Quality Commits: Explicit Software Quality Management in Commit Messages

Marco Ehl[1,*], Amir Shayan Ahmadian[1], Katharina Großer[1], Duaa Adel Ali Elsofi[1], Marc Herrmann[2], Alexander Specht[2], Kurt Schneider[2] and Jan Jürjens[1,3]

[1]Institute for Software Technology, University of Koblenz, Universitätsstraße 1, Koblenz, 56070, Germany
[2]Software Engineering Group, Leibniz University Hannover, Welfengarten 1, Hannover, 30167, Germany
[2]Fraunhofer ISST, Speicherstraße 6, Dortmund, 44147, Germany

Abstract

Software engineering (SE) gets faster and more collaborative with the availability of AI-coding agents. This drastically increases the volume of code changes made by AI that challenges the ability of humans-in-the-loop to manage software quality and maintain traceability. Collaboration in SE is often done via version-controlled code repositories, where a change to the repository is recorded as a commit. Commit messages serve as a record for documenting code changes. Our pre-study reveals that fewer than 15% of AI-authored and fewer than 20% of human-created commit messages on GitHub explicitly reference software quality characteristics. Existing formats for commit messages, such as Conventional Commits, focus on the intent rather than on the quality impact. This leaves a gap between quality requirements and implementation, as changes to non-functional requirements remain implicit and untraceable in the code repository. We propose Quality Commits, a lightweight, backward-compatible extension to Conventional Commits to explicitly tag improved and decreased quality characteristics based on the ISO/IEC 25010 software product quality model. We demonstrate how this format helps the human-in-the-loop with problem triage, mitigation, software testing, review and trace software quality, and enables automated quality management such as patch notes generation and software quality prioritization.

Keywords

Software quality, non-functional requirements, collaboration, human-in-the-loop, commit message

1. Introduction

Software engineering (SE) has evolved from classical SE 1.0 without AI-agents, SE 2.0 where AI-agents generate functions and classes on demand, to SE 3.0 where AI-coding agents plan features autonomously and change complete repositories [1]. AI-agents produce new or changed code fast. Figure 1 illustrates the evolution of SE, its collaboration between human and AI, the resulting increased code frequency, the resulting problem and how our contribution helps developers in their SE tasks. The human-in-the-loop often takes the role of judging if the new code changes should be accepted. The human-in-the-loop does not necessarily have a software engineering background, since the AI-agents lower the bar for developing software. Humans and coding agents collaborate and communicate by loops of prompts and judging and accepting changes, typically via reviewing and accepting pull requests, which are a collection of commits [1]. Since the goal of software engineering is to produce high quality software, every code change should improve the software quality, for example improve functional completeness, security or resource utilization. In SE, especially in Requirements Engineering (RE), traceability is important [2]. However, a critical traceability gap remains between high-level

Joint Proceedings of REFSQ-2026 Workshops, Doctoral Symposium, Posters & Tools Track, and Education and Training Track. Co-located with REFSQ 2026. Poznan, Poland, March 23-26, 2026

*Corresponding author.
mehl@uni-koblenz.de (M. Ehl); ahmadian@uni-koblenz.de (A. S. Ahmadian); grosser@uni-koblenz.de (K. Großer); delsofi@uni-koblenz.de (D. A. A. Elsofi); marc.herrmann@inf.uni-hannover.de (M. Herrmann); alexander.specht@inf.uni-hannover.de (A. Specht); kurt.schneider@inf.uni-hannover.de (K. Schneider); juerjens@uni-koblenz.de (J. Jürjens)
0009-0004-9319-4621 (M. Ehl); 0000-0002-0376-3869 (A. S. Ahmadian); 0000-0003-4532-0270 (K. Großer); 0009-0005-0983-0558 (D. A. A. Elsofi); 0000-0002-3951-3300 (M. Herrmann); 0000-0003-0783-8335 (A. Specht); 0000-0002-7456-8323 (K. Schneider); 0000-0002-8938-0470 (J. Jürjens)

non-functional requirements and low-level code changes. Commit messages serve as the lowest level traceability link between quality requirements and implementation [3]. The central challenge in the era of SE 3.0 is managing software quality effectively when the human-in-the-loop has to review, test, operate, and manage a large volume of code changes when collaborating with AI-agents [4]. This is a problem, because software quality is only covered partly by existing and applied formats for documenting code changes in commit messages. In a pre-study utilizing the AIDev dataset [1], we analyzed 33596 AI-generated and 6618 human-generated pull requests from GitHub[1] projects with at least 300 stars for the presence of ISO/IEC 25010 quality characteristics. The study revealed a lack of quality management: fewer than 15% of AI-generated and fewer than 20% of human-created commit and pull request messages mention software quality characteristics. Our analysis showed that security is mentioned significantly less by AI-agents (3%) compared to humans (7%). The most used format for structuring commit and pull request messages is Conventional Commits[2]. Conventional Commits consists of tags for documenting changes to features (`feat:`), tests (`test:`), or documentation (`doc:`), some of them can be mapped to the quality characteristics of functional completeness, reliability and maintainability. But the Conventional Commits tags cover only part of software quality. ISO/IEC 25010 [5] provides an exhaustive list of software quality characteristics, but is barely referenced in code changes, which hinders quality tracing. Information about software quality changes is crucial for providing information and communication technologies (ICT) [5]. Non-obvious changes to the performance efficiency may have a critical influence on the operation of a system. In cases of high load or denial of service attacks, this directly influences the availability of the system. Switching to a new cryptographic algorithm may improve security, but may decrease the system performance efficiency. The software quality engineering processes call for explicit management of software quality [5], which is not present in the examined commit messages. This leads to the following research questions (RQs).

RQ1 How can a format for commit messages that includes quality characteristics improve **human capabilities** in software quality management?

RQ2 How can a format for commit messages that includes quality characteristics improve **automation** in software quality management?

We propose Quality Commits, an extension of Conventional Commits that makes software quality explicit in modern collaborative software engineering. Quality Commits adds improved and decreased software quality characteristics, based on ISO/IEC 25010, to the format. This enables the human-in-the-loop to triage software problems, review and test code changes with greater focus, provide mitigations for decreased quality. This helps automated approaches to create more specific patch notes, support quality prioritization, and allows the creation of softgoal interdependency graphs.

The Conventional Commits format is:
`<type>[optional scope]: <description>`
Our Quality Commits extension is:
`<type>[optional scope, '+'quality, '-'quality]: <description>`
Quality Commits adds improved (denoted by +) and decreased quality characteristics (-) to the scope.

As an **example** we compare the message in the **Quality Commits** format:
`feat(login, +security, -performance efficiency): Switched from MD5 to SHA256`
to the **Conventional Commits** format: `feat(login): Switched from MD5 to SHA256`

2. Background and Related Work

Software Quality: ISO/IEC 25010 [5] provides a quality model and consists of established top-level quality characteristics (shown in Figure 2): functional suitability, performance efficiency, compatibility, interaction capability, reliability, security, maintainability, flexibility, and safety. These characteristics

[1] GitHub: https://github.com/
[2] Conventional Commits: https://www.conventionalcommits.org/

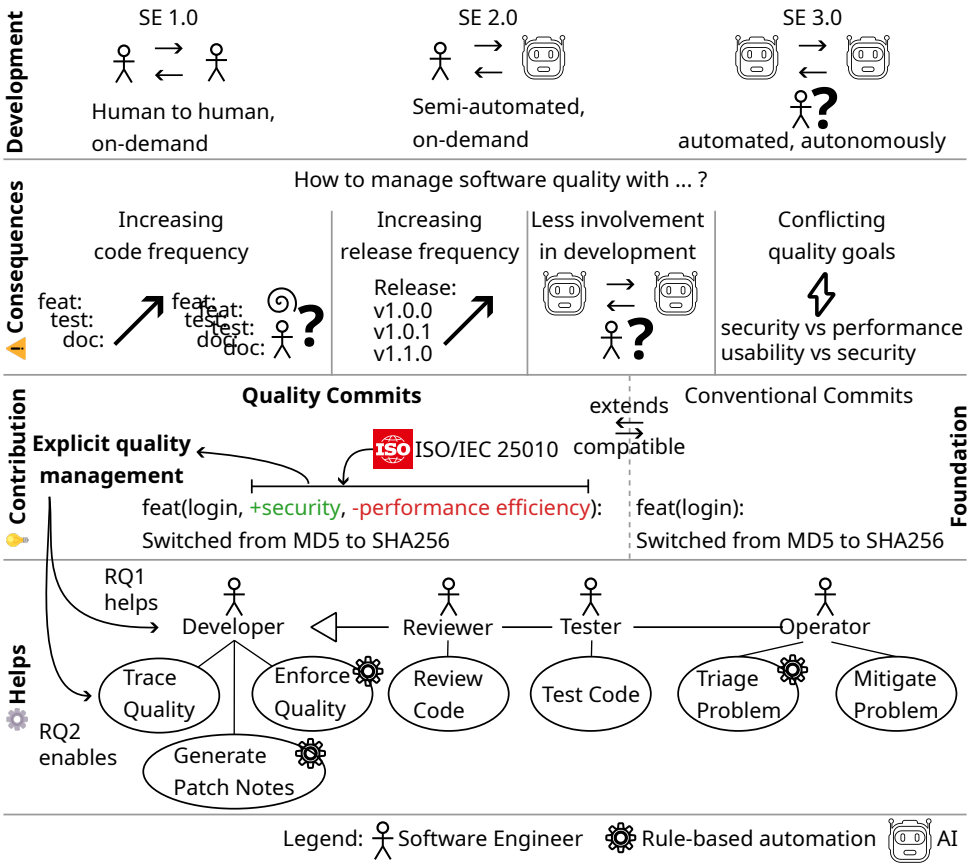

Figure 1: Context, problem, contribution and use of Quality Commits

can have sub-characteristics, for example, security has the sub-characteristics confidentiality and integrity among others. ISO/IEC 25010 is our vocabulary source for the quality characteristics of Quality Commits. **Version Control:** Modern software collaboration uses version control systems like Git[3],

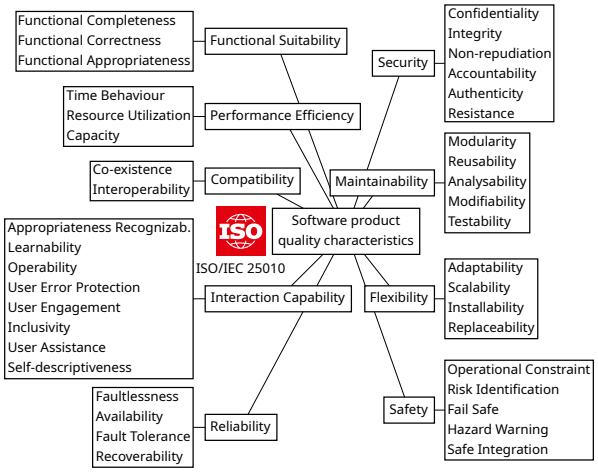

Figure 2: ISO/IEC 25010 quality model with its characteristics and sub-characteristics

Figure 3: Gitmoji

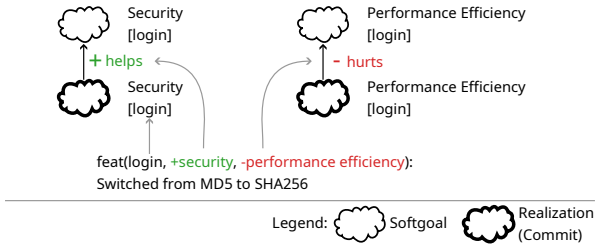

Figure 4: Softgoal interdependency graph (SIG)

where a commit serves as an atomic code change. A commit has a commit message. **Conventional Commits** is a specification for adding human- and machine-readable information to commit messages. The main elements of Conventional Commits are the tags `feat:`, `fix:`, `build:`, `chore:`, `ci:`, `docs:`,

[3]Git: https://git-scm.com/

`style:`, `refactor:`, `perf:`, and `test:` that are added to commit messages to indicate its contents. In addition, a scope can be given, such as `feat(login):` meaning there is a feature changed in the login of the software. **Formats similar to Conventional Commits:** Gitmoji[4] is a guide that aims to standardize the meaning and use of emojis for GitHub. The Gitmoji syntax is `<intention> [scope?][:?] <message>`, where the intention is expressed by an emoji. Figure 3 shows an excerpt of emojis with their intention. SECOM[5] [6] investigates the need for additional security information in commit messages, such as fixed vulnerabilities. **Technical implementations:** Commitlint[6] is a tool to check if the format of a commit message adheres to standards. Softgoal interdependency graphs (SIGs) show interdependencies of softgoals in non-functional requirements (NFRs) [7] as shown in Figure 4. **Decreased quality, side effects and conflicts:** Quality characteristics can be in conflict. Often, code changes have side effects. In our example, we switch a security algorithm with a new algorithm that is more secure but slower, stated more formally, it decreases the performance efficiency. **One-dimensionality and positive/negative problem:** Conventional Commits and other approaches only allow one tag to be used, e.g. `feat:` or `perf:` in Conventional Commits. Our approach decouples the intent, adding a new feature from its impacts, the performance issue. While SECOM focuses solely on security, Quality Commits generalizes this to all ISO/IEC 25010 quality characteristics. Because Quality Commits allows users to add multiple changed quality characteristics and allows users to classify the effect as a positive or negative influence on the mentioned quality characteristic. We solve the one-dimensionality and positive/negative problem of Conventional Commits and Gitmoji, where developers can only tag one characteristic and not its influence, like positive or negative impact.

3. Methodology

To address the software quality management gap in commit messages, we followed the seven **Design Science Research (DSR)** guidelines by Hevner [8]. **1) Design as an artifact:** We developed Quality Commits, a viable artifact in the form of a construct (the Quality Commits format) and an instantiation (Git hooks using the format) that solves the problem of missing quality management in commit messages. **2) Problem relevance:** As detailed in section 1 and Figure 1, the transition from SE 1.0 to SE 3.0 has increased code frequency and limited oversight by humans-in-the-loop. This is a problem for quality management. Our artifacts address this critical business problem by providing Quality Commits to enable software quality management in commit messages. **3) Design evaluation:** We demonstrated the utility of Quality Commits with illustrative scenarios (section 4). We evaluated how Quality Commits supports both human tasks (RQ1) and automated tasks (RQ2). **4) Research contributions:** With Quality Commits we provide a lightweight, backward-compatible extension to Conventional Commits between high-level software quality management (ISO/IEC 25010) and low-level implementation (commit messages). **5) Research rigor:** Quality Commits is grounded in established foundations: the accepted quality management standard ISO/IEC 25010 and the widely adopted Conventional Commits practice to structure commit messages. **6) Design as a search process:** The design process involved a search for a solution to introduce quality management in commit messages which is machine-readable, human-readable, backward-compatible with existing practices, and does not introduce new terms to software engineers. **7) Communication of research:** We present Quality Commits to technology-oriented and management-oriented audiences. For technology-oriented audiences, such as developers, reviewers, testers, and operators, we detail how the format helps to focus quality during implementation, testing and review with detailed scenarios and a prototypical implementation. For management-oriented audiences, such as software architects, we detail how Quality Commits helps high-level software quality management such as quality prioritization and enforcement.

[4]Gitmoji: https://gitmoji.dev/
[5]SECOM: https://github.com/security-commits/secom
[6]Commitlint: https://commitlint.js.org/concepts/commit-conventions.html

4. Utility Demonstration

We demonstrate the feasibility and utility of Quality Commits through illustrative scenarios [8] based on common software engineering tasks. We provide context for each scenario in the form of a use case. For each use case, we identify the target user group and instantiate the task with a concrete scenario.
Use Case 1) Patch note creation (developer, operator, rule-based automation): This facilitates generating patch notes that include improved and decreased quality characteristics. Scenario: As a developer, I want to prepare a release with patch notes that accurately inform operators about software quality changes. Solution: The developer now has the improved and decreased quality characteristics for each commit that is part of the new release and can generate patch notes that link the quality changes for each changed feature, or have an extra section in the patch notes that summarizes the quality changes. This information is important for operators, so that they can triage or mitigate problems with focus on quality (Use Case 2 and 3).
Use Case 2) Triage problems (operator, rule-based assistance): This helps triage software problems, since patches with the marked decreased characteristics could be focused to examine if they caused the problem. Scenario: As an operator, I want to know why the login of the application is slow since the last patch. Solution: The operator refers to the patch notes improved with quality changes (Use Case 1) between the last operated version and the new patch to identify changes mentioning decreased performance efficiency in the login component. This is important for the operator to quickly detect which patch might have caused the problem. Filtering patch notes by quality can be automated.
Use Case 3) Mitigation (operator): Information about improved and decreased characteristics, like in patch notes (Use Case 1), helps software operators to prepare for, e.g., higher load and allocate more resources. Scenario: As an operator, I want to plan resources so that the application I operate is compliant to its KPIs, such as latency. Solution: The operator consults the patch notes enhanced by Quality Commits and quickly sees that performance is reduced. The operator uses the information to test different resource allocations (such as faster CPU or more RAM) until the KPI for latency is reached.
Use Case 4) Review (developer, tester): This facilitates manual code reviews, since the reviewer could verify the claim of the improved characteristics or can check if the tradeoffs could be mitigated. Scenario: As a reviewer, I want to check the claim that SHA256 is more secure than MD5. Solution: The reviewer can focus on evaluating the security of SHA256, e.g. by referring to *NIST Special Publication 800-107 Recommendation for Applications Using Approved Hash Algorithms* [9] to evaluate that the decreased performance efficiency can be mitigated by faster, more secure algorithms.
Use Case 5) Testing (tester): Quality changes can be tested with greater focus. Scenario: As a software tester, I want to ensure the quality of the code I have to test. Solution: The tester can test the changed quality characteristics with greater focus, in our example, the tester can create a load test or capacity test, which are common metrics for performance efficiency.
Use Case 6) Trace quality (developer, tester, reviewer, operator, architect, rule-based automation): Scenario: As a software engineer, I want to be able to trace quality through the project. Solution: The software engineer can trace quality characteristics throughout code changes, test, patch notes, and operation with Quality Commits. This helps confirm that quality requirements are met. Figure 4 shows a softgoal interdependency graph (SIG). The SIG can be created from the information in the Quality Commits format. This gives an overview of softgoals, their relations, realizations, and conflicts.
Use Case 7) Prioritize quality (software architect, rule-based automation):
Scenario: As a software architect, I want to be able to prioritize quality goals. The architect wants to prioritize security over performance efficiency (`security > performance efficiency`). Solution: The architect can prioritize software quality goals as simple rules (Listing 1 a)). Then a commit with the message `feat(login, +performance efficiency, -security)` could be automatically rejected with a Git commit message hook. Git hooks are scripts that are automatically executed on events such as new commits. Listing 1 shows the feasibility of automation with a Git commit message hook which a) allows for defining quality priorities, b) demonstrates how to parse the Quality Commits format, c) checks for the existence of improved qualities, and d) checks for compliance to quality priorities.

Listing 1: Python code of a Git message hook for prioritizing and enforcing quality in commit messages

```python
# a) Prioritize quality characteristics (left is higher priority than right)
PRIORITY_LIST = [
    ["security", "performance efficiency"],
    ["functional correctness", "performance efficiency"]
]

# b) Extract improved and decreased quality characteristics from the message
improved  = set(re.findall(r"\+\s*([\w\s-]+)", commit_message))
decreased = set(re.findall(r"-\s*([\w\s-]+)", commit_message))

# c) Enforce improved quality characteristics are present
if not improved:
    print("Policy Violation: No improved characteristics stated.")
    sys.exit(1)

# d) Enforce high-priority quality is not sacrificed for low-priority item
for priority in PRIORITY_LIST:
    for high_index, high in enumerate(priority):
        if high in decreased: # Check if high priority item is decreased
            for low in priority[high_index+1:]:
                if low in improved: # Check if low priority item is improved
                    print(f"Policy Violation: '{high}' > '{low}'")
                    sys.exit(1)
```

5. Future Work

Linking quality characteristics to common metrics, best practices, and mitigations: For some quality characteristics, commonly used **metrics** exist, e.g., for performance efficiency it is execution time, memory usage, or requests per second, for analysability it is cyclomatic complexity, and for availability it is uptime. There are accepted **best practices** that help reach the quality characteristics. E.g., for security it is STRIDE, DREAD, UMLsec [10], and NIST Cybersecurity Framework (CSF) 2.0 [11]. Linking commonly used metrics and best practices to quality characteristics helps software engineers to make quality measurable and apply best practices. **User Evaluation:** We demonstrated the utility for its users by providing illustrative scenarios and implemented tool prototypes to demonstrate the utility for automation. We propose to empirically evaluate how the usability of Quality Commits is perceived and accepted by its users. [12] provides a technology acceptance model (TAM) that is used to rate the acceptance of a technology by the user. Ergonomics of human-system interaction ISO 9241-11:2018 Part 11: Usability: Definitions and concepts [13] defines human-system interaction and provides methods we plan to use to evaluate our work. We plan to use TAM and ISO 9241-11 to evaluate the usability and the acceptance of Quality Commits. **AI Applications:** Quality Commits could help improve AI coding agents by creating labeled input data that can be used for training. Quality Commits could be used as a (domain-specific) language used in AI prompts. For example, `feat(login, +performance efficiency)` could signal the user intent to improve the performance of the login.

6. Conclusion

We proposed Quality Commits, a lightweight extension to Conventional Commits that embeds ISO/IEC 25010 quality characteristics into commit messages. We demonstrated its utility for human tasks (RQ1), such as problem triage and mitigation planning, quality tracing, software testing and review, and automation tasks (RQ2), like patch notes generation, quality enforcement and prioritization. Quality Commits is especially helpful in collaboration with autonomous AI-agents where developers are confronted with frequent repository scale code changes. Quality Commits closes the gap between quality requirements and documenting and tracing quality changes through the implementation in daily modern collaborative software engineering workflows using version-controlled code repositories.

Acknowledgments

The work is partially funded by Deutsche Forschungsgemeinschaft (DFG) - project No. 500462081 in context of the project *TraceSEC - Tracing and Explaining Security in Software Engineering* and by the German Federal Ministry of Education and Research in the context of the project *PrivacyE2E*.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] H. Li, H. Zhang, A. E. Hassan, The Rise of AI Teammates in Software Engineering (SE) 3.0: How Autonomous Coding Agents Are Reshaping Software Engineering, arXiv preprint arXiv:2507.15003 (2025).

[2] K. Großer, V. Riediger, J. Jürjens, Requirements document relations: A reuse perspective on traceability through standards, Software and Systems Modeling (2022) 1–37. doi:10.1007/s10270-021-00958-y.

[3] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, A. Bernstein, The missing links: bugs and bug-fix commits, in: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10, Association for Computing Machinery, New York, NY, USA, 2010, p. 97–106. doi:10.1145/1882291.1882308.

[4] S. Amershi, D. Weld, M. Vorvoreanu, A. Fourney, B. Nushi, P. Collisson, J. Suh, S. Iqbal, P. N. Bennett, K. Inkpen, J. Teevan, R. Kikin-Gil, E. Horvitz, Guidelines for Human-AI Interaction, in: Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, CHI '19, Association for Computing Machinery, New York, NY, USA, 2019, p. 1–13. URL: https://doi.org/10.1145/3290605.3300233. doi:10.1145/3290605.3300233.

[5] International Organization for Standardization (ISO), ISO/IEC 25010:2023(en) Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model, ISO Standard, ISO, 2023.

[6] S. Reis, R. Abreu, H. Erdogmus, C. S. Pasareanu, SECOM: Towards a convention for security commit messages, in: 19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022, ACM, 2022, pp. 764–765. doi:10.1145/3524842.3528513.

[7] L. Chung, B. A. Nixon, E. Yu, J. Mylopoulos, Non-Functional Requirements in Software Engineering, Springer US, Boston, MA, 2000. doi:10.1007/978-1-4615-5269-7.

[8] A. R. Hevner, S. T. March, J. Park, S. Ram, Design Science in Information Systems Research, MIS Quarterly 28 (2004) 75–105.

[9] Q. H. Dang, Recommendation for applications using approved hash algorithms, 2012. doi:10.6028/nist.sp.800-107r1.

[10] J. Jürjens, UMLsec: Extending UML for Secure Systems Development, in: J.-M. Jézéquel, H. Hussmann, S. Cook (Eds.), ≪UML≫ 2002 — The Unified Modeling Language, 2002, pp. 412–425.

[11] National Institute of Standards and Technology (NIST), The NIST Cybersecurity Framework (CSF) 2.0, Technical Report, NIST, 2024. doi:10.6028/nist.cswp.29.

[12] F. D. Davis, Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology, MIS Quarterly 13 (1989) 319–340. URL: http://www.jstor.org/stable/249008.

[13] International Organization for Standardization (ISO), ISO 9241-11:2018 Ergonomics of human-system interaction Part 11: Usability: Definitions and concepts, ISO Standard, ISO, 2018.