Politecnico di Torino

POLITECNICO DI TORINO

Master's degree in computer engineering

A.Y. 2025 / 2026

Graduation Session March / April 2026

Empirical analysis of defects in R and R Markdown

Supervisors:

Marco Torchiano

Candidate:

Soheil Jamshidi

ABSTRACT

R Markdown supports literate programming workflows where narrative text, executable code, and rendering configuration are tightly coupled. While these workflows improve transparency and reproducibility, they also introduce defect modes that extend beyond conventional source-code bugs, including failures in document execution and rendering pipelines. This thesis presents an empirical repository-mining study of defects in R Markdown ecosystems based on bug-fix commits from GitHub.

We analyzed 57 repositories and extracted 13,473 bug-fix commits. Each commit was automatically classified into a ten-category defect taxonomy using an R-aware classifier that combines diff-level evidence with artifact/path signals. To ensure reliability of quantitative conclusions, we applied repository-level quality control (QC) based on coverage and confidence metrics. After QC filtering, 53 repositories (11,139 commits) were retained for statistical analysis. QC-passing repositories achieved high classification coverage (mean 98.24%, median 99.8%) and low low-confidence rates (mean 2.45%, median 0.63%), with 0% NA classifications and 0% suspect relabels.

Results show that defect activity is concentrated in a small subset of categories. In the typical repository, Documentation / Formatting (mean 35.05%) and Implementation / Logic (mean 33.21%) account for the largest shares of bug fixes, followed by Rendering / Conversion (mean 18.98%). Artifact-touch analysis further shows that defect categories differ in where fixes are applied: rendering-related fixes are usually applied in R Markdown documents (mean 70.45% of commits touch .Rmd), whereas File I/O and Export fixes are typically applied in R scripts (mean 81.48% touch .R). Overall, the findings emphasize that many defects are shaped by interactions between code, documents, and the rendering pipeline, motivating debugging practices and tooling that treat R Markdown documents as first-class software artifacts.

Contents

Figures

Tables

1

INTRODUCTION

1.1 Background: R, R Markdown

R is widely used for statistical computing and data analysis, and its ecosystem supports workflows where narrative explanation and executable analysis are produced together [3]. **R Markdown** enables this workflow by combining text, code chunks, figures, and metadata into a single source document that can be rendered into multiple output formats (e.g., HTML reports, PDF documents, slides, books, and websites) [4], [6]. This literate programming approach helps authors explain assumptions, preserve computational steps, and distribute results in a reusable form [1], [2].

However, the same integration that makes R Markdown attractive also increases complexity. Correct output depends not only on R code, but also on the interaction between code chunks, execution order, chunk options, document structure, rendering engines, and configuration files (often expressed through YAML metadata) [5], [6]. As a result, defects may manifest as conventional code bugs, but also as rendering failures, configuration issues, dependency conflicts, and reproducibility breakdowns that emerge only when the document is executed and rendered [6].

1.2 Motivation and problem statement

Despite the popularity of R Markdown-based workflows in research and industry, empirical evidence about the defect landscape of these projects remains limited. Many empirical studies of defects focus primarily on conventional software artifacts and treat the documentation as secondary. In contrast, R Markdown documents are executable artifacts that act as both documentation and a program, suggesting that defect patterns in these projects may differ from those found in conventional software systems [1], [6].

Understanding defect types and their typical locations is useful for multiple stakeholders. For practitioners, it can inform debugging strategies and development practices. For tool builders, it

can highlight opportunities for editor support, validation, and automated checks tailored to the document execution and rendering pipeline. For researchers, it provides a foundation for defect prediction models, automated repair techniques, or more refined empirical analyses of literate programming ecosystems [2].

The core problem addressed in this thesis is therefore: **what defect types are most common in R Markdown projects, and how do these defect types relate to the artifacts modified during bug fixes?**

To address this gap, this thesis conducts an empirical repository-mining study of bug-fix commits from GitHub repositories that contain R and R Markdown artifacts.

1.3 Research questions

This thesis addresses the problem through a repository-mining study of bug-fix commits in GitHub repositories that contain R and R Markdown artifacts. The study is guided by the following research questions:

- **RQ1.** Which defect categories are most prevalent in R Markdown projects?
- **RQ2.** How do defect categories differ in the artifacts they touch (e.g., .R vs .Rmd)?
- **RQ3.** How consistent are defect patterns across repositories after quality-control (QC) filtering?

1.4 Contributions

This thesis makes three main contributions:

1. **A quality-controlled dataset of bug-fix commits** mined from GitHub repositories containing R and R Markdown artifacts, with repository-level filtering based on predefined QC thresholds to support reliable quantitative analysis.
2. **A defect taxonomy tailored to R Markdown development**, consisting of ten categories: Rendering / Conversion; Dependency / Package; Environment / Configuration; Implementation / Logic; Data / Input Handling; Visualization / Plotting; Reproducibility / Versioning; File I/O and Export; Documentation / Formatting; Miscellaneous / Unknown.

3. **An artifact-aware empirical characterization of defects**, reporting both (i) category prevalence across repositories and (ii) category-specific touch patterns that distinguish code-centric fixes from document-centric fixes.

1.5 Thesis organization

The remainder of this thesis is structured as follows. Section 2 reviews related work on literate programming and reproducible computational workflows, the R Markdown ecosystem, defects in computational notebooks, and mining software repositories for bug-fix commits and defect taxonomies. Section 3 presents the methodology, including repository and commit collection, artifact identification, defect taxonomy, automated classification pipeline, and repository-level quality control. Section 4 reports the empirical results, covering QC outcomes, defect category prevalence, and artifact-touch patterns. Section 5 discusses the implications of the findings for practitioners, tool builders, and researchers. Section 6 outlines threats to validity. Section 7 concludes the thesis and suggests directions for future work.

2

RELATED WORK

This chapter positions the thesis with respect to prior work on literate programming and reproducible computational workflows, tooling in the R Markdown ecosystem, empirical studies of defects in computational notebooks, and established methods for mining bug-fix commits from software repositories.

2.1 Literate programming and reproducible computational workflows

The idea of combining executable code with narrative explanation is rooted in **literate programming**, which frames programs as human-readable documents with embedded computation [1]. In scientific computing, this paradigm is frequently motivated by **reproducible research**, where results should be regenerable from the underlying code and data rather than manually copied into reports [2]. R Markdown directly supports these goals by integrating prose, executable code, and rendered outputs into a single source that can be compiled into multiple formats, and by encouraging workflows where analysis outputs are generated from code at render time rather than by manual editing [6], [12].

2.2 The R Markdown ecosystem: tools and failure surface

R Markdown is typically implemented as a pipeline in which embedded code chunks are executed (commonly via **knitr**) and the resulting intermediate document is rendered into final formats (commonly via **Pandoc**) [6]. This toolchain allows flexible document generation (e.g., HTML/PDF/Word), but also introduces a broader failure surface than conventional scripting: correctness can depend on code, chunk execution order, chunk options, document structure, and metadata (e.g., YAML) in addition to dependencies and runtime configuration [6]. The R Markdown documentation and developer-authored guides emphasize the centrality of this pipeline and the practical importance of configuration and rendering steps when authoring dynamic documents [6].

This motivates an artifact-aware perspective in which defect types are not only counted but also linked to the artifacts they modify (e.g., code-centric .R files versus document-centric .Rmd files), which is a core part of the empirical analysis in Chapter 4.

2.3 Defects in computational notebooks (Jupyter / Python)

A closely related line of research investigates defects in **computational notebooks**, especially Jupyter. De Santana *et al.* report a large empirical study of bugs in Jupyter notebook projects using GitHub repository mining, Stack Overflow posts, and interviews with practitioners, and propose a notebook-specific taxonomy including categories such as Environment and Settings, Implementation, Conversion, and Kernel-related issues [11]. Their study highlights defect modes that are specific to notebook execution and environment behavior, and provides a useful reference point for interpreting defect patterns in other executable-document ecosystems such as R Markdown.

2.4 Mining software repositories and identifying bug-fix commits

This thesis builds on established methods in **mining software repositories (MSR)**, where bug-fix commits are commonly identified using commit-message heuristics and repository mining at scale. Prior work cautions that GitHub-mined datasets can introduce representativeness and data-quality risks, and that repository heterogeneity can affect empirical conclusions if not handled explicitly [7], [8]. In addition, bug-to-change linking and "bug-fix commit" identification may be biased or incomplete, because developers do not consistently reference bug reports or use uniform commit-message conventions [9]. Work on identifying bug-fixing patches and recovering links between bugs and changes further illustrates both the usefulness and limitations of heuristic approaches in large-scale defect studies [10], [13].

2.5 Defect taxonomies and artifact-aware perspectives

Defect taxonomies are widely used to summarize and reason about defect landscapes, but taxonomies must be adapted to the artifact types and workflows of a given ecosystem. Studies on notebooks emphasize categories tied to kernel/session behavior and environment dynamics [11], while R Markdown's pipeline and artifact structure motivate categories tied to rendering/conversion and document-centric configuration [6]. This thesis contributes to this line of work by proposing a taxonomy tailored to R Markdown development and by explicitly

analyzing **artifact-touch patterns** (e.g., .R vs .Rmd) to characterize how different defect types localize in code-centric versus document-centric artifacts.

Overall, prior work establishes the importance of executable documents for reproducible research and highlights that such workflows introduce failure modes that span code, configuration, and rendering pipelines. Empirical defect studies have provided strong evidence for notebook ecosystems (e.g., Jupyter), and MSR methods provide practical approaches for mining bug-fix commits at scale, but there is comparatively less empirical characterization of defects specific to the R Markdown artifact structure and toolchain. This thesis addresses this gap by proposing an R Markdown–aware defect taxonomy and a transparent, artifact- and diff-driven classification approach, and by quantifying both defect prevalence and artifact-touch patterns across repositories. The next chapter describes the dataset construction, taxonomy operationalization, classification pipeline, and QC procedure used in this study.

3

METHODOLOGY

This section describes the study design, data collection procedure, defect taxonomy, automated classification pipeline, and the quality-control process used to ensure that the quantitative analyses are based on reliable classifications.

Pipeline overview

The study follows an end-to-end mining and analysis pipeline. We first identify GitHub repositories containing R and R Markdown artifacts and extract candidate bug-fix commits using keyword-based heuristics on commit messages. We then retrieve the modified files and diffs for each candidate commit and classify the commit into one of ten defect categories using an automated, R-aware classifier that combines diff-level evidence with artifact/path signals. Next, we compute repository-level quality-control (QC) metrics—coverage, low-confidence rate, NA rate, and suspect relabel rate—and partition repositories into QC-passing and QC-failing sets according to predefined thresholds. All quantitative analyses are conducted on QC-passing repositories only: we compute per-repository category distributions (RQ1), category-specific artifact-touch patterns (RQ2), and dispersion-based indicators of cross-repository variability (RQ3). QC-failing repositories are retained for transparency and reserved for qualitative inspection without influencing the main quantitative conclusions.

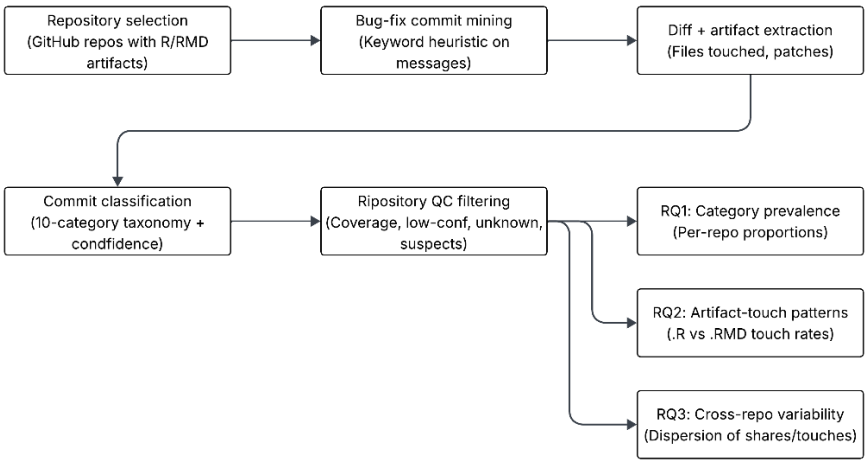

Figure 3.1 – Study pipeline: repository mining, commit classification, QC filtering, and analysis outputs (RQ1–RQ3).

3.1 Study design and unit of analysis

This thesis is a repository-mining study centered on **bug-fix commits**. The unit of analysis is a single commit identified as bug-fixing, together with its changed files and diffs. Each commit is assigned exactly one label from the defect taxonomy (Section 3.4) and is also characterized by the artifacts it touches (e.g., .R, .Rmd).

3.2 Repository set and commit collection

The final dataset consists of **57 GitHub repositories**. The repository set was constructed through a staged selection process. We first queried GitHub for candidate repositories containing R-related artifacts and then filtered the results to retain projects that include **R Markdown artifacts** (e.g., .Rmd) and are sufficiently active for repository-mining (to avoid abandoned or toy repositories). We further excluded repositories that were not suitable for analysis, such as forks or mirrors, and repositories where the retrieved history did not yield a minimum amount of bug-fix evidence under our commit-mining heuristic. After applying these inclusion and exclusion criteria, the final dataset comprised **57 repositories**. The complete list of repositories and the exact selection procedure (queries, filters, and scripts) are provided in **Appendix A** and in the accompanying replication package (see the repository link at the end of this section).

Across these repositories, we extracted **13,473** bug-fix commits. Candidate bug-fix commits were identified using a keyword-based heuristic on commit messages (e.g., “fix”, “bug”, “error”, “issue”). This approach is commonly used in large-scale mining studies because it scales across many repositories without requiring full linkage between commits and issue trackers; however, prior work has also shown that commit–issue linking can be incomplete and biased, motivating careful interpretation and validation practices [9]. To further ground the use of commit-message and patch-based signals for identifying bug-fix changes, we draw on established empirical work studying automatic identification of bug-fixing patches in large projects [10]. Finally, since mining GitHub repositories can introduce sampling and data-quality pitfalls (e.g., project heterogeneity, activity noise, and repository usage patterns), we follow common recommendations and explicitly account for these limitations in our design and validity discussion [7], [8].

To support transparency and reproducibility, we provide a replication package containing the mining scripts, intermediate outputs, and the full repository list at:

https://github.com/Arissj4/Empirical-analysis-of-defect-in-R-Markdown

3.3 Artifact identification (R and R Markdown)

For each bug-fix commit, we record which artifact types are touched. In particular, we distinguish between R scripts (.R) and R Markdown documents (.Rmd). In a small number of repositories, .qmd files may occur; when present, they are treated as the same executable-document artifact type for the purpose of touch statistics. This does not change the scope of the thesis, which focuses on defects in **R Markdown** projects and reports touch patterns primarily in terms of .R versus .Rmd.

3.4 Defect taxonomy

To characterize the defect landscape of R Markdown projects, we defined a defect taxonomy tailored to the development workflow of executable documents. The taxonomy was informed by prior work on computational notebooks, particularly the study by de Santana et al. [11], which motivates taxonomy design to capture ecosystem-specific failure modes and to make defect analyses actionable for tool improvement. Following this motivation, we adapted the idea of notebook-specific categories to the R Markdown setting, where failures are often mediated by the render pipeline, chunk options, document structure, and the interaction between documents and supporting R scripts [6], [11].

Category definitions were iteratively refined during pilot analysis to ensure that each category corresponds to a coherent class of bug-fix changes observable in diffs. Categories are intended to be mutually exclusive at the commit level; each bug-fix commit receives exactly one primary label based on the most direct cause of the defect as evidenced by the patch. Commits that do not provide sufficient evidence for a specific category, or that mix multiple unrelated changes without a clear dominant defect type, are assigned to Miscellaneous / Unknown to preserve transparency.

Table 3.1 summarizes the **ten categories** with operational definitions and an illustrative example for each category. The examples are representative patterns observed in the mined repositories (e.g., changes to YAML headers, chunk options, package dependencies, and R code).

Category	Operational definition (what the defect concerns)	Typical artifacts touched	Illustrative example of a fix
Rendering / Conversion	Defects in the document-to-output pipeline (e.g., knitr/Pandoc/LaTeX), output format selection, chunk execution order affecting rendering, or conversion failures.	.Rmd/.qmd, YAML header, render scripts	Fix a PDF render failure by correcting YAML output options (e.g., switching to a supported LaTeX engine) or adjusting chunk options that produce invalid markup.
Dependency / Package	Missing, incompatible, or incorrectly specified R packages and dependencies (installation, version constraints, namespace conflicts).	.R, DESCRIPTION, renv.lock	Add a missing package import (library()) and update dependency/version specification to resolve an error introduced by a package update.
Environment / Configuration	Execution environment, toolchain, or configuration issues not primarily about an individual package (paths, locale, CI settings, system requirements, R version).	.Rprofile, CI configs, YAML, scripts	Fix a failure in CI/render by setting required environment variables or configuring the runtime (e.g., installing a TeX distribution for PDF builds).
Implementation / Logic	Incorrect or incomplete program logic in R code (algorithmic mistakes, wrong conditionals, faulty transformations).	.R, sometimes .Rmd code chunks	Correct a conditional or calculation in an analysis function that produced wrong results (e.g., wrong join key or incorrect aggregation).
Data / Input Handling	Errors in reading, validating, transforming, or cleaning input data (schemas, missing values, parsing, file paths to inputs).	.R, .Rmd, data-loading scripts	Make data loading robust by using a stable project-root path and explicit parsing (e.g., here::here(...) and setting column types).

Visualization / Plotting	Defects in generating plots or visual outputs (ggplot settings, aesthetics, scales, device settings, figure dimensions).	.R, .Rmd, plotting utilities	Fix a plotting error by adjusting ggplot aesthetics or scale types (e.g., converting a variable to factor before mapping to color).
Reproducibility / Versioning	Defects affecting repeatability across machines/runs (randomness, pinned versions, lockfiles, session assumptions).	renv.lock, .R, .Rmd, scripts	Add set.seed() and snapshot/update a lockfile to ensure the analysis yields stable results across environments.
File I/O and Export	Defects in writing outputs and exported artifacts (paths, permissions, missing directories, serialization formats).	.R, .Rmd, output scripts	Fix export failures by creating output directories before writing files or by correcting write.csv()/saveRDS()/ggsave() parameters.
Documentation / Formatting	Issues in narrative text, markdown structure, citations, formatting directives, or chunk metadata that affect document readability or correctness.	.Rmd/.qmd, markdown text, chunk labels	Resolve a knitr error caused by duplicated chunk labels, or fix a markdown structure issue (e.g., unclosed code fences) that breaks rendering.
Miscellaneous / Unknown	Catch-all for commits that lack clear evidence for a single category or involve mixed unrelated changes.	Any	A broad refactor touching many files without a dominant defect signal, making classification uncertain.

Table 3.1 – Defect taxonomy definitions and illustrative examples.

The taxonomy is designed to capture both conventional code-centric defects and defect modes that are specific to executable documents, such as failures related to rendering pipelines and document configuration.

Terminology note (to avoid ambiguity later):

- **NA (QC outcome)** refers to commits for which the classifier cannot produce a usable label (i.e., the label is missing/not available).
- **Miscellaneous / Unknown (taxonomy label)** is a valid defect category used when a commit provides insufficient evidence to match the other nine categories.

In the final dataset, the QC NA rate is **0%**, meaning all commits received a usable label; the taxonomy label *Miscellaneous / Unknown* remains part of the taxonomy definition for completeness.

3.5 Automated classification pipeline

Commits are classified using an automated, R-aware classifier that prioritizes evidence specific to R / R Markdown ecosystems:

- **Artifact / path evidence (primary):** which artifact types and paths are modified (e.g., .R, .Rmd, configuration files).
- **Diff evidence (primary):** change patterns in the diff that indicate typical defect / fix behavior (e.g., rendering directives, chunk-related edits, dependency adjustments, plotting code changes).
- **Commit message evidence (secondary):** used mainly for candidate selection and only as weak supporting evidence during classification.

The classifier outputs both a category label and a confidence signal. Confidence is not used to remove individual commits; instead, it is aggregated at repository level and used as part of QC filtering.

3.6 Quality control (QC) and dataset partitioning

This study uses an automated classifier to assign each mined bug-fix commit to exactly one defect category from the ten-category taxonomy (Table 3.1), together with an associated confidence outcome. The classifier is deterministic and rule-based: it relies primarily on observable evidence from the changed artifacts (file paths and extensions) and diff-level cues, while commit messages are treated only as secondary, weak evidence. This design supports transparency and reproducibility and keeps the labelling aligned with what changes in the repository.

For each commit, the pipeline extracts (i) the set of changed files and derives artifact-touch flags (e.g., changes in .R scripts, .Rmd/.qmd documents, YAML/build configuration, and related resources), and (ii) the diff hunks to detect taxonomy-specific signals (e.g., edits to chunk options, YAML keys, rendering/build directives, dependency and reproducibility adjustments, plotting changes, or data I/O and export logic). These signals contribute to category-specific evidence scores. A commit is assigned to the category with the strongest evidence when that evidence

exceeds a minimum strength requirement and is sufficiently separated from competing categories; otherwise, the commit is marked as **low-confidence** or **NA** (as defined below). The classifier was constructed by translating the operational definitions of the taxonomy into an initial library of path- and diff-based rules and then iteratively refining these rules through manual audit of sampled commits across repositories, adjusting rules and thresholds to reduce systematic ambiguity and improve robustness to common project layout variation.

The full mining, classification, and QC workflow is implemented in Python as a reproducible batch pipeline. For each repository, the pipeline (i) extracts the bug-fix commit set, (ii) runs per-commit classification to produce a label and confidence outcome, and (iii) aggregates repository-level QC metrics from these per-commit outputs. Intermediate results are stored in machine-readable outputs (per-commit classification results and repository summaries), enabling QC computation to be rerun deterministically and supporting targeted auditing and sensitivity checks.

To ensure that quantitative results are based on repositories where the classifier behaves reliably, we apply QC thresholds defined a priori in the pipeline configuration and applied uniformly to every repository. The threshold values were selected based on pilot runs and manual audit of sampled commits, with the goal of retaining repositories where classification coverage is high and ambiguous outputs are limited, while filtering out repositories where the classifier's signals are unreliable. QC is computed at the repository level using four metrics:

- **Coverage:** proportion of bug-fix commits that receive a usable category label from the classifier (regardless of confidence level).
- **Low-confidence rate:** proportion of bug-fix commits that receive a usable label but are flagged as low-confidence due to weak or competing evidence (e.g., near-ties between categories).
- **NA rate:** proportion of bug-fix commits for which the classifier cannot produce a usable label (QC outcome = NA), typically because signals are absent or mutually contradictory. By construction, **Coverage + NA rate = 100%**; the low-confidence rate is a subset of the covered commits and therefore does not "complete" the remaining percentage.
- **Suspect relabel rate:** proportion of bug-fix commits flagged by automated audit heuristics as potentially misclassified. Audit heuristics are post-classification consistency checks (e.g., a commit labeled as document-centric while touching only code artifacts, or a label that conflicts with strong artifact-touch flags), used to identify cases that warrant inspection.

In this study, a repository is considered QC-passing only if it satisfies all predefined thresholds simultaneously (**coverage ≥ 85%**, **low-confidence rate ≤ 15%**, **NA rate ≤ 10%**, **suspect relabel rate ≤ 10%**; with an additional warning flag when the suspect relabel rate exceeds **2%**). Otherwise, the repository is marked QC-failing. QC is applied as follows: for each repository, all mined bug-fix commits are processed by the classifier and aggregated into the four QC metrics above; the resulting pass/fail decision produces the dataset partition used throughout the thesis. Additional repository-level QC breakdowns and examples of failure modes are reported in the Results chapter.

After QC, **53 repositories passed and 4 repositories failed**. This partitions the bug-fix commits as follows:

- **QC-passing repositories:** 11,139 commits
- **QC-failing repositories:** 2,334 commits
- **Total:** 13,473 commits

These totals refer to automatically mined and automatically classified bug-fix commits; manual inspection was performed only on targeted samples for rule refinement and validation, not on the full set of **13,473** commits.

QC metrics for the QC-passing repositories are strong: coverage is high (mean **98.24%,** median **99.8%**) and the low-confidence rate is low (mean **2.45%**, median **0.63%**). The **NA rate** is **0%** and the **suspect relabel rate** is **0%** across all QC-passing repositories. Note that the **low-confidence rate** represents a subset of the covered commits; it therefore does not need to sum with **coverage** and **NA** to **100%**. QC-failing repositories are retained for transparency and may be inspected qualitatively, but they are excluded from the quantitative aggregation reported in Section 4.

3.7 Data analysis procedures

To answer the research questions, analyses are performed on the QC-passing repositories only.

Category prevalence (RQ1). For each repository, we compute the proportion of its bug-fix commits assigned to each defect category. We then aggregate these per-repository proportions across repositories (reporting summary statistics such as mean and median). This approach reduces the influence of very large repositories and reflects the "typical repository" profile.

Artifact-touch patterns (RQ2). For each category, we compute the proportion of commits that touch at least one .R file and the proportion that touch at least one .Rmd file. These touch

rates are computed per repository and then summarized across repositories in which the category appears.

Cross-repository variability (RQ3). Consistency is assessed through dispersion in per-repository category shares and touch rates (e.g., wide ranges and large mean–median gaps). Large dispersion indicates that defect profiles depend strongly on repository characteristics and workflow.

4

RESULTS

This section reports the empirical results for the three research questions, based on the repositories that passed the predefined quality-control (QC) thresholds. We first summarize QC outcomes and the final analysis dataset. We then present defect category prevalence (RQ1), artifact-touch patterns (RQ2), and cross-repository variability after QC filtering (RQ3).

4.1 Quality-control outcomes and final dataset

The study analyzed **57 repositories** and **13,473** bug-fix commits. After applying repository-level QC thresholds, **53 repositories passed** and **4 repositories failed**. The QC-passing repositories contain **11,139 commits**, while QC-failing repositories contain **2,334 commits**.

QC-passing repositories show high classification reliability overall: mean coverage is **98.24%** (median **99.80%**) and the mean low-confidence rate is **2.45%** (median **0.63%**). The QC NA rate is **0%** and the suspect relabel rate is **0%** across repositories. In contrast, QC-failing repositories exhibit substantially lower mean coverage (**81.70%**) and higher mean low-confidence rate (**31.09%**), and all four failures are due to coverage below the 85% threshold and low-confidence above the 15% threshold.

Metric	QC-PASS (n=53)	QC-FAIL (n=4)
Repositories (count)	53	4
Bug-fix commits (count)	11,139	2,334
Coverage (mean, %)	98.24	81.70
Coverage (median, %)	99.80	83.06
Low-confidence (mean, %)	2.45	31.09
Low-confidence (median, %)	0.63	31.26
NA (mean, %)	0.00	0.00
Suspects (mean, %)	0.00	0.00

Table 4.1. Quality-control (QC) summary for QC-passing and QC-failing repositories. Percentages are computed over bug-fix commits.

Repository	CDCgov_MIRA_bug_commits_classified	rjournal_rjournal.github.io_bug_commits_classified	KDE_rkward_bug_commits_classified	chapter-three_next-drupal_bug_commits_classified
Commits	33	241	1728	332
Coverage (%)	75.76	81.33	84.78	84.94
Low-conf (%)	30.30	23.65	38.19	32.23
NA (%)	0.00	0.00	0.00	0.00
Suspects (%)	0.00	0.00	0.00	0.00
reasons	Coverage 75.8% < 85% LowConf 30.3% > 15%	Coverage 81.3% < 85% LowConf 23.7% > 15%	Coverage 84.8% < 85% LowConf 38.2% > 15%	Coverage 84.9% < 85% LowConf 32.2% > 15%

Table 4.2. QC-failing repositories and failure reasons.

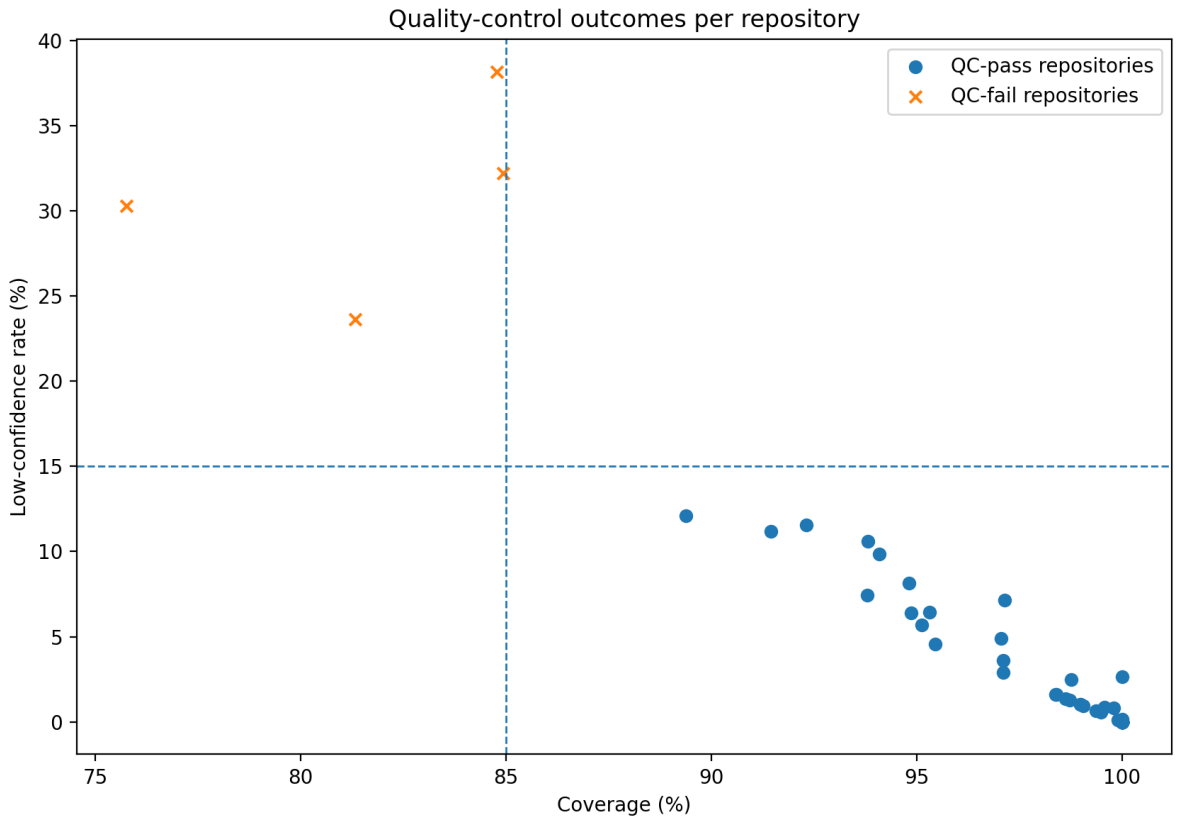

Figure 4.1 – Quality-control outcomes per repository (coverage vs low-confidence); dashed lines indicate QC thresholds.

4.1.1 Repository overview

Before presenting the RQ1 results, we summarize the repositories included in the quantitative analysis. The study started from 57 repositories mined from GitHub and identified 13,473 candidate bug-fix commits. After applying repository-level QC, 53 repositories (11,139 bug-fix commits) were retained as the QC-passing set used for quantitative aggregation, while 4 repositories (2,334 bug-fix commits) were marked QC-failing and excluded from the quantitative results.

To contextualize the prevalence findings, we also retain descriptive repository information captured during dataset construction. This includes basic project signals such as repository popularity and activity snapshots (e.g., stars and recent update time), as well as size proxies. In addition, the mined bug-fix commit data provide commit-level timestamps and file lists, which allow repository-level summaries such as the number of mined bug-fix commits per repository, the time span covered by the mined bug-fix commits (earliest to latest), and file-touch proxies derived from bug-fix commits (e.g., number of files changed per commit and number of unique files touched by mined bug-fix commits). These metadata are used only to describe and contextualize the dataset; they are not used by the classifier or as predictors in the RQ analyses, and platform-level counts should be interpreted as a snapshot taken during data collection.

4.2 Defect category prevalence in R Markdown projects (RQ1)

To characterize which defect categories are most prevalent, we computed per-repository category proportions and summarized them across QC-passing repositories. This repository-normalized view describes the "typical repository" rather than being dominated by a small number of very large projects.

Two categories dominate the typical repository: **Documentation / Formatting** (mean **35.05%**, median **34.3%**, present in **53/53** repositories) and **Implementation / Logic** (mean **33.21%**, median **33.3%**, present in **49/53** repositories). The third most prevalent category is **Rendering / Conversion** (mean **18.98%**, median **11.8%**, present in **47/53** repositories). Together, these results indicate that bug fixing in R Markdown projects commonly involves (i) changes to narrative / formatting and document structure, (ii) conventional logic-level code corrections, and (iii) issues specific to rendering and conversion.

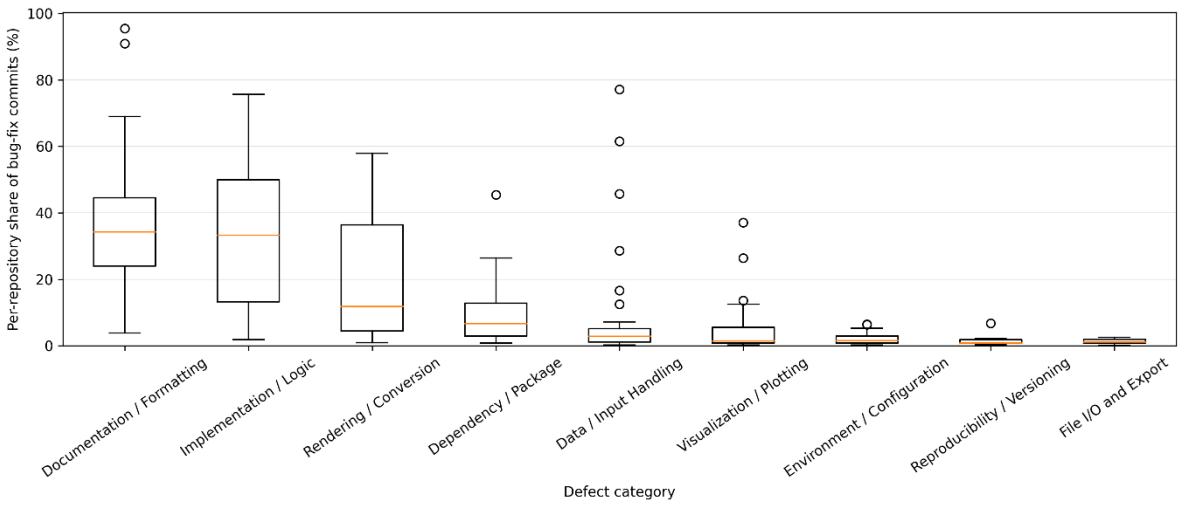

Figure 4.2 –Distribution of per-repository defect category shares in QC-passing repositories.

Figure 4.2 complements Table 4.3 by visualizing the dispersion of category prevalence across repositories, highlighting skewed long-tail behavior that can be obscured when focusing on mean prevalence values alone.

The remaining categories form a long tail. **Dependency / Package** (mean **9.29%**) and **Data / Input Handling** (mean **9.08%**) appear in many repositories but show large variability across projects. **Visualization / Plotting** is less prevalent on average (mean **6.13%**) and appears in fewer repositories (21/53). **Environment / Configuration**, **Reproducibility / Versioning**, and **File I/O and Export** are comparatively rare in the typical repository (means **2.24%**, **1.39%**, and **1.33%**, respectively), with File I/O and Export appearing in only **9/53** repositories.

Category	Mean (%)	Median (%)	Min (%)	Max (%)	Std (%)	Repos (count)
Documentation / Formatting	35.05	34.30	3.90	95.50	18.51	53
Implementation / Logic	33.21	33.30	1.90	75.60	21.05	49
Rendering / Conversion	18.98	11.80	0.90	57.90	17.66	47
Dependency / Package	9.29	6.70	0.80	45.40	8.64	41
Data / Input Handling	9.08	2.90	0.20	77.10	17.56	35
Visualization / Plotting	6.13	1.50	0.20	37.10	9.61	21
Environment / Configuration	2.24	1.60	0.20	6.50	1.78	26
Reproducibility / Versioning	1.39	0.80	0.30	6.80	1.49	18
File I/O and Export	1.33	1.20	0.10	2.60	0.89	9

Table 4.3. Defect category prevalence across QC-passing repositories (repository-normalized). Percentages refer to the share of bug-fix commits in a repository.

4.3 Artifact-touch patterns by defect category (RQ2)

To understand how defect categories differ in the artifacts they touch, we computed (per repository) the fraction of commits in each category that touch at least one **.R** file and at least one **.Rmd** file, and then summarized those touch rates across repositories.

The results show strong category-specific localization. **Rendering / Conversion** fixes are predominantly document-centric: the mean **.Rmd touch rate is 70.45%** (median **84.6%**), while the mean **.R touch rate is 15.95%** (median **2.8%**). In contrast, **File I/O and Export** fixes are predominantly code-centric: the mean **.R touch rate is 81.48%** (median **100%**) and the mean **.Rmd touch rate is 7.41%**.

Other categories exhibit mixed behavior. **Implementation / Logic** fixes tend to be code-centric (mean **.R touch rate 56.63%**, median **86.9%**) but still frequently involve document edits (mean **.Rmd touch rate 35.17%**), reflecting that many fixes occur within executable documents or require coordinated changes. **Documentation / Formatting** shows substantial touches to both

artifact types (mean **.R touch rate 37.17%**, mean **.Rmd touch rate 43.27%**), consistent with repositories where documentation, examples, and analysis narratives co-evolve with code.

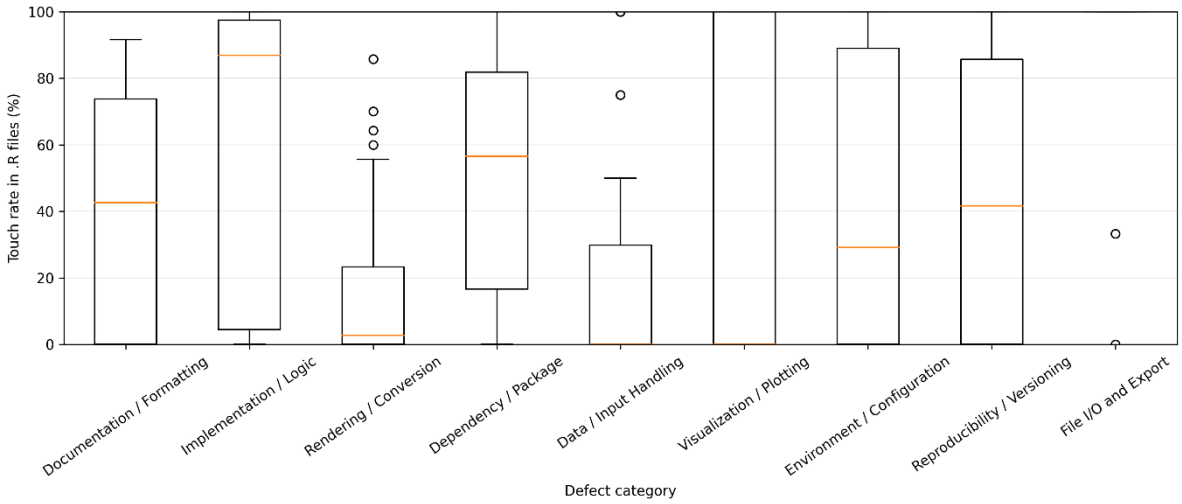

Figure 4.3 – Distribution of touch rates in .R files by defect category (QC-passing repositories). Touch rates are computed per repository as the fraction of commits in a category that touch at least one .R file. Boxes indicate the interquartile range (Q1–Q3) with the median; whiskers show the non-outlier range (1.5×IQR), and points indicate outliers.

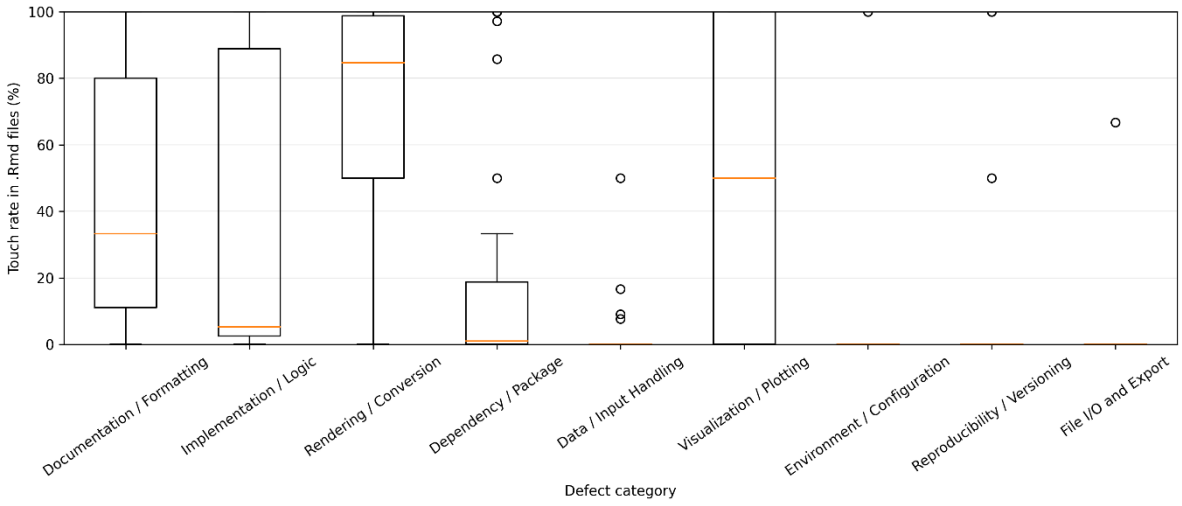

Figure 4.4 – Distribution of touch rates in .Rmd files by defect category (QC-passing repositories). Touch rates are computed per repository as the fraction of commits in a category that touch at least one .Rmd files. Boxes indicate the interquartile range (Q1–Q3) with the median; whiskers show the non-outlier range (1.5×IQR), and points indicate outliers.

Overall, these touch patterns highlight that R Markdown defects are not purely "code bugs" or purely "document issues": many categories involve cross-artifact interaction, motivating debugging practices and tool support that treat executable documents as first-class artifacts.

Figures 4.3 and 4.4 complement Tables 4.4 and 4.5 by visualizing the repository-to-repository dispersion of touch rates, including skew and outliers that summary statistics alone may obscure.

Category	Repos (count)	Touch .R (mean, %)	Touch .R (median, %)	Touch .R (min-max, %)
Data / Input Handling	35	22.05	0.00	0.0–100.0
Dependency / Package	41	51.21	56.50	0.0–100.0
Documentation / Formatting	53	37.17	42.60	0.0–91.6
Environment / Configuration	26	42.07	29.15	0.0–100.0
File I/O and Export	9	81.48	100.00	0.0–100.0
Implementation / Logic	49	56.63	86.90	0.0–100.0
Rendering / Conversion	47	15.95	2.80	0.0–85.7
Reproducibility / Versioning	18	43.67	41.65	0.0–100.0
Visualization / Plotting	21	34.13	0.00	0.0–100.0

Table 4.4. Touch rates in .R files by defect category (QC-passing repositories). Touch rates are computed per repository; a commit may touch multiple artifact types.

Category	Repos (count)	Touch .Rmd (mean, %)	Touch .Rmd (median, %)	Touch .Rmd (min-max, %)
Data / Input Handling	35	2.39	0.00	0.0–50.0
Dependency / Package	41	17.45	1.00	0.0–100.0
Documentation / Formatting	53	43.27	33.30	0.0–100.0
Environment / Configuration	26	7.69	0.00	0.0–100.0
File I/O and Export	9	7.41	0.00	0.0–66.7
Implementation / Logic	49	35.17	5.30	0.0–100.0
Rendering / Conversion	47	70.45	84.60	0.0–100.0
Reproducibility / Versioning	18	13.89	0.00	0.0–100.0
Visualization / Plotting	21	47.50	50.00	0.0–100.0

Table 4.5. Touch rates in .Rmd documents by defect category (QC-passing repositories). Touch rates are computed per repository; a commit may touch multiple artifact types.

4.4 Cross-repository consistency after QC filtering (RQ3)

To assess how consistent defect patterns are across repositories, we examined dispersion in per-repository category shares and touch rates. Several categories show substantial variability. For example, **Documentation / Formatting** ranges from **3.9%** to **95.5%** across repositories,

while **Implementation / Logic** ranges from **1.9%** to **75.6%**. **Data / Input Handling** shows particularly high dispersion, reaching up to **77.1%** in some repositories despite a low median (**2.9%**), suggesting that a minority of repositories are strongly data-handling dominated.

Artifact-touch patterns also vary widely across repositories within the same category (e.g., many categories have touch-rate ranges spanning 0–100%). This indicates that while broad trends are observable (e.g., Rendering / Conversion being document-centric), repository workflow and structure strongly influence the precise manifestation of defect categories and where fixes occur.

4.4.1 Repository characteristics and variability (exploratory)

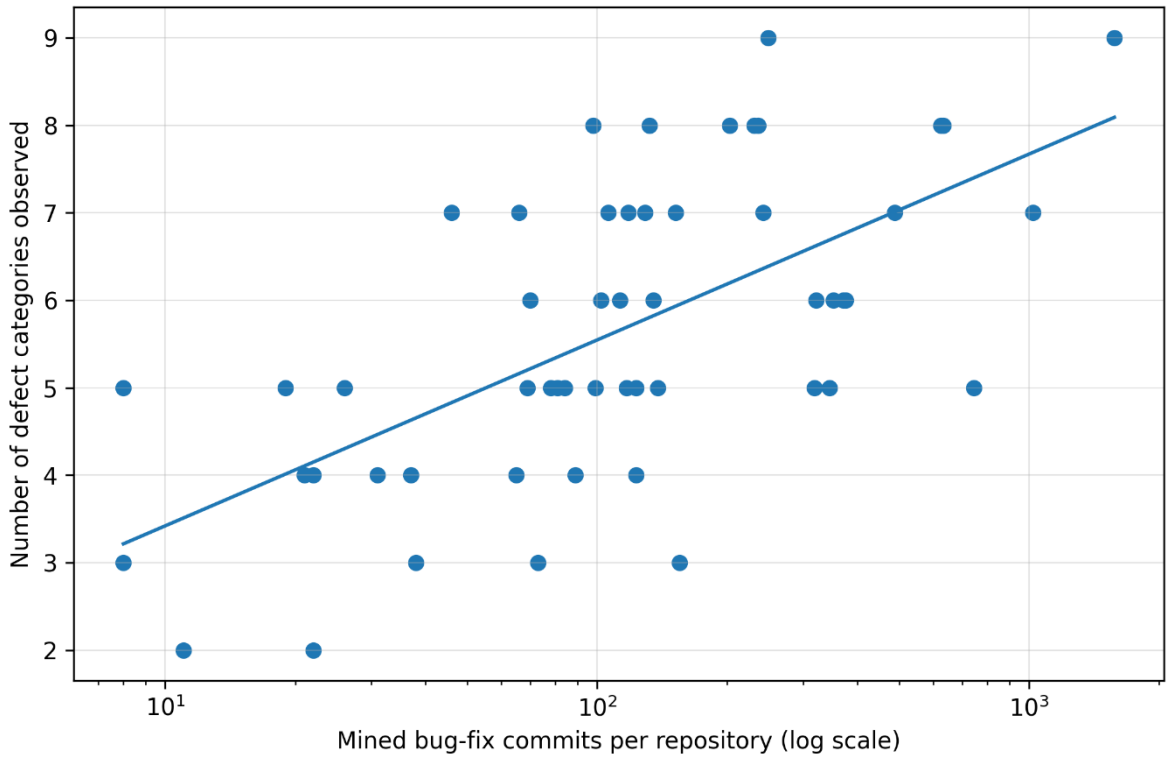

Figure 4.5 - Repository activity versus breadth of observed defect categories (QC-passing repositories). The x-axis shows the number of mined bug-fix commits per repository (log scale), and the y-axis shows the number of defect categories with non-zero prevalence in that repository. Spearman $\rho = 0.61$, $p < 0.001$.

The repositories in this dataset are not all the same size or type, so it is reasonable to ask whether this affects how much the defect patterns vary across repositories. To explore this question, we use simple repository-level descriptors collected during dataset construction. As a size/activity proxy, we use the number of mined bug-fix commits per repository (and a repository size proxy when available). As a rough indicator of application domain, we use repository topic

tags when they are present. These checks are descriptive: they are meant to provide context, not to change the main results reported for the research questions.

We find that larger, more active repositories tend to show a wider range of defect categories. Concretely, repositories with more mined bug-fix commits usually have more categories with non-zero prevalence. This relationship is strong and statistically significant. A practical interpretation is that small repositories may appear "narrow" simply because fewer bug-fix commits are available, while larger repositories reveal more types of defects over time.

At the same time, a larger repository does not automatically mean that defect categories are more evenly distributed. In many repositories, a few categories still dominate while the remaining categories form a long tail. Finally, domain comparisons based on topic tags should be interpreted cautiously because topic tags are missing or inconsistent for some repositories; we therefore use them only as qualitative context rather than as a basis for strong conclusions.

4.5 QC-failing repositories (qualitative summary and role in the thesis)

The four QC-failing repositories are excluded from the quantitative aggregation above to preserve reliability. However, they are retained for transparency and were manually inspected to understand why the automated classification produced low coverage and/or high low-confidence rates. This qualitative inspection is used (i) to contextualize QC filtering decisions in this section and (ii) to strengthen the **Threats to Validity** discussion by illustrating the boundary conditions under which the classifier is less reliable.

Manual inspection suggests that QC failures are primarily associated with repositories where bug-fix commits provide weak or ambiguous signals for the classifier. Common patterns include (i) large commits that mix multiple concerns and artifact types, reducing category specificity; (ii) atypical project layouts or non-standard file organization that weakens path-based cues; and/or (iii) recurring edits to generated, vendor, or configuration-heavy content where diff patterns are less indicative of a single defect category. These characteristics reduce classification coverage and increase low-confidence assignments, motivating the use of repository-level QC filtering for the quantitative analyses reported in Sections 4.2–4.4.

5

DISCUSSION

This chapter interprets the empirical results and discusses their implications for R Markdown practitioners, tool builders, and researchers. The discussion also situates the findings relative to prior work on computational notebooks, focusing on the Jupyter / Python bug taxonomy reported by de Santana *et al.* [11]. Finally, we clarify how the QC filtering and the qualitative analysis of QC-failing repositories affect the strength and scope of the conclusions.

5.1 What the defect landscape suggests about R Markdown development

Across analyzed repositories, defect fixing in R Markdown projects is dominated by **Documentation / Formatting** and **Implementation / Logic**, with **Rendering / Conversion** forming a substantial third cluster. A straightforward interpretation is that R Markdown projects behave like "hybrid systems": they contain conventional program logic and data processing, but also a document layer that actively shapes execution and output. In literate programming terms, the executable document is not merely a comment wrapper around code; it is a primary artifact whose structure, metadata, and rendering pipeline contribute directly to correctness [1], [6].

At first glance, the prominence of Documentation / Formatting might look surprising if one assumes "documentation bugs" are low impact. However, in R Markdown, formatting and document structure are part of the executable artifact: chunk options, headings, references, figure placement, and output configuration all influence what gets executed, how results are embedded, and whether the document renders successfully [6]. In this sense, a portion of what is labeled as Documentation / Formatting reflects *operational correctness* of the final rendered artifact rather than purely cosmetic edits. This observation reinforces the core motivation of this thesis: executable documents should be treated as first-class software artifacts rather than secondary by-products.

5.2 Cross-artifact coupling: why "where the fix happens" matters

The artifact-touch results show that defect categories are not only distinguishable by *what* they concern, but also by *where* they tend to manifest. The most pronounced example is **Rendering / Conversion**, which is strongly document-centric: fixes primarily touch .Rmd artifacts. This aligns with the fact that rendering failures and conversion issues are often mediated by document-level configuration (e.g., YAML metadata) and chunk-level settings rather than by changes in standalone scripts [6].

Conversely, **File I/O and Export** is strongly code-centric, with fixes primarily touching .R. This is consistent with typical I/O logic being implemented in scripts and functions rather than in prose-driven narrative. Between these extremes, several categories show mixed touch patterns. In particular, **Implementation / Logic** frequently spans both .R and .Rmd, suggesting that fixes often involve coordinated changes across scripts and executable documents (e.g., updating a function in .R while updating the analysis narrative or call sites in .Rmd). This is a practical signal for debugging: when a defect is suspected to be logic-related, it is often insufficient to inspect only the .R scripts or only the .Rmd documents—cross-artifact interactions are common.

From a workflow perspective, these touch patterns motivate "render-aware debugging": diagnosing failures in executable documents should incorporate document structure, chunk options, and render configuration in addition to conventional code inspection [6]. More broadly, the results suggest that build/render pipelines for literate artifacts deserve many of the same engineering practices typically reserved for code-only systems (e.g., automated checks, validation, and structured logging) [2], [6].

5.3 Consistency and variation across repositories

After QC filtering, several categories still show large variability across repositories. This has two important implications.

First, it cautions against treating a single global distribution as a universal "defect profile" for R Markdown projects. Repository characteristics—such as whether a project is document-heavy (reports/books) versus package-heavy (functions + vignettes), or whether it relies heavily on external data and plotting—can shift the defect mix substantially. Second, the observed dispersion suggests that defect prediction or automated repair approaches for R Markdown will likely need to incorporate repository context (e.g., project type, artifact composition, and build pipeline) rather than assuming a uniform defect distribution.

Methodologically, the repository-normalized reporting (per-repository proportions summarized across repositories) is intended to represent the "typical repository" rather than allowing a small number of large repositories to dominate results. This design choice follows common practices in empirical mining when project sizes are highly skewed [7], [8].

5.4 Comparison with defects in Jupyter / Python notebook projects

A useful lens for interpreting the R Markdown results is to compare them with findings from computational notebook ecosystems. de Santana et al. conducted a large empirical study of bugs in Jupyter notebook projects, combining evidence from GitHub repository mining, Stack Overflow posts, and interviews with data scientists [11]. Their taxonomy includes eight notebook-specific bug categories: Kernel, Conversion, Environment and Settings, Connection, Processing, Cell Defect, Portability, and Implementation [11]. Importantly, they report that the most frequent bug categories are **Environments and Settings** and **Implementation** [11].

Two contrasts stand out when placed next to the R Markdown results:

1. **Environment and configuration dominates in Jupyter; it is comparatively rare in R Markdown GitHub fixes.** In the Jupyter study, environment and settings issues are the most frequent bug type (in both Stack Overflow and GitHub data), and their root causes often include configuration and version problems [11]. In the R Markdown dataset, **Environment / Configuration** appears as a comparatively small share in the typical repository. One plausible interpretation is that Jupyter's execution model (kernels, interactive sessions, and dependency isolation) makes environment drift and configuration failures especially visible to practitioners, and therefore frequently reported and discussed [11]. In contrast, R Markdown workflows may surface more issues at render time and in document structure, which can be captured in categories like Rendering / Conversion and Documentation / Formatting [6]. At the same time, this comparison must be interpreted carefully because the data sources differ: the Jupyter study includes Stack Overflow posts and interviews (which amplify practitioner pain points), whereas this thesis focuses on GitHub bug-fix commits and reports repository-normalized distributions [11].

2. **Document-centric categories are more prominent in R Markdown.** R Markdown's core artifact is an executable document that explicitly interleaves narrative and code [6]. This naturally increases the "surface area" for defects tied to document structure and presentation-integrated execution. Jupyter notebooks also interleave code

and narrative, but the Jupyter taxonomy explicitly includes kernel- and execution-environment-oriented categories (Kernel, Connection, Processing) that do not have direct analogs in R Markdown's render pipeline [11]. In other words, both ecosystems are "literate," but they fail differently: Jupyter failures are often mediated by kernel/session and environment dynamics, whereas R Markdown failures are frequently mediated by rendering / conversion and document structure [6], [11].

Overall, the cross-ecosystem comparison supports a broader point: literate programming systems are not just "code plus text." They introduce ecosystem-specific failure modes that deserve dedicated taxonomies, tooling, and debugging practices [1], [6], [11].

To make the comparison more explicit, Table 5.1 provides a best-effort, non-bijective mapping between the ten R Markdown defect categories used in this thesis and the eight Jupyter notebook bug categories reported by de Santana *et al.* [11]. Because the taxonomies and data sources differ, the mapping is intended for qualitative interpretation rather than one-to-one equivalence.

R Markdown category (this thesis)	**Closest Jupyter category [11]**	**Rationale / notes**	**Match**
Rendering / Conversion	Conversion (CV)	Both concern transforming notebooks/documents into other formats (e.g., HTML/PDF) and failures in conversion/render pipelines.	High
Environment / Configuration	Environments and Settings (ES)	Both include environment setup and configuration problems (toolchain, runtime settings).	High
Dependency / Package	Environments and Settings (ES)	Package installation/version/dependency conflicts are treated as part of environment/settings issues in the Jupyter taxonomy.	Partial
Reproducibility / Versioning	Portability (PB); Environments and Settings (ES)	Reproducibility issues often relate to portability across machines and version/environment drift.	Partial
Implementation / Logic	Implementation (IP)	Incorrect/buggy code logic and algorithmic implementation defects.	High
Data / Input Handling	Implementation (IP); Processing (PC)	Data parsing/cleaning/validation issues are often implemented in code; may overlap with notebook processing behaviors.	Partial
Visualization / Plotting	Implementation (IP)	Plotting code defects are typically categorized as implementation-level in the Jupyter taxonomy.	Partial

File I/O and Export	Implementation (IP); Conversion (CV)	Reading/writing/export logic is implementation; exporting artifacts can also overlap with conversion.	Partial
Documentation / Formatting	Cell Defect (CD)	Formatting/doc-structure issues are closest to cell rendering/markdown/output issues in notebooks, though not fully equivalent.	Partial
Miscellaneous / Unknown	—	Catch-all category; no direct correspondence.	—

Table 5.1 – Cross-study mapping between R Markdown defect categories and Jupyter/Python notebook bug types

Jupyter categories with limited/no direct analog in the R Markdown taxonomy [11]:

• Kernel (KN): Notebook kernel lifecycle/crashes; no direct R Markdown equivalent (render typically runs a fresh R session).

• Connection (CN): Connectivity issues (e.g., remote kernels) are notebook-specific; not represented as a primary category in the R Markdown taxonomy.

• Processing (PC): Notebook execution / processing behaviors do not map cleanly; overlaps partially with implementation / data handling.

Figure 5.1 complements Table 5.1 by contrasting the overall category distributions reported in the two studies, highlighting both shared patterns and ecosystem-specific differences.

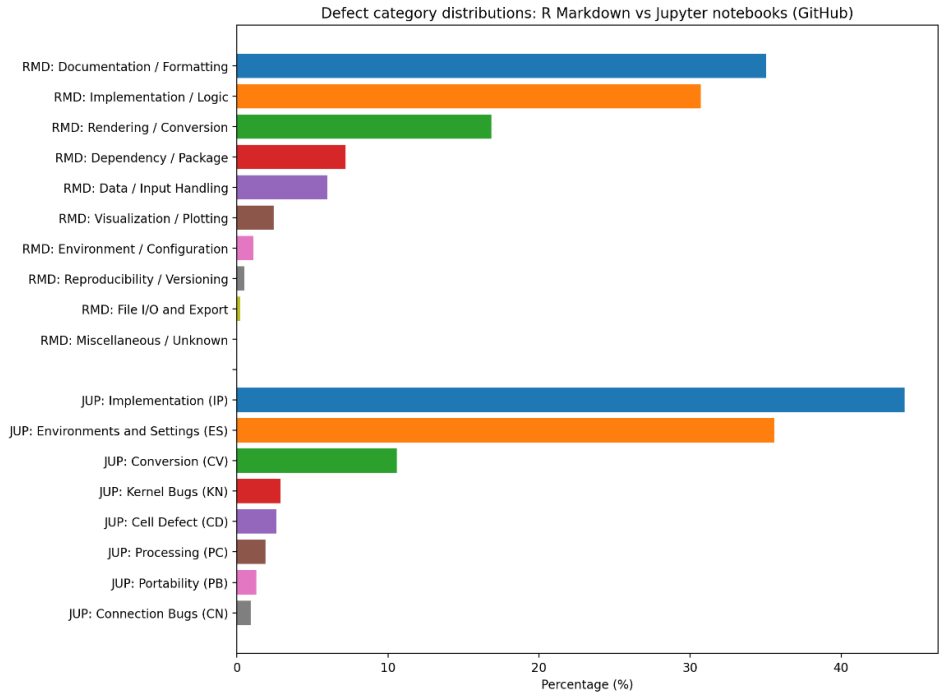

Figure 5.1 – Defect category distributions in R Markdown projects (this thesis) and Jupyter notebooks (GitHub) [11].

Limitations of the cross-study comparison:

While the comparison with Jupyter notebook projects provides a useful lens, it has important limitations. The two studies differ in taxonomy design, data sources, and aggregation strategies: **de Santana et al.** combine GitHub evidence with Stack Overflow posts and practitioner interviews, whereas this thesis analyzes GitHub bug-fix commits and reports repository-normalized distributions [11]. As a result, differences in category frequencies may reflect not only ecosystem-specific defect modes but also methodological choices and reporting biases (e.g., the types of problems that practitioners discuss online versus those that are fixed and committed in repositories). Therefore, the comparison should be interpreted as qualitative and indicative rather than as a direct quantitative equivalence.

5.5 Implications for practitioners

The results suggest several practical takeaways for developers working in R Markdown projects:

- **Treat rendering as part of debugging, not as a final step.** Rendering failures and conversion defects are document-centric and often arise from YAML configuration and chunk-level settings [6]. Debugging should routinely include checking document metadata, chunk options, and render logs—not only R code.

- **Expect cross-artifact fixes.** Several categories show mixed touches across .R and .Rmd, indicating that fixing a defect often requires coordinated updates across scripts and executable documents. Practically, this argues for reviewing bug fixes in a "project-wide" lens rather than isolating attention to only one artifact type.

- **Stabilize execution to support reproducibility.** Even when reproducibility / versioning defects are not the most prevalent, they can be disproportionately costly when they occur. Practices such as explicit session information capture, consistent package management, and stable data access patterns are aligned with reproducible research goals [2], [6].

5.6 Implications for tool builders

The artifact-localization patterns point toward concrete opportunities for tooling:

- **Render-aware diagnostics for executable documents.** Since many failures are document-centric, IDEs and CI pipelines can add pre-render validation (YAML sanity

checks, chunk option validation, detection of duplicated chunk labels, and warnings for non-deterministic execution patterns) [6].

- **Cross-artifact impact awareness.** Tools can surface when a code change in .R is likely to require synchronized updates in .Rmd (e.g., changed function signatures referenced in reports). Even lightweight "call-site awareness" could reduce breakage.

- **Better error localization.** Many rendering failures are experienced as "the render broke," but the root cause can be a specific chunk or option. Improving the mapping from render errors back to precise document regions would reduce debugging time [6].

These opportunities resemble the kinds of recommendations that emerge from Jupyter bug studies as well, where environment and configuration issues motivate better dependency and environment management support [11]. The difference is that R Markdown tooling must focus more on the render pipeline and document semantics, while Jupyter tooling often focuses more on kernel/session and environment robustness [11].

5.7 Implications for researchers

The results suggest several implications for empirical research on literate and executable-document ecosystems. First, the strong category-specific artifact-touch patterns indicate that defect analyses and prediction models should incorporate **artifact-aware features** (e.g., whether fixes touch .R, .Rmd, or both) rather than treating projects as code-only repositories.

Second, the observed cross-repository variability implies that defect distributions are not uniform across projects; future studies and models may benefit from incorporating repository context (e.g., artifact composition and project type) and reporting results in ways that avoid dominance by a small number of large repositories (as done here through repository-normalized aggregation).

Third, the comparison with Jupyter notebook defects highlights that literate ecosystems expose different failure surfaces (e.g., kernel/environment dynamics in notebooks versus render/document-structure issues in R Markdown), motivating further replication studies that apply consistent mining and QC procedures across ecosystems to separate methodological effects from genuine ecosystem differences. Finally, the taxonomy and the quality-controlled dataset provide a foundation for follow-up work such as multi-label classification of mixed-purpose fixes,

linking commits to issues/CI logs for stronger ground truth, and evaluating render-aware analyses or automated checks in real development workflows.

5.8 The role of QC filtering and QC-failing repositories

QC filtering is central to interpreting the quantitative findings. Mining-based studies can be distorted when automated classification is unreliable, or when repositories contain atypical patterns that weaken classifier signals [7], [8]. For this reason, the main quantitative results in Chapter 4 are computed only on repositories that pass predefined QC thresholds.

The **four QC-failing repositories** play an important complementary role. They provide qualitative evidence about *where the pipeline struggles* (e.g., conditions that reduce classification coverage or confidence), helping define the boundary of applicability for the automated approach. In this thesis, the QC-failing set is therefore used for **qualitative inspection and validity reasoning**, not for influencing the quantitative distributions. This separation strengthens the reliability of the reported prevalence and touch patterns, while still keeping the analysis transparent and informative.

6

THREATS TO VALIDITY

This chapter discusses limitations of the study design and analysis, following common validity dimensions for empirical software engineering studies. The main quantitative results are based on analyzed repositories only (Section 4.1), while QC-failing repositories are used qualitatively to characterize boundary conditions and inform validity arguments (Sections 4.5 and 5.8).

6.1 Construct validity

Construct validity concerns whether the operationalizations used in the study accurately measure the intended concepts.

Identifying bug-fix commits. Bug-fix commits were detected using a commit-message keyword heuristic (e.g., “fix”, “bug”, “error”). This approach is widely used in mining studies, but it can introduce both false positives (commits that use “fix” in a non-bug context) and false negatives (bug fixes not described with the chosen keywords) [9], [10]. As a result, the extracted commit set may not represent all bug fixes in each repository.

Defect taxonomy and single-label assignment. Each commit is assigned exactly one category from the ten-category taxonomy. In practice, some bug-fix commits address multiple issues (e.g., simultaneously adjusting dependencies and updating code logic), which may not be perfectly captured by a single label. This may lead to under-representation of mixed or multi-cause defect types and can increase ambiguity for large or broad commits.

Automated classification as a proxy for “defect type.” Defect categories are inferred from diffs and artifact/path signals rather than from ground-truth issue reports or developer intent. While diffs provide direct evidence of what changed, they may not fully reveal root cause. The QC process mitigates this risk by excluding repositories where classification signals are weak (low coverage and high low-confidence) from the quantitative analysis (Section 4.1), but the possibility of residual misclassification remains.

Touch analysis as a proxy for "defect localization." The artifact-touch analysis uses "file touched" to indicate where fixes occur. Touch does not necessarily imply that the defect originated in that artifact. For example, a rendering failure may be fixed by changing an .R script even if the symptom occurs during .Rmd rendering. Therefore, touch patterns should be interpreted as *fix-localization*, not definitive *fault-localization*.

6.2 Internal validity

Internal validity concerns whether the observed results could be explained by unintended factors in the analysis process rather than the phenomena being studied.

Classifier confidence and QC filtering. A key internal-validity risk is systematic misclassification. The study addresses this risk through repository-level QC thresholds based on coverage and confidence metrics and excludes QC-failing repositories from the quantitative aggregation (Section 4.1). This reduces the likelihood that overall results are driven by repositories where the classifier is unreliable. However, QC filtering can also introduce a form of selection: repositories that are harder to classify (e.g., due to atypical layouts or noisy commits) are removed from quantitative analysis, potentially shifting category proportions relative to the full set.

Qualitative role of QC-failing repositories. Manual inspection of the four QC-failing repositories is used to characterize boundary conditions of the pipeline rather than to influence category statistics. These repositories often include weak or ambiguous classification signals (e.g., large mixed-purpose commits or project structures that weaken path cues), which plausibly explains lower coverage and higher low-confidence rates. This qualitative evidence supports the internal validity of restricting quantitative analysis to analyzed repositories while preserving transparency about excluded cases.

Repository-normalized aggregation. Category prevalence is summarized using per-repository proportions (i.e., the "typical repository" view). This reduces dominance by very large projects but can also change the interpretation relative to a commit-weighted aggregation. In particular, categories that are common in many small repositories may appear more prominent than in a volume-weighted view. The choice is intentional, but readers should interpret results as repository-typical rather than ecosystem-total.

6.3 External validity

External validity concerns generalizability of the findings beyond the analyzed dataset.

Representativeness of GitHub repositories. Mining GitHub introduces known risks related to project heterogeneity, repository usage patterns, and selection bias [7], [8]. The dataset includes only repositories that meet selection criteria and are sufficiently active/available, and may under-represent private, industrial, or less-maintained projects.

Generalizing beyond analyzed repositories. Quantitative findings are strictly supported for analyzed repositories. Projects that resemble the QC-failing repositories (e.g., with atypical structures or low-signal diffs) may exhibit different defect distributions or may be harder to analyze automatically. The thesis therefore generalizes the main category prevalence and touch patterns to repositories where the classifier performs reliably under the defined QC thresholds.

Ecosystem comparison limits. The cross-ecosystem comparison with Jupyter notebook bugs (Section 5.4) is qualitative and indicative. De Santana et al. combine GitHub evidence with Stack Overflow and interviews, and their taxonomy differs from the one used in this thesis [11]. As a result, observed differences may reflect methodological and reporting differences in addition to ecosystem-specific defect modes.

6.4 Reliability and replicability

Reliability concerns whether the study would produce consistent results if repeated.

Determinism of the pipeline. The analysis pipeline is script-driven and uses predefined thresholds and taxonomy definitions. This supports repeatability when run on the same dataset snapshot. However, GitHub repositories evolve over time; rerunning repository mining at a later date may yield different commit sets and distributions due to new commits, rewritten histories, or repository changes [7], [8].

Heuristic parameters. Results can be sensitive to choices such as keyword lists for commit mining and thresholds for QC filtering. The study mitigates this risk by documenting thresholds and reporting QC outcomes explicitly (Section 4.1). Nonetheless, alternative reasonable parameterizations could produce slightly different datasets or distributions.

Measurement uncertainty. Although the QC process reduces classification uncertainty at repository level, residual uncertainty remains within analyzed repositories. To support

transparency, the thesis reports confidence-related QC statistics and separates QC-failing repositories from quantitative aggregation.

7

CONCLUSION

7.1 Conclusion

This thesis presented an empirical analysis of defects in R Markdown projects through a repository-mining study of bug-fix commits from GitHub repositories containing R and R Markdown artifacts. Using an automated, R-aware classification pipeline and repository-level quality control, we constructed a quality-controlled dataset and characterized defect types and their artifact-touch patterns.

Across the **analyzed repositories (53 repositories; 11,139 bug-fix commits)**, defect fixing in the "typical repository" is dominated by **Documentation / Formatting** and **Implementation / Logic**, with **Rendering / Conversion** forming a substantial third cluster. Artifact-touch analysis showed strong category-specific localization: **Rendering / Conversion** fixes are predominantly document-centric (high .Rmd touch rates), whereas **File I/O and Export** fixes are predominantly code-centric (high .R touch rates). Many other categories exhibit mixed touch patterns, highlighting that defects in R Markdown projects frequently involve cross-artifact interactions between scripts and executable documents.

Overall, the results support the view that R Markdown projects behave as hybrid systems in which the executable document is a first-class software artifact: correctness depends not only on code logic but also on document structure, rendering configuration, and the execution/render pipeline. These findings motivate development and debugging practices—and future tool support—that are explicitly render-aware and artifact-aware.

7.2 Future work

Several directions can extend this thesis:

1. **Multi-label and finer-grained classification.** Some bug-fix commits address multiple concerns (e.g., dependency changes plus logic fixes). Extending the classifier to support

multi-label assignment, or adding a second-stage "secondary label," could better represent mixed-cause fixes.

2. **Broader evidence sources.** This thesis focuses on GitHub bug-fix commits; incorporating issue reports, CI logs, or discussion platforms (e.g., Stack Overflow) could reveal defect types that are painful in practice but not frequently represented in commits, and would support stronger triangulation with prior notebook studies.

3. **Richer artifact coverage.** Future analyses could expand artifact-touch tracking to additional files that influence rendering and execution (e.g., YAML configuration, make/CI scripts, package metadata), and analyze higher-order touch patterns (e.g., which combinations of artifacts co-change).

4. **Tooling and evaluation.** The artifact-localization patterns suggest concrete tool opportunities (e.g., pre-render validation, chunk-option checking, improved error localization). Implementing and evaluating such tools in controlled studies or field deployments would translate empirical insights into measurable productivity improvements.

5. **Cross-ecosystem replication.** Extending the study design to other literate/executable document ecosystems (while maintaining consistent QC and reporting practices) would help clarify which defect modes are ecosystem-specific and which are common across literate programming systems.

Appendix A - Repository list and selection details

This appendix lists the 57 GitHub repositories included in the dataset and summarizes the repository selection procedure. Repositories were selected via GitHub search and filtered using predefined inclusion/exclusion criteria (presence of R Markdown artifacts, sufficient activity, and suitability for mining). Forks/mirrors and repositories failing basic eligibility checks were excluded. The complete replication package, including scripts, search queries, filters, and intermediate outputs, is available at:

https://github.com/Arissj4/Empirical-analysis-of-defect-in-R-Markdown

A.1 Included repositories (n = 57)

#	Repository (owner/name)	URL
1	ArgoCanada/argoFloats	https://github.com/ArgoCanada/argoFloats
2	Bioconductor/pkgrevdocs	https://github.com/Bioconductor/pkgrevdocs
3	CDCgov/MIRA	https://github.com/CDCgov/MIRA
4	Economic-and-Financial-Data-Discovery/imfdatapy	https://github.com/Economic-and-Financial-Data-Discovery/imfdatapy
5	IndrajeetPatil/ggstatsplot	https://github.com/IndrajeetPatil/ggstatsplot
6	KDE/rkward	https://github.com/KDE/rkward
7	NBISweden/workshop-mlbiostatistics	https://github.com/NBISweden/workshop-mlbiostatistics
8	PhanstielLab/plotgardener	https://github.com/PhanstielLab/plotgardener
9	Polkas/pacs	https://github.com/Polkas/pacs
10	RConsortium/S7	https://github.com/RConsortium/S7
11	SBOHVM/RPiR	https://github.com/SBOHVM/RPiR
12	SISBID/Data-Wrangling	https://github.com/SISBID/Data-Wrangling
13	SLCLADAL/SLCLADAL.github.io	https://github.com/SLCLADAL/SLCLADAL.github.io

14	Triangle-Modeling-and-Analytics/TRMG2	https://github.com/Triangle-Modeling-and-Analytics/TRMG2
15	XueyiDong/LongReadBenchmark	https://github.com/XueyiDong/LongReadBenchmark
16	ajrgodfrey/BrailleR	https://github.com/ajrgodfrey/BrailleR
17	andreamazzella/IntRo	https://github.com/andreamazzella/IntRo
18	andrewmarx/samc	https://github.com/andrewmarx/samc
19	bcgov/CCISS_ShinyApp	https://github.com/bcgov/CCISS_ShinyApp
20	biol607/biol607.github.io	https://github.com/biol607/biol607.github.io
21	chapter-three/next-drupal	https://github.com/chapter-three/next-drupal
22	compsocialscience/summer-institute	https://github.com/compsocialscience/summer-institute
23	cxli233/FriendsDontLetFriends	https://github.com/cxli233/FriendsDontLetFriends
24	daviddalpiaz/appliedstats	https://github.com/daviddalpiaz/appliedstats
25	dfo-mar-odis/shinySpatialApp	https://github.com/dfo-mar-odis/shinySpatialApp
26	easystats/effectsize	https://github.com/easystats/effectsize
27	easystats/parameters	https://github.com/easystats/parameters
28	ecmerkle/blavaan	https://github.com/ecmerkle/blavaan
29	emptyfield-ds/teaching_warehouse	https://github.com/emptyfield-ds/teaching_warehouse
30	gaynorr/AlphaSimR	https://github.com/gaynorr/AlphaSimR
31	hyunjimoon/SBC	https://github.com/hyunjimoon/SBC
32	info340/book	https://github.com/info340/book
33	leppott/ContDataQC	https://github.com/leppott/ContDataQC
34	neurogenomics/orthogene	https://github.com/neurogenomics/orthogene
35	nipraxis/textbook	https://github.com/nipraxis/textbook

36	oliviergimenez/banana-book	https://github.com/oliviergimenez/banana-book
37	opensaludlab/ciencia_datos	https://github.com/opensaludlab/ciencia_datos
38	r-lib/roxygen2	https://github.com/r-lib/roxygen2
39	r-tmap/tmap	https://github.com/r-tmap/tmap
40	rafalab/dsbook	https://github.com/rafalab/dsbook
41	rjournal/rjournal.github.io	https://github.com/rjournal/rjournal.github.io
42	ropensci/bold	https://github.com/ropensci/bold
43	rstudio/bslib	https://github.com/rstudio/bslib
44	rstudio/rticles	https://github.com/rstudio/rticles
45	rstudio/tensorflow.rstudio.com	https://github.com/rstudio/tensorflow.rstudio.com
46	satijalab/seurat	https://github.com/satijalab/seurat
47	stan-dev/cmdstanr	https://github.com/stan-dev/cmdstanr
48	strengejacke/sjPlot	https://github.com/strengejacke/sjPlot
49	stuart-lab/signac	https://github.com/stuart-lab/signac
50	thaum-xyz/ankhmorpork	https://github.com/thaum-xyz/ankhmorpork
51	tidyverse/datascience-box	https://github.com/tidyverse/datascience-box
52	tidyverse/readxl	https://github.com/tidyverse/readxl
53	tidyverse/tidyverse.org	https://github.com/tidyverse/tidyverse.org
54	tsahota/NMproject	https://github.com/tsahota/NMproject
55	wkumler/RaMS	https://github.com/wkumler/RaMS
56	yuryzablotski/yuzaR-Blog	https://github.com/yuryzablotski/yuzaR-Blog
57	z3tt/TidyTuesday	https://github.com/z3tt/TidyTuesday

REFERENCES

[1] D. E. Knuth, "Literate programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, 1984, doi: 10.1093/comjnl/27.2.97.
Available: https://doi.org/10.1093/comjnl/27.2.97

[2] R. D. Peng, "Reproducible research in computational science," *Science*, vol. 334, no. 6060, pp. 1226–1227, 2011, doi: 10.1126/science.1213847.
Available: https://doi.org/10.1126/science.1213847

[3] R Core Team, *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing, 2025.
Available: https://www.r-project.org/

[4] J. J. Allaire et al., "rmarkdown: Dynamic Documents for R," R package, CRAN, doi: 10.32614/CRAN.package.rmarkdown.
Available: https://cran.r-project.org/package=rmarkdown

[5] Y. Xie, "knitr: A General-Purpose Package for Dynamic Report Generation in R," R package, CRAN.
Available: https://cran.r-project.org/package=knitr

[6] Y. Xie, J. J. Allaire, and G. Grolemund, *R Markdown: The Definitive Guide*. Boca Raton, FL, USA: Chapman & Hall/CRC, 2018, doi: 10.1201/9781138359444.
Available: https://doi.org/10.1201/9781138359444
(Free online version) https://bookdown.org/yihui/rmarkdown/

[7] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. Germán, and D. E. Damian, "The promises and perils of mining GitHub," in *Proc. MSR*, 2014, pp. 92–101, doi: 10.1145/2597073.2597074.
Available: https://doi.org/10.1145/2597073.2597074

[8] V. Cosentino, J. L. Cánovas Izquierdo, and J. Cabot, "Findings from GitHub: Methods, datasets and limitations," in *Proc. MSR*, 2016, doi: 10.1145/2901739.2901776.
Available: https://doi.org/10.1145/2901739.2901776

[9] A. Bachmann, C. Bird, F. Rahman, P. T. Devanbu, and A. Bernstein, "The missing links: Bugs and bug-fix commits," in *Proc. FSE*, 2010, pp. 97–106, doi: 10.1145/1882291.1882308.
Available: https://doi.org/10.1145/1882291.1882308

[10] Y. Tian, J. Lawall, and D. Lo, "Identifying Linux bug fixing patches," in *Proc. ICSE*, 2012, doi: 10.1109/ICSE.2012.6227176.
Available: https://doi.org/10.1109/ICSE.2012.6227176

[11] T. L. de Santana, P. A. da M. Silveira Neto, E. S. de Almeida, and I. Ahmed, "Bug analysis in Jupyter notebook projects: An empirical study," *ACM Trans. Softw. Eng. Methodol.*, doi: 10.1145/3641539.
Available: https://doi.org/10.1145/3641539

[12] Y. Xie, *Dynamic Documents with R and knitr*, 2nd ed. Boca Raton, FL, USA: Chapman & Hall/CRC, 2017, doi: 10.1201/9781315382487.
Available: https://doi.org/10.1201/9781315382487

[13] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "ReLink: Recovering links between bugs and changes," in *Proc. ESEC/FSE*, 2011, doi: 10.1145/2025113.2025120.
Available: https://doi.org/10.1145/2025113.2025120