Dan Hao, and Lu Zhang. 2020. Understanding build issue resolution in practice: symptoms and fix patterns. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 617–628.
[22] Leonel Merino, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz. 2018. A systematic literature review of software visualization evaluation. *Journal of systems and software* 144 (2018), 165–180.
[23] Tamara Munzner. 2014. *Visualization analysis and design*. CRC Press.
[24] Oscar Marius Nierstrasz and Tudor Gîrba. 2024. Moldable Development Patterns. In *Proceedings of the 29th European Conference on Pattern Languages of Programs, People, and Practices*. 1–14.
[25] Fred Paas, Alexander Renkl, and John Sweller. 2003. Cognitive load theory and instructional design: Recent developments. *Educational psychologist* 38, 1 (2003), 1–4.
[26] Gustavo Pinto, Fernando Castor, Rodrigo Bonifacio, and Marcel Rebouças. 2018. Work practices and challenges in continuous integration: A survey with Travis CI users. *Software: Practice and Experience* 48, 12 (2018), 2223–2236.
[27] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. 2017. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 345–355.
[28] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. 2020. Predicting continuous integration build failures using evolutionary search. *Information and Software Technology* 128 (2020), 106392.
[29] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. 2022. Improving the prediction of continuous integration build failures using deep learning. *Automated Software Engineering* 29, 1 (2022), 21.
[30] Jadson Santos, Daniel Alencar da Costa, Shane McIntosh, and Uirá Kulesza. 2024. On the Need to Monitor Continuous Integration Practices–An Empirical Study. *arXiv preprint arXiv:2409.05101* (2024).
[31] Wolfgang Schnotz and Christian Kürschner. 2007. A reconsideration of cognitive load theory. *Educational psychology review* 19, 4 (2007), 469–508.
[32] John Sweller, Jeroen JG Van Merrienboer, and Fred GWC Paas. 1998. Cognitive architecture and instructional design. *Educational psychology review* 10, 3 (1998), 251–296.
[33] Barbara Tversky, Julie Bauer Morrison, and Mireille Betrancourt. 2002. Animation: can it facilitate? *International journal of human-computer studies* 57, 4 (2002), 247–262.
[34] RP Vale. 2024. The Hidden Costs of Automation: An Empirical Study on GitHub Actions Workflow Maintenance Replication Package. https://zenodo.org/records/12485092. Accessed: 2024-06-23.
[35] Pablo Valenzuela-Toledo and Alexandre Bergel. 2022. Evolution of GitHub action workflows. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 123–127.
[36] Pablo Valenzuela-Toledo, Alexandre Bergel, Timo Kehrer, and Oscar Nierstrasz. 2023. EGAD: A moldable tool for GitHub Action analysis. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories*. IEEE, 260–264.
[37] Pablo Valenzuela-Toledo, Chuyue Wu, Sandro Hernández, Alexander Boll, Roman Machacek, Sebastiano Panichella, and Timo Kehrer. 2025. Explaining GitHub Actions Failures with Large Language Models: Challenges, Insights, and Limitations. doi:10.5281/zenodo.14750197
[38] Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C Gall. 2018. Un-break my build: Assisting developers with build repair hints. In *Proceedings of the 26th Conference on Program Comprehension*. 41–51.
[39] Nimmi Weeraddana, Mahmoud Alfadel, and Shane McIntosh. 2024. Characterizing timeout builds in continuous integration. *IEEE Transactions on Software Engineering* 50, 6 (2024), 1450–1463.
[40] Jing Xia and Yanhui Li. 2017. Could we predict the result of a continuous integration build? An empirical study. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 311–315.
[41] Chen Zhang, Bihuan Chen, Linlin Chen, Xin Peng, and Wenyun Zhao. 2019. A large-scale empirical study of compiler errors in continuous integration. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 176–187.
[42] Lianyu Zheng, Shuang Li, Xi Huang, Jiangnan Huang, Bin Lin, Jinfu Chen, and Jifeng Xuan. 2025. Why Do GitHub Actions Workflows Fail? An Empirical Study. *ACM Transactions on Software Engineering and Methodology* (2025).
[43] Celal Ziftci and Jim Reardon. 2017. Who broke the build? Automatically identifying changes that induce test failures in continuous integration at google scale. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 113–122.