RETHINKING SOFTWARE SUPPLY CHAINS IN THE AGE OF GENERATIVE SYSTEMS: A CASE STUDY OF USE-CASE-ORIENTED REGENERATION

by

Tanmay Singla

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Master of Science in Electrical and Computer Engineering

Elmore Family School of Electrical and Computer Engineering

West Lafayette, Indiana

May 2026

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. James C. Davis

School of Electrical and Computer Engineering

Dr. Santiago Torres-Arias

School of Electrical and Computer Engineering

Dr. Aravind Machiry

School of Electrical and Computer Engineering

Approved by:

Dr. Milind Kulkarni

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor, Professor James C. Davis, for his mentorship, guidance, and support throughout my graduate studies. His thoughtful feedback, encouragement, and commitment to rigorous research have had a lasting impact on me as both a researcher and a person. I am thankful for the opportunity to learn from him and to pursue the work that led to this thesis.

I am also grateful to my collaborators, co-authors, and labmates for the many conversations, ideas, and feedback that shaped this work. I have been fortunate to work with people who are not only talented researchers, but also generous colleagues. Their support and collaboration made my graduate experience both intellectually exciting and personally meaningful.

I would like to thank the School of Electrical and Computer Engineering at Purdue University for providing a strong and supportive academic environment. I also appreciate the faculty, staff, and students who contributed to my growth during my time at Purdue.

Finally, I would like to thank my family and friends for their support, encouragement, and patience. This thesis would not have been possible without them.

TABLE OF CONTENTS

LIST OF TABLES 7

LIST OF FIGURES 8

GLOSSARY 9

ABSTRACT 11

1 INTRODUCTION 12
1.1 Context and Problem Statement 12
1.2 Thesis Statement 13
1.3 Contributions 13
1.3.1 Works that are part of this thesis 14
Works on which I am a lead author 14
Works on which I am a co-author 14

2 BACKGROUND 16
2.1 Software Supply Chains 16
2.2 Build-vs.-Buy in Software Sourcing 19
2.3 Trust and Its Limits in Software Supply Chains 20
2.4 Large Language Models and AI Coding Agents 22

3 AI CODING AGENTS AND DEPENDENCY DECISION-MAKING IN SOFTWARE REUSE 25
3.1 Introduction 25
3.2 Dependency Decision-making 26
3.3 Research Questions 27
3.4 Methodology 28
3.4.1 Dataset 29
3.4.2 Identifying Dependency Decisions 30
3.4.3 Measuring the Prevalence and Form of Dependency Changes 30
3.4.4 PR-Time Vulnerability Labeling 31
3.4.5 Comparison Against Human-Authored Pull Requests 32
3.4.6 Scope of the Analysis 33

3.5 Results . . . 33
3.5.1 RQ1: How often do AI coding agents modify dependencies? . . . 33
3.5.2 RQ2: What kinds of dependency changes do AI coding agents make, and how do those changes compare with human-authored pull requests? 34
3.5.3 RQ3: How often do AI coding agents select PR-time known-vulnerable dependency versions? . . . 35
3.6 Discussion . . . 37
3.7 Threats to Validity . . . 38

4 USE-CASE-ORIENTED REGENERATION AS A SOFTWARE SOURCING STRATEGY . . . 40
4.1 Introduction . . . 40
4.2 Background: Dependency Minimization and Internalization Strategies . . . 41
4.2.1 Debloating . . . 42
4.2.2 Tree Shaking and Dead-Code Elimination . . . 42
4.2.3 Vendoring . . . 43
4.2.4 Regeneration . . . 43
4.3 Research Questions . . . 45
4.4 Experimental Design . . . 46
4.4.1 Task Formulation . . . 47
4.4.2 Dependency Selection . . . 47
4.4.3 Repository Selection . . . 49
4.4.4 Baseline Preparation . . . 49
4.4.5 Agent, Prompting, and Regeneration Procedure . . . 50
4.5 Evaluation Methodology . . . 53
4.5.1 RQ1: Regeneration Feasibility . . . 55
4.5.2 RQ2: Behavioral Preservation . . . 56
4.5.3 RQ3: Differences from the Original Dependency . . . 56
4.5.4 Interpretive Scope . . . 57
4.6 Results . . . 57
4.6.1 RQ1: Regeneration Feasibility . . . 57
4.6.2 RQ2: Behavioral Preservation . . . 60
4.6.3 RQ3: Size and Dependency Footprint . . . 62
4.6.4 Validation Coverage and Behavioral Confidence . . . 66
4.7 Discussion and Limitations . . . 67

5 CONCLUSION 70
5.1 Summary of Findings 70
5.2 Future Work 71

REFERENCES 74

A EXTENDED REGENERATION RESULTS 80
A.1 Small Dependencies 80
A.1.1 `nanoid` 80
A.1.2 `change-case` 81
A.1.3 `chalk` 82
A.2 Medium Dependencies 83
A.2.1 `express` 83
A.2.2 `postcss` 84
A.2.3 `semver` 85
A.3 Large Dependencies 86
A.3.1 `axios` 86
A.3.2 `lodash` 87
A.3.3 `zod` 88
A.4 Failed Regeneration Outcomes 88

LIST OF TABLES

2.1 Core actors and components in a software supply chain. 18

3.1 Metrics used to evaluate the PR-time security impact of dependency decisions. . 32

3.2 PR-time vulnerability and remediation metrics for dependencies introduced via additions and updates. 36

4.1 Comparison of dependency reduction and replacement strategies 45

4.2 Target dependencies used in Chapter 4 48

4.3 Metrics used to evaluate use-case-oriented regeneration. 54

4.4 Representative regeneration outcomes for three dependencies each tested against a small, medium, and large repository. 58

4.5 Summary of regeneration results across all 180 repositories. 59

4.6 Focused dependency-specific test results 60

4.7 Average size and LOC reduction 63

4.8 Dependency structure and edit scope 64

4.9 API surface reduction 65

4.10 Average generated-code coverage across dependencies 66

A.1 Successful `nanoid` regeneration across repository–dependency pairs 80

A.2 Successful `change-case` regeneration across repository–dependency pairs 81

A.3 Successful `chalk` regeneration across repository–dependency pairs 82

A.4 Successful `express` regeneration across repository–dependency pairs 83

A.5 Successful `postcss` regeneration across repository–dependency pairs 84

A.6 Successful `semver` regeneration across repository–dependency pairs 85

A.7 Successful `axios` regeneration across repository–dependency pairs 86

A.8 Successful `lodash` regeneration across repository–dependency pairs 87

A.9 Successful `zod` regeneration across repository–dependency pairs 88

A.10 Failed regeneration outcomes and likely root causes. 89

LIST OF FIGURES

2.1 A simplified view of the software supply chain 17
2.2 Trust signals in the traditional software supply chain 21
3.1 Methodological overview for Chapter 3 29
3.2 Breakdown of dependency changes made by agents and humans. 34
4.1 Experimental workflow for use-case-oriented regeneration 46
4.2 Inputs and execution context for the regeneration agent 50

GLOSSARY

Software supply chain	The system through which software components are created, published, discovered, incorporated into downstream systems, and evolved over time. In this thesis, the software supply chain includes software artifacts, the contributors who produce them, the registries that distribute them, and the downstream developers and organizations that reuse them.
Package	A reusable software artifact that provides functionality to downstream consumers.
Dependency	A reusable package or component incorporated into a project to provide externally sourced functionality.
Contributor	An engineer or maintainer who creates, updates, and publishes software packages.
Registry	A platform that hosts packages and supports discovery, versioning, and distribution.
Reuser	A downstream developer or organization that incorporates external packages into its own project or system.
System	The resulting software system assembled from local code together with direct and transitive dependencies.
Direct dependency	A dependency explicitly declared by a project maintainer in a manifest file such as `package.json`, `requirements.txt`, or `pom.xml`.
Transitive dependency	A dependency introduced indirectly through another dependency rather than declared explicitly by the consuming project.

Repository–dependency pair	The combination of a downstream repository and a specific dependency used by that repository. In Chapter 4, this pair is the primary unit of analysis for evaluating use-case-oriented regeneration.
Dependency decision-making	The process of deciding whether to introduce, remove, or update a dependency, including which package or version to adopt in a project manifest.
AI coding agent	A repository-level generative system that can inspect codebases, modify files, invoke tools, and iteratively refine changes in order to complete software engineering tasks.
Use-case-oriented regeneration	The local regeneration of only the dependency functionality that a specific downstream repository actually uses, rather than recreating the dependency as a full general-purpose package.
Validation artifacts	The tests, builds, scripts, checks, and other repository-provided mechanisms used to evaluate whether a repository continues to behave as expected after a change.
First-party runtime reliance	Runtime imports or usages of a dependency from project-owned source code, excluding dependency use that is limited to third-party packages, development tooling, or external infrastructure.
Behavioral preservation	Preservation of repository behavior under the available validation artifacts. In this thesis, behavioral preservation does not imply full semantic equivalence with the original dependency on all possible inputs.
PR-time vulnerability	A vulnerability affecting a dependency version that was publicly known at the time the pull request introducing or updating that version was submitted.

ABSTRACT

Modern software systems are increasingly built through dependency-mediated reuse, making software supply chains central to how functionality is obtained, inherited, and maintained. At the same time, generative systems are beginning to change software sourcing in ways that traditional artifact-centric models do not fully capture. This thesis studies that shift in two related ways.

First, it examines how AI coding agents already influence software reuse within existing package ecosystems by making dependency decisions in real repositories. Using agent-authored and human-authored pull requests across 2,807 GitHub repositories spanning seven ecosystems, this work studies how often agents modify dependencies, what kinds of changes they make, and whether those decisions have measurable security consequences. The results show that agents routinely participate in dependency decision-making, perform version updates more frequently than humans, and more often select PR-time known-vulnerable dependency versions when introducing or updating dependencies.

Second, this thesis investigates whether generative systems may reshape software sourcing more fundamentally through use-case-oriented regeneration: the local regeneration of only the dependency functionality that a downstream repository actually uses. Rather than asking whether a package can be recreated in full, this work evaluates whether repository-used slices of dependency functionality can be regenerated as local implementations while preserving repository behavior under repository-level validation. Across 180 repository–dependency pairs, generated implementations preserved 99.8% of baseline validation checks, with 166 pairs showing perfect preservation.

Taken together, these findings show that generative systems are beginning to reshape software supply chains both within todays package ecosystems and beyond them: first by changing dependency decisions about which upstream artifacts projects adopt, and second by making repository-specific local regeneration a plausible sourcing strategy in some settings.

1. INTRODUCTION

1.1 Context and Problem Statement

Modern software development depends heavily on externally sourced components. Applications are commonly assembled from third-party packages obtained through software ecosystems such as npm, PyPI, Maven, and Cargo [1, 2]. As a result, software supply chains have become central to how functionality is obtained, inherited, and maintained in contemporary practice. This model enables rapid development and extensive reuse, but it also exposes downstream projects to inherited security, maintenance, and governance risks [3, 4].

At the same time, generative systems are beginning to change the structure of software sourcing. Large language models and agentic systems do not only assist with localized code generation. They increasingly operate at the level of repositories and engineering tasks, where they can modify code, tests, and dependency manifests as part of producing working solutions [5, 6]. In doing so, they interact directly with software supply chains, influencing how external functionality is selected, integrated, and maintained.

This thesis studies two related ways in which generative systems may reshape dependency-mediated software reuse. First, AI coding agents already participate in dependency decision-making within existing package ecosystems by adding, removing, and updating dependencies in real repositories. Second, increasingly capable generative workflows may support dependency replacement through use-case-oriented regeneration, where the functionality a repository uses from an upstream dependency is regenerated locally rather than obtained through continued reliance on the full external package.

Under this view, the relevant unit of analysis is not only the package in isolation, but also the repository–dependency pair. The central question is therefore not simply whether a project should build or reuse a full upstream artifact, but how generative systems affect both the decisions projects make about dependencies and the possibility of replacing repository-used dependency functionality with local generated implementations. This thesis studies both forms of change: generative systems as participants in dependency decision-making, and generative systems as mechanisms for repository-specific dependency replacement.

1.2 Thesis Statement

This thesis argues that generative systems can reshape software supply chains in two related ways. First, AI coding agents already influence software reuse within existing ecosystems by making dependency decisions about which external components to add, remove, and update. Second, large language models and agentic workflows may further reshape software sourcing by enabling *use-case-oriented regeneration*: the local regeneration of repository-used slices of dependency functionality in place of continued reliance on a full upstream package. Taken together, these developments suggest that software supply chains should be understood not only in terms of persistent upstream dependencies, but also in terms of software that may be generated, regenerated, validated, and maintained through generative workflows.

1.3 Contributions

This thesis makes two primary contributions.

First, Chapter 3 studies how AI coding agents participate in software reuse through the dependency decisions they make in real repositories. Using agent-authored and human-authored pull requests, it examines how often agents modify dependencies and evaluates the security consequences of those decisions, including the selection of PR-time known-vulnerable package versions [7]. These findings show that generative systems already influence software reuse within existing software ecosystems by changing which external components are introduced, updated, or retained.

Second, Chapter 4 investigates whether large language models and agentic workflows enable dependency replacement through use-case-oriented regeneration. Rather than asking whether widely used packages can be recreated in full, it studies whether a client repository can regenerate only the subset of dependency functionality it actually exercises as a local implementation. By evaluating regeneration feasibility, repository-observed behavioral preservation, and differences from the original dependencies in size and dependency footprint, the chapter examines when use-case-oriented regeneration may serve as a practical alternative to traditional dependency reuse.

Together, these contributions motivate a broader view of software supply chains in the age of generative systems. They suggest that generative systems affect software sourcing in at least two ways: first, by changing dependency decisions within existing ecosystems, and second, by enabling repository-specific dependency replacement in some settings.

1.3.1 Works that are part of this thesis

Works on which I am a lead author

The primary research contributions of this thesis are based on work for which I am the lead author.

- **Towards a Benchmark for Dependency Decision-Making** [7]. This work provides the foundation for Chapter 3, which examines how AI coding agents are changing software reuse through dependency decision-making in real repositories.
- **How Generative Systems May Reshape the Software Supply Chain** (in preparation). This work provides the foundation for Chapter 4, which explores whether large language models and agentic workflows can enable use-case-oriented regeneration and thereby reshape software sourcing practices.
- **An empirical study on using large language models to analyze software supply chain security failures** [8]. This work informs the broader background and motivation of this thesis, particularly with respect to software supply chains and their security implications, but it is not one of the thesis's central empirical chapters.

Works on which I am a co-author

This thesis also draws on broader collaborative work on which I am a co-author. These works primarily inform the framing, motivation, and background discussion of the thesis rather than serving as the basis of its central empirical chapters.

- **An industry interview study of software signing for supply chain security** [9]. KG Kalu, T Singla, C Okafor, S Torres-Arias, JC Davis.

- **AgentHub: A Research Agenda for Agent Sharing Infrastructure** [10]. E Pautsch, T Singla, W Jiang, H Peng, B Hassanshahi, K Läufer, ...
- **Learning From Software Failures: A Case Study at a National Space Research Center** [11]. D Anandayuvaraj, T Singla, Z Hammadeh, A Lund, A Holloway, JC Davis.
- **ZTD: Mitigating Software Supply Chain Vulnerabilities via Zero-Trust Dependencies** [12]. PC Amusuo, KA Robinson, T Singla, H Peng, A Machiry, S Torres-Arias, ...
- **Why Johnny Signs with Next-Generation Tools: A Usability Case Study of Sigstore** [13]. KG Kalu, S Okorafor, T Singla, S Chen, S Torres-Arias, JC Davis.

2. BACKGROUND

2.1 Software Supply Chains

In modern software engineering, substantial portions of application functionality are sourced from externally developed components distributed through package ecosystems such as npm, PyPI, Maven, and Cargo [1, 2]. These dependencies form part of what is commonly referred to as the software supply chain: the broader system through which software artifacts are produced, distributed, discovered, incorporated into downstream systems, and evolved over time.

In this thesis, the term *software supply chain* refers to the system through which software components are created, published, discovered, incorporated into downstream systems, and evolved over time. This includes not only the software artifacts themselves, but also the contributors who produce them, the registries that host and distribute them, and the downstream developers and organizations that reuse them. Figure 2.1 illustrates this view. Contributors publish packages into a software registry, packages evolve through continued development and through their own dependencies on other packages, and downstream reusers incorporate those packages into larger systems.

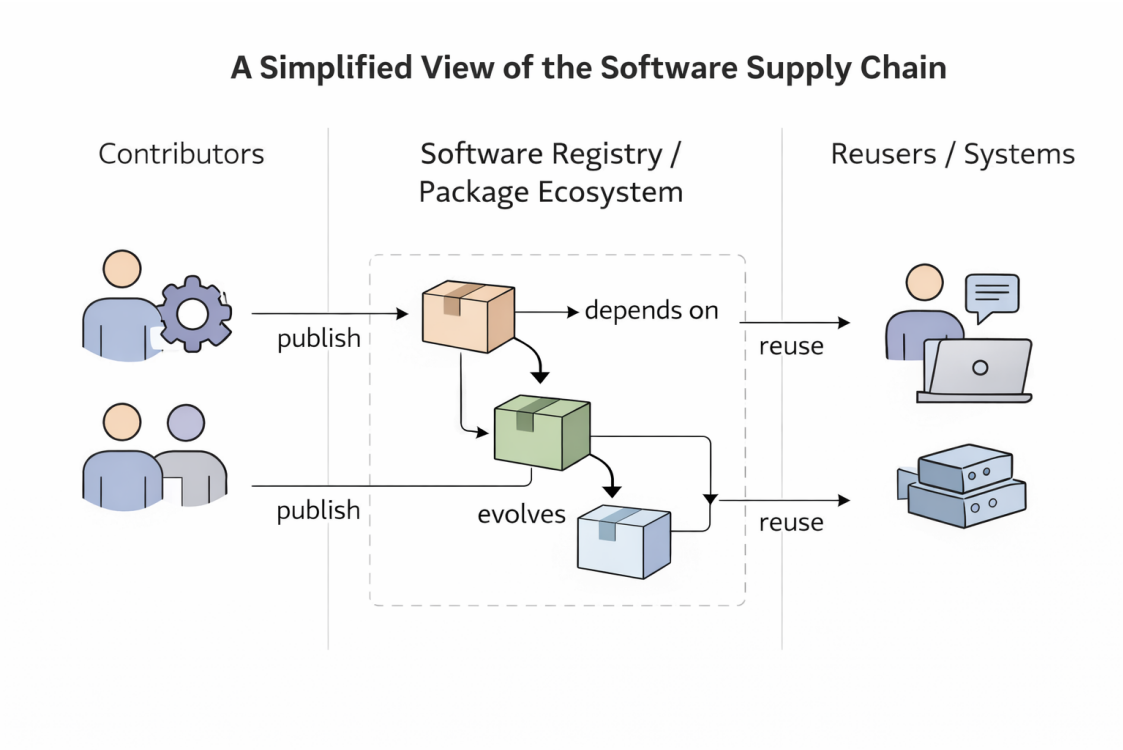

Figure 2.1. A simplified view of the software supply chain. Contributors publish packages to software registries, packages evolve through ongoing development and dependency relationships, and downstream reusers incorporate those packages into larger systems.

This model of software construction is fundamentally dependency-centric. Projects routinely rely on third-party packages to provide functionality ranging from low-level utilities to major frameworks and infrastructure libraries. Some of these dependencies are *direct*, meaning they are explicitly selected and declared by project maintainers in manifest files such as `package.json`, `requirements.txt`, or `pom.xml`. Others are *transitive*, meaning they are introduced indirectly through other dependencies [4, 14]. Consequently, the behavior, security posture, and maintenance profile of a software system are often shaped not only by its own code, but also by a much larger graph of externally sourced components [3, 4].

Package ecosystems provide the technical and social infrastructure that makes this style of development possible. Registries host reusable artifacts, enforce naming and versioning conventions, and support automated installation and update workflows. At the same time, ecosystems are not merely distribution channels. They also embody relationships among

contributors, maintainers, publishers, and downstream consumers, as well as conventions around release management, package discovery, and package trust [2, 15]. Table 2.1 summarizes several of the central actors and components in this ecosystem-oriented view of the software supply chain.

Table 2.1. Core actors and components in a software supply chain.

Component	Description
Packages	Reusable software artifacts that provide functionality to downstream consumers.
Contributors	Engineers and maintainers who create, update, and publish packages.
Registries	Platforms that host packages and support discovery, versioning, and distribution.
Reusers	Downstream developers or organizations that incorporate packages into their own projects.
Systems	The end-result software systems assembled from direct and transitive dependencies.

The scale of dependency-mediated reuse means that many software systems now inherit substantial amounts of functionality from upstream packages. This arrangement can significantly reduce development effort and accelerate delivery, since widely used capabilities can be incorporated through reuse rather than rebuilt from scratch. However, it also means that downstream projects inherit not only functionality, but also the maintenance practices, release cadence, dependency choices, and potential weaknesses of upstream components [16, 17]. In practice, software reuse therefore extends beyond simple code sharing: it creates ongoing technical and organizational relationships between upstream producers and downstream consumers.

This broader perspective is important for the rest of the thesis. Software supply chains are not merely lists of dependencies or inventories of packages. They are the larger socio-technical

systems through which software artifacts are produced, distributed, selected, updated, and inherited across projects. Understanding this structure is necessary for reasoning about how trust is established, how risk propagates, and how new development paradigms, including generative systems and AI coding agents, may alter the traditional model of software reuse.

2.2 Build-vs.-Buy in Software Sourcing

Traditional software sourcing is often understood through the build-vs.-buy abstraction. When engineers need new functionality, they must decide whether to implement it locally or obtain it from an external source, such as a reusable library, framework, service, or commercial off-the-shelf component [18–23]. Although this framing is simple, it captures a fundamental economic decision in software engineering: whether the expected cost of local development is lower or higher than the lifecycle cost of adopting and maintaining an externally produced artifact.

This tradeoff is broader than initial acquisition alone. Building functionality locally typically has high acquisition cost because engineers must design, implement, and test the functionality themselves, but it can provide tighter control over system fit, assurance, and long-term evolution. Buying or reusing an external component often has low acquisition cost because functionality can be incorporated quickly, but it may increase adaptation, assurance, and evolution costs because the project must integrate externally designed code, track upstream changes, manage compatibility, and inherit the component's dependency and security profile [18, 22–24].

A useful way to frame this tradeoff is with a simple total cost of ownership model. For a candidate sourcing strategy s, the total cost of ownership (TCO) can be viewed as:

$$TCO(s) = C_{\text{acquire}}(s) + C_{\text{adapt/integrate}}(s) + C_{\text{assure}}(s) + C_{\text{evolve}}(s) \tag{2.1}$$

Here, C_{acquire} captures the upfront cost of obtaining or producing the functionality, $C_{\text{adapt/integrate}}$ captures the effort needed to fit it into the surrounding system, C_{assure} captures testing, review, and validation, and C_{evolve} captures ongoing maintenance, compatibility work, and upgrade burden. This formulation is not intended as a calibrated predictive model,

but as a compact synthesis of the broader literature on software cost, reuse, and lifecycle tradeoffs [18, 21–23].

The model helps explain why the preferred sourcing strategy depends on the functionality being sourced. For small, well-scoped functionality, building locally may be attractive because acquisition cost is manageable and the implementation can be narrow and easy to adapt. For large or complex functionality, reusing an external component may be attractive because building from scratch would be costly. However, reuse can impose continuing assurance and evolution costs when the adopted package is much broader than the functionality actually exercised by the repository.

Generative systems expand this design space by introducing a third possibility: dependency replacement through use-case-oriented regeneration. Rather than manually building a full implementation or continuing to rely on a full upstream package, a project may regenerate only the portion of dependency functionality it actually uses. In TCO terms, regeneration may reduce acquisition and evolution costs by producing a narrower repository-specific implementation, but it may increase assurance cost because the generated replacement must be validated and may not preserve untested behaviors.

2.3 Trust and Its Limits in Software Supply Chains

In the traditional software supply chain, trust is largely mediated through signals tied to persistent upstream artifacts and the infrastructure around them. Developers reason about whether a dependency is acceptable by relying on cues such as package identity, ecosystem and registry context, maintainer reputation, version history, public advisories, and, increasingly, provenance and signing metadata [9, 15, 25, 26]. Software Bills of Materials (SBOMs) complement these mechanisms by helping organizations inventory dependencies and assess exposure to known vulnerabilities [27]. Taken together, these signals are intended to provide confidence that a component originates from an expected source, has not been tampered with, and remains suitable for continued use over time [9, 26]. Figure 2.2 summarizes this artifact-centric trust model.

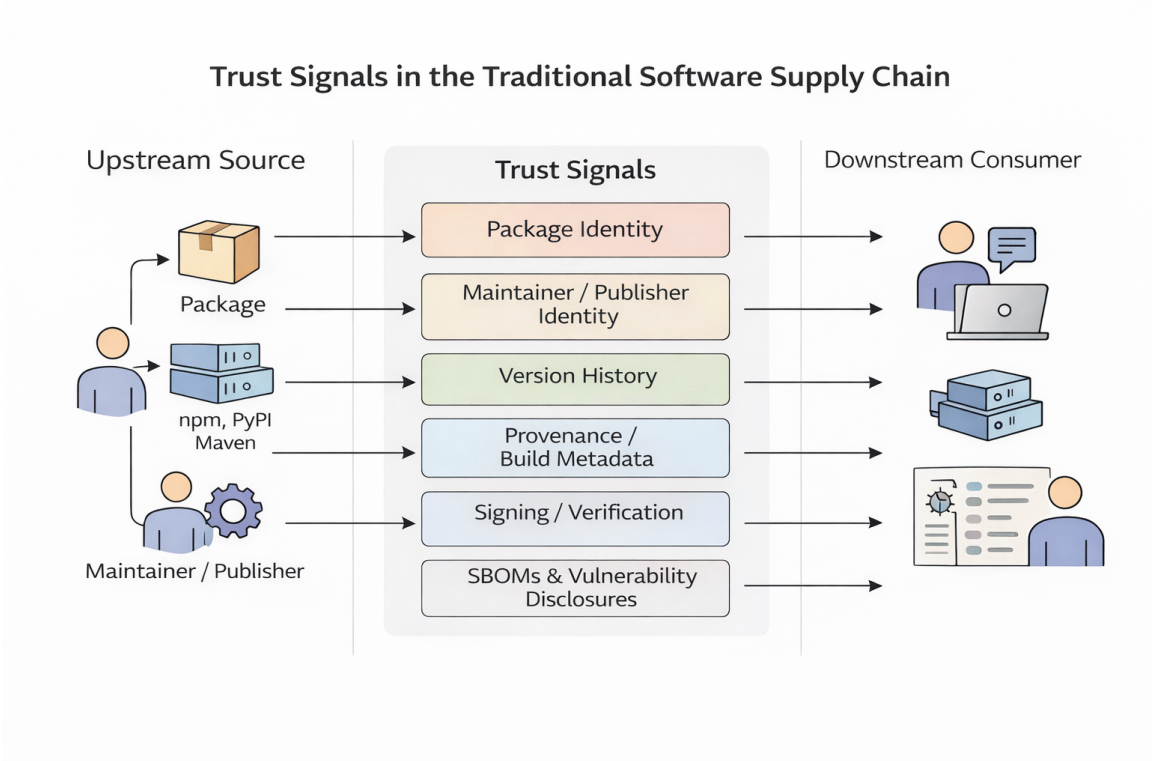

Figure 2.2. Trust in the traditional software supply chain is mediated through signals tied to persistent upstream artifacts and their surrounding infrastructure. Downstream consumers reason about packages using information such as package identity, maintainer identity, version history, provenance and build metadata, signing and verification, and SBOMs or vulnerability disclosures [9, 15, 26, 27].

At the same time, these trust signals are only partial proxies for the properties developers actually care about. They do not directly guarantee benign behavior, secure maintenance, or uncompromised release infrastructure. Instead, they depend on a larger chain of assumptions: that maintainers remain trustworthy, that accounts are not hijacked, that registries distribute the intended artifact, that build and release pipelines are not subverted, and that downstream users can correctly interpret the available signals. Trust in software supply chains is therefore not absolute. It is conditional on the continued integrity of upstream people, processes, and infrastructure, not just on the package name or version number alone [15, 25].

Recent supply chain incidents illustrate these limits clearly. In March 2026, malicious versions of the widely used `axios` npm package were published through a compromised maintainer account, introducing a hidden dependency that deployed a remote access trojan

during installation [28]. Similarly, compromises affecting widely used CI/CD distribution mechanisms, such as the force-updating of tags in the Trivy GitHub Action supply chain, showed that even trusted delivery paths and automation infrastructure can be turned into attack vectors [29]. Older ecosystem failures such as the `left-pad` unpublish incident highlight a related limitation: even when malicious code is not involved, downstream systems may still depend heavily on the continued availability and stability of small upstream packages [30]. Taken together, these incidents show that software supply chain trust is not only a question of whether an artifact is authentic, but also whether the assumptions that make dependency-mediated reuse workable continue to hold.

This limitation matters for the rest of the thesis because both empirical chapters examine how generative systems interact with artifact-centric trust. In Chapter 3, AI coding agents operate within existing package ecosystems and therefore influence which upstream artifacts are selected, updated, and trusted in practice. In Chapter 4, use-case-oriented regeneration raises a different possibility: that some repository-used slices of functionality may be sourced without continued reliance on a full upstream package at all. If local generation becomes a practical alternative in some settings, engineers may place greater emphasis on validating locally produced functionality rather than relying exclusively on signals attached to persistent upstream artifacts. Generative systems therefore do not eliminate the need for trust, but they may shift part of the trust problem from evaluating persistent upstream artifacts toward validating functionality.

2.4 Large Language Models and AI Coding Agents

Large language models have changed how software is produced, modified, and evaluated. Initial use of LLMs in software engineering often focused on localized tasks such as code completion, snippet generation, explanation, or small-scale repair. More recent systems operate at the level of repositories and engineering tasks: they inspect codebases, edit multiple files, invoke tools, respond to feedback, and iteratively refine solutions in order to complete non-trivial software engineering work [31–34].

This broader capability has given rise to AI coding agents. Rather than serving only as autocomplete systems, these agents function as active participants in development workflows. They may read repository context, update source files, modify tests, edit configuration, interact with manifests, call compilers or package managers, and use the results of tool execution to guide subsequent actions. In effect, they can participate in many of the same workflow steps that human developers use when implementing features, fixing bugs, or maintaining repositories.

Existing evaluations of repository-level agents have largely focused on functional success. For example, SWE-bench evaluates whether agents can resolve real GitHub issues, RepoBench evaluates repository-level code completion and retrieval, and DependEval evaluates repository-level dependency problem solving [31–33]. These benchmarks provide relatively objective measures of task performance, such as whether a generated patch passes tests, whether code can be completed correctly, or whether a dependency-related task is solved. Such evaluations are valuable for tracking progress in code synthesis and repository-level reasoning.

At the same time, objective task success does not capture all aspects of agent behavior that matter in software engineering. A patch may compile, pass tests, or complete a benchmark task while still making questionable engineering decisions, such as introducing an unnecessary dependency, selecting a poorly maintained package, increasing configuration complexity, or violating project policy. As a result, existing evaluations tend to foreground immediate functional success while underemphasizing the broader dependency, maintainability, governance, and policy decisions that arise during repository modification.

This distinction matters because AI coding agents do not operate outside software supply chains. When they modify repositories, they may also affect the set of external components on which those repositories depend. An agent may introduce a new library, remove an existing one, update package versions, or alter manifests in ways that change the composition of the resulting system. These actions can preserve functional correctness while still materially affecting security posture, attack surface, maintenance burden, and governance constraints. Generative systems are therefore relevant not only because they produce code,

but also because they can influence software supply chains through the repository changes they make.

3. AI CODING AGENTS AND DEPENDENCY DECISION-MAKING IN SOFTWARE REUSE

3.1 Introduction

Statement of Authorship. This chapter is based on *Towards a Benchmark for Dependency Decision-Making*, accepted to the Journal Ahead Workshop (JAWs) at ICSE 2026 [7]. I am the lead author and contributed to the study design, dependency-change extraction methodology, vulnerability-labeling procedure, data analysis, and writing. Collaborators contributed to dataset construction, methodological feedback, and manuscript revision.

As discussed in Chapter 2, modern software reuse is heavily dependency-mediated, and AI coding agents increasingly participate in repository-level software engineering tasks. This chapter builds on that background by focusing specifically on dependency decision-making as an observable mechanism through which agents affect software reuse in practice.

When coding agents modify repositories, they do not only generate source code. They may also add new libraries, remove existing packages, or update dependency versions as part of completing a task [31–34]. These choices are consequential because they determine which third-party code becomes part of the resulting system. A dependency change may preserve immediate correctness while still being undesirable from a broader software engineering perspective, for example if it introduces an unnecessary library, selects a risky version, or increases future remediation burden. Dependency decision-making therefore provides a concrete way to study how generative systems are already changing software reuse within existing ecosystems.

Existing evaluations of repository-level agents, however, largely emphasize functional success. Benchmarks commonly measure whether an agent completes the task, compiles the code, or passes tests [31–33]. These evaluations have been highly useful for tracking progress in repository-level reasoning and code synthesis, but they generally do not explicitly assess the quality of dependency decisions. As a result, an agent may appear successful under standard benchmarks even when it makes dependency choices that are undesirable from the perspective of security, maintenance, or policy.

To study this issue, this chapter analyzes dependency changes in agent-authored and human-authored pull requests across a large multi-ecosystem dataset. Across 2,807 GitHub repositories spanning seven ecosystems, the study examines 117,062 dependency changes drawn from 33,596 agent-authored pull requests and 6,618 human-authored pull requests. It evaluates how often agents modify dependencies, what kinds of dependency changes they make, and whether those choices have measurable security consequences under a PR-time view of vulnerability exposure. The results show that agents perform version updates more often than humans and, conditional on introducing or updating a dependency, select PR-time known-vulnerable versions more frequently [7].

These findings show that AI coding agents are already changing software reuse in practice through the dependency decisions they make in real repositories. This is a more immediate and measurable form of change than full generative replacement, yet it already has important implications for software security, maintenance, and evaluation.

The remainder of this chapter is organized as follows. Section 3.2 defines dependency decision-making as a software engineering activity. Section 3.3 presents the research questions. Section 3.4 describes the methodology. Section 3.5 presents the results. Section 3.6 discusses the implications of the findings, and Section 3.7 describes threats to validity.

3.2 Dependency Decision-making

Dependency decision-making is a core software engineering activity in projects that rely on third-party packages. In practice, maintainers must decide whether external functionality should be introduced at all, which package should be selected, which version should be adopted, whether an existing dependency can be reused instead of introducing a new one, and how the resulting choice should be encoded in project manifests [35, 36]. These decisions are not purely functional. They may also reflect organizational preferences and non-functional requirements, such as restricting adoption to trusted publishers, avoiding packages with poor maintenance signals, preferring stable versions, or choosing options that minimize future security and maintenance burden. Prior work has shown that these choices can be non-

trivial, with engineers often balancing convenience against compatibility, stability against security, and short-term implementation effort against long-term maintenance costs [37, 38].

Dependency choices matter because they determine which third-party code, together with its full transitive dependency chain, becomes part of the resulting system. A dependency can introduce functional bugs, performance regressions, license complications, or security vulnerabilities, and it can impose ongoing lifecycle costs as maintainers must track updates, compatibility breaks, and newly disclosed advisories over time [39, 40]. In this sense, dependency decision-making is not merely a packaging detail. It is a concrete mechanism through which software reuse affects a system's attack surface, maintenance profile, and long-term evolution.

This perspective is especially important in the context of AI coding agents. When these systems modify repositories, they do not only produce source code changes in isolation. They may also add libraries, remove packages, update versions, or otherwise alter dependency manifests as part of completing a task. As a result, they participate directly in dependency decision-making and therefore in software reuse within existing ecosystems. Studying those decisions provides a concrete way to measure how generative systems are already affecting software engineering practice, even before considering more ambitious forms of software generation or replacement. Despite their importance, dependency decisions are often not explicitly evaluated in existing repository-level agent benchmarks, which tend to emphasize functional outcomes such as compilation success, task completion, or passing tests [31–33]. As a result, an agent may appear successful even when it makes dependency choices that are undesirable from the perspective of security, maintenance, or policy.

3.3 Research Questions

This chapter studies how AI coding agents are changing software reuse through the dependency decisions they make in real repositories. It focuses on the prevalence, form, and security consequences of those decisions in comparison with human-authored pull requests.

To guide the analysis, this chapter addresses the following research questions:

RQ1. How often do AI coding agents modify dependencies in real software repositories?

RQ2. What kinds of dependency changes do AI coding agents make, and how do those changes compare with human-authored pull requests?

RQ3. How often do AI coding agents select PR-time known-vulnerable dependency versions?

3.4 Methodology

This chapter studies how AI coding agents change software reuse through the dependency decisions they make in real repositories. To do so, it analyzes dependency-related edits in agent-authored and human-authored pull requests and evaluates the security consequences of those edits under a pull-request-time view of vulnerability exposure. Human-authored pull requests are used as a contextual baseline rather than a directly matched control group, since the underlying tasks, constraints, and project contexts may differ across the two populations. Figure 3.1 summarizes this methodological pipeline.

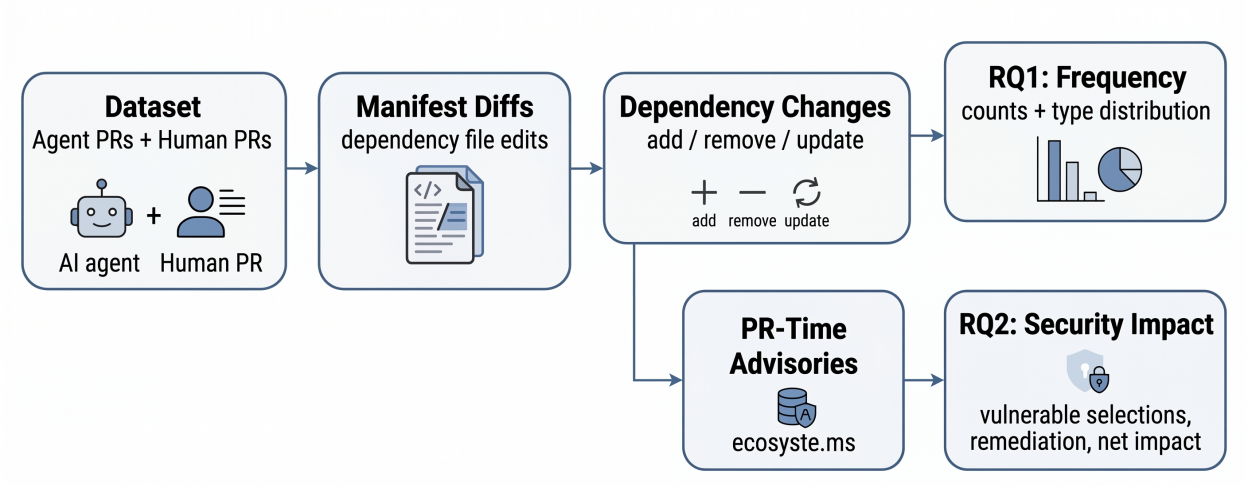

Figure 3.1. Methodological overview for Chapter 3. Starting from agent-authored and human-authored pull requests, the analysis identifies dependency-related manifest edits, classifies observable dependency changes as additions, removals, or version updates, and uses those changes to study both dependency-change frequency and PR-time security impact through advisory-aware labeling.

3.4.1 Dataset

The analysis uses the AIDev-pop dataset [34], which was released as part of the MSR 2026 Mining Challenge.[1] The dataset contains 33,596 agent-authored pull requests and 6,618 human-authored pull requests drawn from 2,807 GitHub repositories spanning seven package ecosystems. These data make it possible to study the dependency choices that agents and human developers made while completing real software engineering tasks in public repositories. In total, the dataset contains 117,062 observable dependency changes, including 53,277 changes in agent-authored pull requests and 63,785 changes in human-authored pull requests.

The goal of using this dataset is not to infer causal differences between agents and humans under identical task conditions. Instead, the dataset provides a large-scale observational view of how dependency changes occur in practice. Accordingly, the human-authored pull requests serve as a contextual baseline for comparison, while the primary goal is to characterize the

[1] https://2026.msrconf.org/track/msr-2026-mining-challenge

prevalence, form, and security consequences of dependency decisions made by AI coding agents.

3.4.2 Identifying Dependency Decisions

Dependency decision-making is operationalized through changes to dependency manifest files in pull requests. Each manifest edit is treated as an explicit decision point at which the author altered the project's dependency graph. This includes decisions to introduce a new dependency, remove an existing one, or change the version associated with a dependency.

For each pull request, the unified diff of the relevant manifest files is analyzed. Using manifest-format-specific parsing rules and regular expressions, the analysis extracts the affected package name and version information from the dependency edit. Each extracted change is then classified into one of three categories:

- **Dependency addition**: a new dependency is introduced into the manifest.
- **Dependency removal**: an existing dependency is deleted from the manifest.
- **Dependency version update**: the version or version constraint of an existing dependency is changed.

This representation treats manifest edits as observable reuse decisions made during software engineering work. It allows the analysis to measure both how frequently agents modify dependencies and what kinds of dependency changes they tend to perform.

3.4.3 Measuring the Prevalence and Form of Dependency Changes

To characterize how AI coding agents modify dependencies in practice, the analysis measures the frequency and distribution of manifest edits in the dataset. For each population, it reports the volume of dependency changes and the relative share of additions, removals, and version updates. This makes it possible to determine whether agents routinely engage in dependency decision-making and whether the kinds of changes they make differ from those observed in human-authored pull requests.

The analysis focuses on dependency changes as the primary unit of observation rather than pull requests alone. This choice reflects the fact that a single pull request may contain multiple dependency edits and therefore multiple distinct decision points. Studying changes at this granularity provides a more direct view of reuse behavior than treating each pull request as a single binary event.

3.4.4 PR-Time Vulnerability Labeling

To assess whether dependency decisions have measurable security consequences, the analysis evaluates whether a dependency addition or version update introduced a package version that was already known to be vulnerable at the time the pull request was submitted. This PR-time perspective is intended to approximate the information that would have been available to the author when making the dependency decision.

For each dependency change involving an addition or version update, the analysis extracts the package name and introduced version or version constraint. It then queries a registry-aware vulnerability source, specifically the ecosyste.ms vulnerability database, to identify advisories affecting that package that had been disclosed before the pull request timestamp. When available, the analysis also records the associated severity information and the first patched version.

Using this information, the chapter measures:

- the proportion of additions and version updates that introduce PR-time known-vulnerable dependency versions;
- whether a non-vulnerable version was already available at pull-request time;
- the remediation effort required to reach a patched release, approximated by the semantic version distance to the first available non-vulnerable version; and
- the aggregate security impact of dependency maintenance activity, computed as the balance between vulnerability-introducing and vulnerability-fixing dependency edits.

Table 3.1 summarizes these metrics, including the intuition behind each measure and how it is operationalized in our analysis. This approach does not attempt to measure ex-

ploitability or runtime reachability. Instead, it uses advisory availability at pull-request time as a practical approximation of whether a dependency decision was security-relevant and whether safer alternatives were already available when the change was made.

Table 3.1. Metrics used to evaluate the PR-time security impact of dependency decisions.

Metric	Intuition	Measurement
Vulnerable selection rate	How often dependency choices introduce known-vulnerable versions.	Percentage of dependency additions and version updates that select a version affected by a vulnerability disclosed before the pull request was opened.
Mitigatability	Whether a safer version was already available.	Percentage of vulnerable selections for which a non-vulnerable version existed at pull-request time.
Remediation effort	How disruptive the fix may be.	Semantic-version distance from the selected vulnerable version to the first non-vulnerable version: patch, minor, major, or other.
Net security impact	Overall effect of dependency maintenance on known vulnerability exposure.	Number of vulnerability-fixing dependency edits minus vulnerability-introducing dependency edits.

3.4.5 Comparison Against Human-Authored Pull Requests

To contextualize the observed behavior of AI coding agents, the analysis compares agent-authored pull requests against human-authored pull requests from the same dataset. These human-authored pull requests are not treated as matched controls, since the two groups may differ in task mix, repository context, or other unobserved constraints. Instead, they provide a practical baseline that helps interpret whether the dependency decision patterns observed in agent-authored pull requests appear unusual in the context of routine software maintenance.

This comparison is used throughout the chapter when reporting the relative frequency of additions, removals, and version updates, as well as when evaluating PR-time vulnerability rates and remediation burden. The goal is not to claim that any observed differences are

solely caused by authorship type, but rather to situate agent behavior against a meaningful reference point drawn from real software engineering practice.

3.4.6 Scope of the Analysis

The methodology in this chapter is designed to capture observable dependency decisions and their immediate security implications. It does not evaluate whether every added dependency was necessary, whether an alternative already existed in the repository, or whether a selected version was optimal with respect to all possible project constraints. Those questions require richer task context than can be recovered reliably from manifest diffs alone.

Similarly, the security analysis is limited to publicly available vulnerability advisories and registry metadata. As a result, it reflects advisory-visible risk rather than full downstream exploitability. Even with these limitations, the methodology provides a concrete and scalable way to study how AI coding agents participate in software reuse through dependency changes in real repositories, and to assess whether those changes carry meaningful security consequences.

3.5 Results

This section presents the results of the empirical analysis of dependency changes in agent-authored and human-authored pull requests. It is organized around the three research questions introduced earlier in the chapter.

3.5.1 RQ1: How often do AI coding agents modify dependencies?

Dependency modification is a recurring part of agent-authored repository work in the AIDev-pop dataset. We identified 117,062 dependency changes in the AIDev-pop dataset, of which 53,277 occur in 33,596 agent-authored pull requests. The remaining 63,785 dependency changes occur in 6,618 human-authored pull requests. These changes represent observable dependency decisions made by pull request authors while completing software engineering tasks.

The scale of this activity shows that AI coding agents do not merely generate source code while leaving dependency structure untouched. They also participate routinely in manifest-level repository maintenance that changes how projects depend on external components. In this sense, dependency decision-making is not an edge case in agent-authored repository work. Instead, it is a recurring part of how coding agents complete real software engineering tasks.

These observations make dependency modification a useful lens for studying how generative systems are already changing software reuse in practice. They show that the role of coding agents extends beyond producing functionally correct patches. Agents also make decisions that alter the set of external components on which a project depends, thereby affecting how software reuse unfolds within existing ecosystems.

3.5.2 RQ2: What kinds of dependency changes do AI coding agents make, and how do those changes compare with human-authored pull requests?

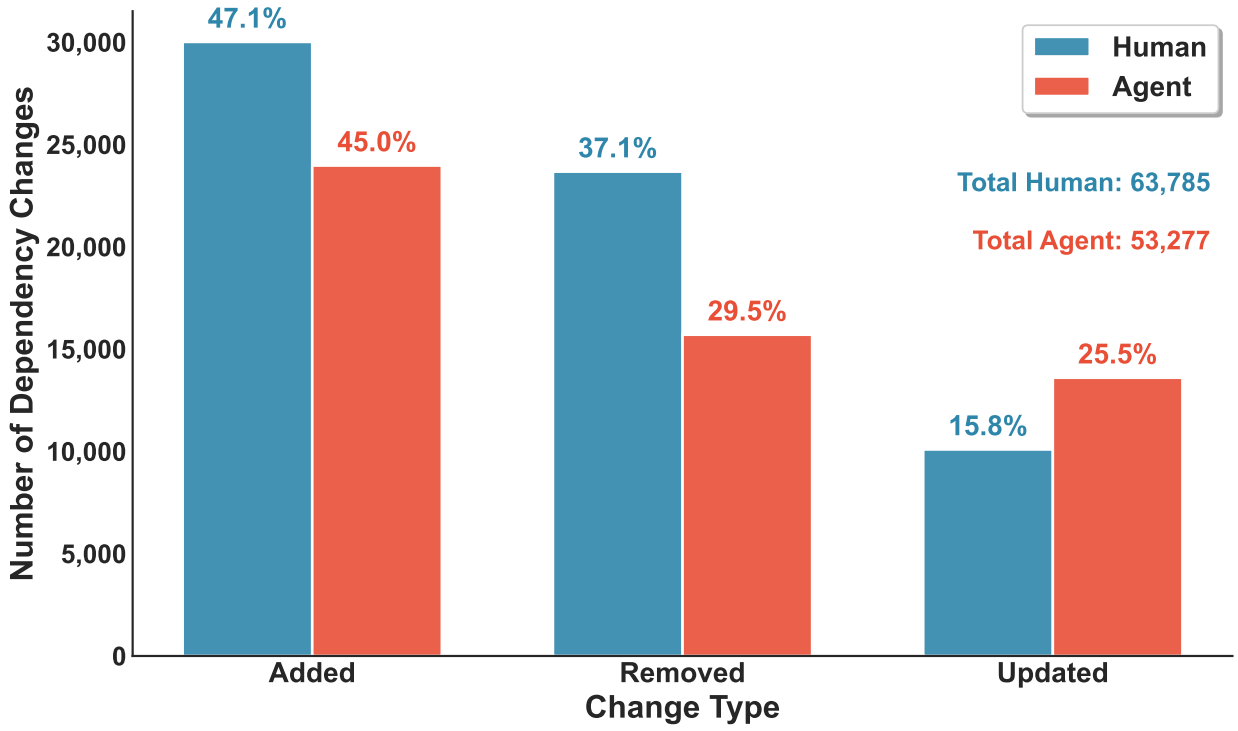

Figure 3.2. Breakdown of dependency changes made by agents and humans.

Classifying each dependency change by type shows that agent-authored dependency edits are diverse. Among agent changes, 45.0% are dependency additions, 29.5% are removals, and 25.5% are version updates (fig. 3.2). By comparison, human-authored dependency edits are 47.1% additions, 37.1% removals, and 15.8% version updates.

These results reveal two important patterns. First, additions are the most common type of dependency change for both groups, indicating that software maintenance often involves introducing new externally sourced functionality rather than only adjusting existing dependencies. Second, agents perform version updates substantially more often than humans (25.5% vs. 15.8%), while humans remove dependencies more often than agents (37.1% vs. 29.5%). This suggests that AI coding agents are comparatively more likely to intervene through version selection and version movement, whereas human-authored maintenance includes relatively more removal activity.

Taken together, these findings show that AI coding agents make multiple kinds of software reuse decisions. They do not only introduce new dependencies. They also remove existing dependencies and alter the versions of reused components already present in a project. This indicates that agent-authored software maintenance is not limited to localized code generation, but includes repeated interventions in the dependency graph of real repositories.

3.5.3 RQ3: How often do AI coding agents select PR-time known-vulnerable dependency versions?

To evaluate the security consequences of dependency decisions, we focus on additions and version updates, since these edits require selecting a concrete dependency version or version constraint. Table 3.2 reports the resulting PR-time vulnerability and remediation metrics.

Table 3.2. PR-time vulnerability and remediation metrics for dependencies introduced via additions and updates.

Metric	**Agent**	**Human**
Dependencies introduced or updated	37,574	40,110
Vulnerable dependencies	924 (2.46%)	659 (1.64%)
Mitigatable (safe version available)	86.58%	83.31%
Remediation effort for vulnerable selections		
Bug-fix (1.2.X $\rightarrow$ 1.2.Y)	188 (20.3%)	266 (40.4%)
Minor (1.X $\rightarrow$ 1.Y)	351 (38.0%)	283 (43.0%)
Major (1 $\rightarrow$ 2)	340 (36.8%)	85 (12.9%)
Other	45 (4.9%)	25 (3.8%)

Agent dependency decisions select PR-time known-vulnerable versions more often than human decisions. Conditional on introducing or updating a dependency, the vulnerable-selection rate is 2.46% for agents compared with 1.64% for humans. This indicates that the dependency decisions made by AI coding agents are not only frequent, but also measurably security-relevant.

In many of these cases, safer alternatives were already available when the pull request was opened. Across vulnerable agent selections, 86.58% had a patched version available at PR time but did not select it. This suggests that a substantial fraction of unsafe dependency choices were avoidable under comparatively simple advisory-aware decision procedures.

The severity distribution of introduced vulnerabilities provides additional context for these outcomes. Among agent-introduced vulnerable selections, 52.3% are Moderate, 26.0% are High, 12.0% are Critical, and 9.7% are Low. Human-authored vulnerable selections include a larger share of Critical cases (23.3%), with the remainder distributed across Moderate (44.9%), High (21.7%), and Low (10.1%) severities. Thus, agent-introduced vulnerabilities are concentrated primarily in the Moderate and High ranges, while human-authored selections include relatively more Critical issues.

The remediation burden associated with unsafe selections is also notable. Among vulnerable versions introduced by agents, 36.8% (340) require a major-version upgrade to reach a patched release, compared with 12.9% (85) for human selections. By contrast, human-authored vulnerable selections more often admit non-breaking remediation paths, including bug-fix and minor upgrades. This means that unsafe dependency decisions by agents are not only more frequent, but also more likely to impose disruptive follow-up maintenance when remediation is required.

Finally, aggregating outcomes across all dependency edits shows that agent-authored dependency work is net security-negative in this dataset. Agent-authored pull requests fix 826 vulnerabilities but introduce 924, yielding a net impact of −98. Human-authored pull requests fix 1,975 vulnerabilities and introduce 659, yielding a net impact of +1,316. Human-authored fixes also target Critical vulnerabilities more frequently (33.3% vs. 15.9%).

Overall, these results show that AI coding agents are already changing software reuse in ways that matter beyond immediate task completion. Their dependency decisions affect which third-party code becomes part of a system, whether known-vulnerable versions are adopted despite available fixes, and how difficult those choices may be to repair later. These are important practical consequences that remain largely invisible to evaluations focused only on functional correctness or test passing.

3.6 Discussion

The results of this chapter show that AI coding agents are already changing software reuse in practice. They do so not by replacing software supply chains outright, but by participating in the everyday dependency decisions through which projects add, remove, and update third-party components. This is a more immediate and observable form of change than full generative replacement, yet it already has important implications for software security, maintenance, and evaluation.

The prevalence and diversity of dependency changes show that agent-authored repository maintenance extends beyond localized code generation. Agents routinely alter dependency manifests, meaning that they do not only produce source code but also reshape the set of

external components on which a project depends. The comparison with human-authored pull requests further suggests that agent-authored reuse behavior is not simply a smaller version of human maintenance behavior. Agents perform version updates more frequently, while humans remove dependencies more often. Because version-selection decisions can introduce security and compatibility consequences without breaking immediate functionality, this helps explain why a task can appear successful under test-based evaluation even when the underlying dependency choice is undesirable.

The PR-time vulnerability analysis makes this problem more concrete. Conditional on introducing or updating a dependency, agents select known-vulnerable versions more often than humans, and many of these selections were avoidable because safer releases were already available at pull-request time. Unsafe selections by agents are also more likely to require major-version jumps to reach a patched release, increasing follow-up maintenance burden. Taken together, these findings show that AI coding agents already participate in software reuse in ways that materially affect downstream security and maintenance, even within the traditional package-centric software supply chain.

3.7 Threats to Validity

This chapter has several limitations. The analysis relies on public vulnerability advisories and registry metadata, so it captures advisory-visible risk rather than full exploitability or runtime reachability. Similarly, remediation effort is approximated using the semantic version jump needed to reach a non-vulnerable release, which is useful as a proxy for disruption but does not fully capture project-specific compatibility constraints or engineering effort. The extraction of dependency changes from manifest diffs is also necessarily approximate, and unusual manifest formats or non-standard edits may be missed or misclassified.

In addition, the comparison between agent-authored and human-authored pull requests should be interpreted as contextual rather than causal. The two groups may differ in task mix, repository context, and other unobserved constraints, so the observed differences cannot be attributed solely to authorship type. The dataset is also limited to public GitHub repositories and a fixed set of contemporary coding agents, which may limit generalization to

private repositories, other ecosystems, or future agent systems. Even with these limitations, the chapter provides a concrete and scalable view of how AI coding agents participate in software reuse through dependency decisions in real repositories.

4. USE-CASE-ORIENTED REGENERATION AS A SOFTWARE SOURCING STRATEGY

4.1 Introduction

Chapter 3 showed that generative systems already affect software reuse within existing package ecosystems through the dependency decisions they make. This chapter examines a different and more fundamental possibility: whether generative systems can change the *unit of software sourcing itself* through use-case-oriented regeneration.

Traditional software sourcing is organized around persistent upstream artifacts such as packages, libraries, and frameworks. Reusable functionality is typically obtained by selecting and maintaining an external component over time [23, 41, 42]. Prior work on dependency regeneration or replacement has largely operated at this level, asking whether a dependency can be recreated as a general-purpose artifact [43, 44].

This chapter shifts the perspective to the downstream repository. Rather than asking whether a dependency can be reproduced in full, it asks whether the *subset of functionality actually exercised by a repository* can be regenerated locally while preserving repository behavior.

This shift is motivated by a simple observation: client repositories often rely on only a bounded portion of a dependency's API surface. Accordingly, the relevant unit of analysis is not the package in isolation, but the repository–dependency pair. The evaluation target also changes: instead of full package equivalence, the goal is to preserve behavior under the repository's existing validation artifacts.

This reframing has both practical and conceptual implications. Practically, if use-case-oriented regeneration is viable, it may reduce dependency surface, simplify carried functionality, and give developers greater control over assurance and evolution for the code they actually rely on. Conceptually, it challenges the assumption that software reuse must always be mediated through stable upstream artifacts. While whole-package reuse remains central to modern software engineering, generative systems may make selective regeneration plausible when repository usage is narrow and observable [45–49].

To investigate this possibility, this chapter evaluates whether an agent can regenerate exercised dependency functionality across a benchmark of repository–dependency pairs. It studies three related questions: whether regeneration is feasible for a given pair, how well generated implementations preserve repository behavior in practice, and how those implementations differ from the original dependency in terms of size and dependency footprint. Section 4.3 formalizes these questions.

4.2 Background: Dependency Minimization and Internalization Strategies

Traditional software sourcing is typically organized around whole external artifacts. When developers need functionality, they often choose between implementing it locally and adopting an existing dependency. Under this framing, the dominant unit of reuse is the external artifact itself: projects select, version, and maintain packages as whole units rather than only the subset of functionality they actually exercise [18, 19, 21–23, 41]. This artifact-centric view underlies much of modern package-based software development.

At the same time, repositories do not always use the entirety of the functionality they adopt. This has motivated several lines of work that attempt to reduce, internalize, or otherwise narrow the effective footprint of dependencies after or alongside adoption. These approaches are related to the ideas studied in this chapter, but they differ in both mechanism and objective. This section distinguishes four such strategies: debloating, tree shaking and dead-code elimination, vendoring, and regeneration.

The remainder of this section distinguishes use-case-oriented regeneration from several related strategies for reducing or internalizing dependency reliance. Table 4.1 summarizes these strategies along several dimensions: whether they operate before or after dependency adoption, whether they preserve or replace the upstream implementation, whether they target the full dependency or only repository-used functionality, and what form of reliance remains after the transformation.

4.2.1 Debloating

Debloating seeks to remove unnecessary code from an existing software artifact while preserving required behavior. In the dependency setting, the goal is typically to reduce attack surface, code size, or maintenance burden by eliminating functionality that is not needed by the downstream system. Prior work shows that some redundant dependency code can be removed without breaking existing tests, although the practical effectiveness of such techniques may depend on language and ecosystem characteristics, including dynamic loading behavior and reflection-like mechanisms [50, 51].

Conceptually, debloating begins from an already adopted artifact. The dependency remains the source artifact, but some portion of its code is identified as unnecessary and removed. Thus, debloating primarily changes the amount of code retained from a dependency, rather than reconsidering whether the full upstream package relationship is needed in the first place.

4.2.2 Tree Shaking and Dead-Code Elimination

Tree shaking and dead-code elimination remove code that is unreachable from the perspective of a particular build, bundle, or execution path. These techniques are especially common in front-end and bundling workflows, where unused exports or modules can sometimes be excluded from shipped artifacts [48, 49].

Like debloating, these techniques usually operate after a dependency has already been selected. Their goal is to reduce shipped or reachable code, not to replace the dependency as the source of functionality. This distinction is especially relevant in JavaScript ecosystems. For example, webpack documentation notes that tree shaking depends on statically analyzable ES module structure and therefore does not operate in the same way for CommonJS modules [49]. Similarly, Liu et al. show that bloated dependencies remain common in server-side JavaScript packages built on CommonJS, reporting that 50.6% of dependencies in their dataset were bloated, including 13.8% of direct dependencies [51].

These results highlight both the value and the limits of post-adoption minimization. Even when shipped code is reduced, the downstream repository may still retain the broader

dependency relationship, including versioning obligations, transitive dependencies, and trust assumptions tied to the upstream artifact.

4.2.3 Vendoring

Vendoring internalizes an upstream dependency by copying its source code into the downstream repository [52, 53]. This can provide tighter local control over patching, review, distribution, or build reproducibility, since the dependency is no longer fetched only as an external package at build or install time. However, vendoring still begins from the original upstream implementation and typically aims to preserve that implementation, rather than generating a repository-specific replacement.

Accordingly, vendoring changes where the code is carried and maintained, but not necessarily the underlying sourcing logic. The repository still depends, conceptually and semantically, on the upstream artifact that was copied. The full codebase may now live locally, yet the downstream project often still inherits its design, API structure, maintenance burden, and assurance needs.

4.2.4 Regeneration

Regeneration offers a different response to the costs and risks of dependency-mediated reuse. Instead of continuing to rely on an upstream package, a developer may attempt to recreate the needed functionality as local code. In principle, this can reduce several forms of ongoing external reliance. A local implementation does not require the downstream project to continuously track upstream releases, audit future package updates, or inherit new transitive dependencies introduced by the upstream maintainer. It may also reduce exposure to future supply chain failures such as compromised maintainer accounts, malicious releases, backdoored updates, or registry-level disruption.

This motivation is especially relevant when the main cost of reuse is not the initial act of obtaining functionality, but the long-term burden of trusting and maintaining an external artifact. A dependency may be easy to install, but adopting it also means accepting its future evolution, release practices, security posture, transitive dependency graph, and maintenance

assumptions. If the downstream project needs only a small amount of functionality, continued reliance on the full upstream package may be disproportionate to the actual use case. In such cases, regeneration offers a potential way to keep the needed behavior while reducing ongoing dependence on the broader artifact.

However, regenerating an entire dependency is often unnecessary and impractical. Many packages are general-purpose artifacts designed to support a wide range of users, configurations, APIs, and edge cases. A downstream repository typically exercises only a subset of that functionality. Recreating the full dependency would therefore reproduce much of the same scope that made the dependency costly to trust, audit, and maintain in the first place. Full-package regeneration is also difficult to validate, since preserving all behaviors of a mature package across all possible inputs would require far more evidence than most downstream projects possess.

Use-case-oriented regeneration narrows this task. Rather than attempting to recreate a dependency as a complete general-purpose package, it targets only the functionality actually used by a specific downstream repository. The goal is to replace first-party runtime reliance on the external dependency with a local implementation that preserves repository-observed behavior under the repository's available validation artifacts. This makes the repository–dependency pair, rather than the dependency alone, the relevant unit of analysis. The resulting implementation may be smaller and easier to inspect than the original package, but it also introduces new assurance obligations: developers must validate that the generated replacement preserves the behaviors their repository depends on and does not introduce new defects.

Table 4.1. Comparison of related strategies for reducing, internalizing, or replacing dependency functionality. Debloating, tree shaking, and vendoring begin from an adopted or copied upstream artifact. Use-case-oriented regeneration differs in that it targets the repository-used slice of functionality as the unit of sourcing and replacement.

Approach	**Starts from**	**Main goal**	**Relies on full upstream artifact?**	**Assurance basis**
Debloating	Adopted artifact	Remove unnecessary code and reduce attack surface	Yes, typically	Tests, reachability analysis, removal validation
Tree shaking / DCE	Adopted artifact and build graph	Remove unreachable shipped code	Yes	Build semantics, static analysis, tests
Vendoring	Upstream source artifact	Internalize code for local control, patching, or reproducibility	Yes, conceptually	Review and maintenance of copied upstream code
Use-case-oriented regeneration	Repository–dependency pair	Replace repository-used functionality with a local implementation	Not necessarily	Repository validation artifacts and local validation

Table 4.1 summarizes these distinctions. Debloating, tree shaking, and vendoring are all useful techniques, but they share an important common assumption: the upstream artifact has already been adopted as the source of functionality. Use-case-oriented regeneration differs because it questions whether continued reliance on that full artifact is necessary once the repository's actual usage is taken into account.

4.3 Research Questions

This chapter asks three questions:

RQ1. Given a client repository and a dependency it uses, can an agent regenerate the exercised dependency functionality as a local implementation that preserves repository behavior?

RQ2. To what extent do generated implementations preserve repository behavior across repository–dependency pairs?

RQ3. How do generated implementations compare with the original dependency in terms of size and dependency footprint?

4.4 Experimental Design

This section describes the design of our use-case-oriented regeneration study. Our goal is not to evaluate whether a model can recreate a package in full, but whether an agent can regenerate the subset of dependency functionality that a client repository actually exercises. This use-case-oriented framing differs from prior dependency regeneration work that focuses more directly on recreating or replacing the dependency itself as a general-purpose artifact [43, 44]. Accordingly, the study is organized around repository–dependency pairs rather than dependencies in isolation, and evaluates regeneration relative to each repository's existing validation artifacts.

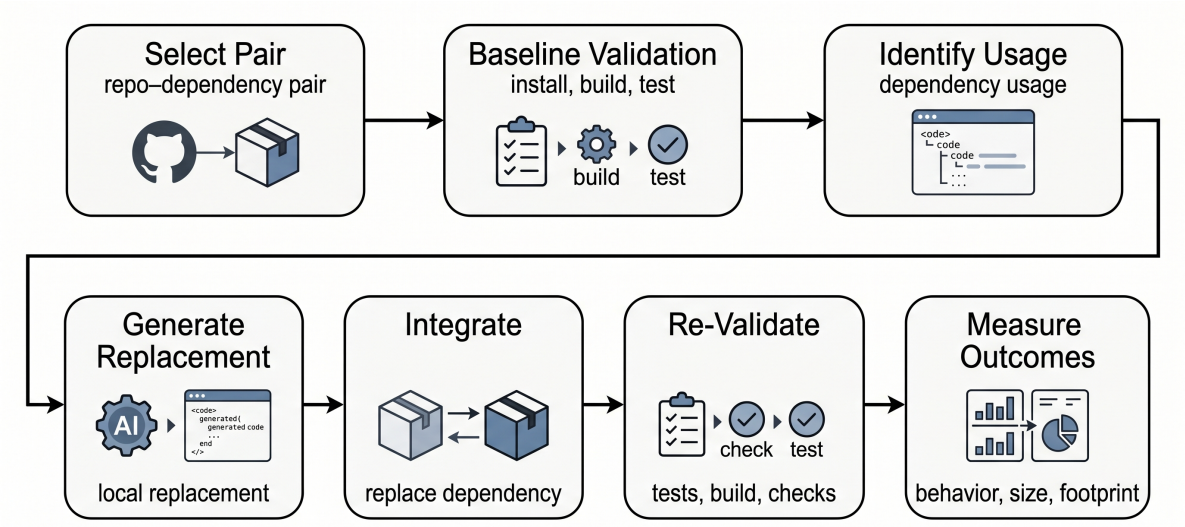

Figure 4.1. Experimental workflow for evaluating use-case-oriented regeneration over repository–dependency pairs. The agent receives the prompt, client repository, validation artifacts, and target dependency as inputs. It then generates a local implementation for the repository-used functionality, integrates that implementation into the repository, reruns the repository's validation artifacts, and measures outcomes in terms of behavioral preservation, size, and dependency footprint differences.

Figure 4.1 summarizes the workflow used to evaluate use-case-oriented regeneration over repository–dependency pairs.

4.4.1 Task Formulation

We formulate use-case-oriented regeneration as a task over repository–dependency pairs. Each instance consists of a client repository R, a third-party dependency D used by R, and the subset of dependency functionality F_D^R that is exercised by R. The objective is to replace runtime reliance on D with a local implementation D' while preserving the behavior of D as observed through the repository.

Let I_R denote the set of inputs exercised by the repository's validation artifacts, including tests, builds, and other checks. Let $\mathcal{B}(R, D, \mathrm{i})$ denote the observable behavior of repository R when it uses dependency D on input $\mathrm{i} \in I_R$. A generated replacement D' is correct with respect to repository R if it preserves this observed behavior:

$$\forall \mathrm{i} \in I_R, \quad \mathcal{B}(R, D, \mathrm{i}) = \mathcal{B}(R, D', \mathrm{i}). \tag{4.1}$$

This correctness condition is intentionally repository-scoped. It does not require D' to be semantically equivalent to D for all possible inputs, APIs, or downstream clients. Instead, it requires equivalence only over the behavior exercised by R and captured by its available validation artifacts. Thus, use-case-oriented regeneration evaluates correctness relative to repository-observed behavior rather than full dependency reimplementation.

This formulation differs from full dependency reimplementation. The goal is not to reproduce the entire API surface or all possible behaviors of the original package. Instead, the goal is to construct a local replacement that is sufficient for the specific repository–dependency pair under evaluation.

4.4.2 Dependency Selection

We target nine JavaScript/TypeScript dependencies chosen to span a range of package sizes, ecosystem roles, and expected regeneration difficulty. To avoid drawing conclusions

from only one class of package, we include small utility-style dependencies, medium-sized infrastructure libraries, and larger, more feature-rich packages. These dependencies vary in purpose, including identifier generation, case conversion, terminal styling, HTTP communication, schema validation, semantic version handling, CSS transformation, general-purpose utility functionality, and web framework behavior.

This selection is intended to provide variation along several dimensions that may affect regeneration feasibility, including approximate package size, popularity, dependent count, library role, and the likely scope of functionality exercised by downstream repositories. Our aim is not to construct a statistically representative sample of all npm packages, but to evaluate use-case-oriented regeneration across a deliberately varied set of dependencies for which repository-specific regeneration may plausibly range from straightforward to difficult.

Table 4.2. Target dependencies used in our study, grouped by approximate package size from the npm `Code` tab. Sizes are estimated by summing the visible folders and files shown by npm. Weekly downloads and dependent counts are taken from the corresponding npm package pages. Counts were collected as of April 15th, 2026

Size Tier	Dependency	Type / Role	Weekly Downloads	Dependents	Size
Small	`nanoid`	ID generation	131,153,197	16,911	12.7 kB
	`change-case`	String case conversion	18,139,142	4,576	35.9 kB
	`chalk`	CLI string styling	379,842,768	149,745	44.4 kB
Medium	`express`	Web server framework	84,716,349	102,105	75.4 kB
	`semver`	Version parsing / ranges	589,991,606	37,457	97.8 kB
	`postcss`	CSS processing framework	185,842,580	15,516	203.8 kB
Large	`lodash`	General utility library	125,235,251	196,953	1.44 MB
	`axios`	HTTP client	93,115,906	174,704	2.44 MB
	`zod`	Schema validation	133,003,378	78,701	4.34 MB

4.4.3 Repository Selection

For each target dependency, we identified candidate client repositories through code search over public GitHub repositories that imported the dependency. We then filtered this pool to retain repositories that were evaluable in practice: each repository had to build in a clean environment, expose validation artifacts such as test suites or executable scripts, and pass its baseline validation before any regeneration attempt was made.

From this filtered pool, we selected repositories for regeneration experiments. Selection was not intended to be statistically representative of all downstream clients. Instead, it was guided primarily by evaluability and bounded dependency usage. When multiple repositories were similarly suitable, repository popularity was used as a secondary preference signal to prioritize projects with greater visibility or ecosystem relevance.

4.4.4 Baseline Preparation

For each repository–dependency pair, we first established a clean baseline before attempting regeneration. We cloned the repository at a fixed revision, installed its declared dependencies using the package manager expected by the project, and ran the repository's existing validation artifacts, such as its test suite or executable checks. Only repositories that were runnable and for which baseline validation succeeded were retained for direct before/after comparison.

This step served two purposes. First, it reduced the risk of attributing failures to the regeneration attempt when they instead originated from a broken starting state. Second, it provided a baseline count of passing validation checks against which post-regeneration behavior could be compared. Repositories with missing system dependencies, incompatible package-manager requirements, or pre-existing validation failures were excluded, flagged as partial cases, or analyzed separately depending on the nature of the issue.

4.4.5 Agent, Prompting, and Regeneration Procedure

For each repository–dependency pair, we attempted use-case-oriented regeneration using an agentic workflow implemented with Claude Code. All regeneration attempts were conducted using Claude Code with Claude Opus 4.6 in the 1M-token context configuration [54, 55]. The agent was given access to the client repository, the source code of the target dependency, and the repository's existing validation artifacts, including its test suite. We allowed the agent to use Claude Code's default internal tooling for repository inspection, file editing, and command execution. Figure 4.2 provides a more detailed view of the information available to the agent and the immediate execution loop through which local implementations were generated, integrated, and revalidated.

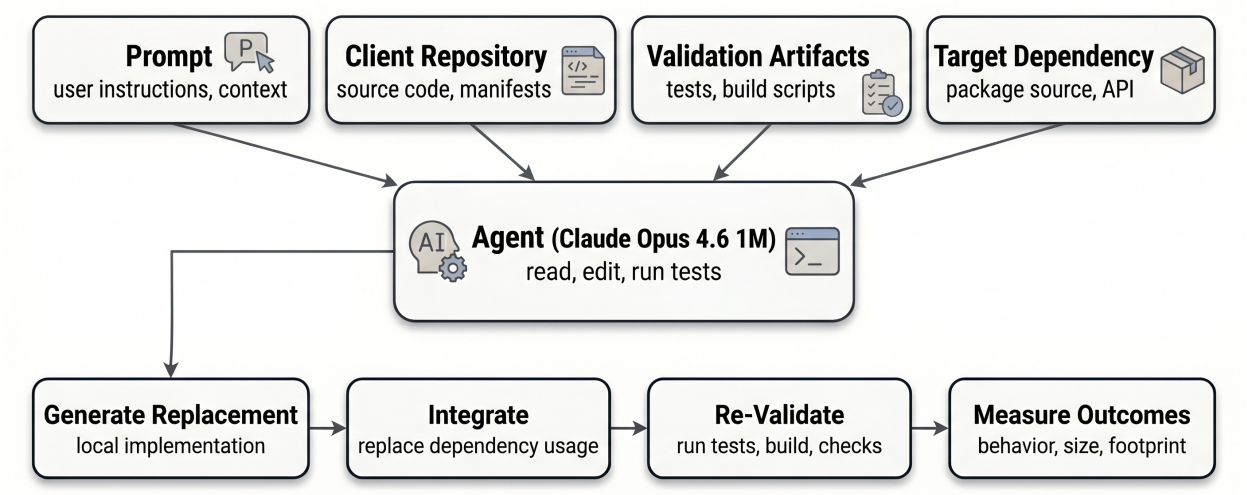

Figure 4.2. Inputs and execution context for the regeneration agent. For each repository–dependency pair, the agent receives the user prompt, the client repository, the repository's validation artifacts, and the target dependency's source and API context. Using these inputs, it reads and edits files, executes validation steps, generates a local implementation for repository-used functionality, integrates that implementation into the repository, and reruns validation before outcomes are measured.

The prompt was use-case-oriented rather than dependency-oriented. The agent was instructed to eliminate first-party runtime reliance on the target dependency while preserving repository behavior and baseline validation outcomes. It was explicitly directed not to recreate the dependency broadly, but instead to implement only the subset of functionality that

the repository actually required. The prompt also emphasized minimizing change surface, preserving existing coding style, and maintaining compatibility with the repository's language and build conventions. We did not conduct a formal prompt-engineering or ablation study over alternative prompt variants. Instead, we used a single fixed prompt template across repository–dependency pairs in order to keep the task formulation consistent. Our goal was to evaluate use-case-oriented regeneration under a stable prompting setup rather than to optimize prompt wording for maximum success. Accordingly, the results reported in this chapter should be interpreted with respect to this prompt formulation, not as an upper bound obtained through prompt optimization.

Operationally, the agent first identified first-party imports of the target dependency and inspected how those imports were used within the repository. It then established a baseline by running the repository's validation steps and recording the passing result on the original codebase. Next, it created local modules implementing only the repository-observed behavior needed for the task, updated first-party imports to use those local modules, and reran validation. When validation failed, the resulting errors and failing test outputs were used to iteratively refine the generated implementation. We instructed the agent to remove the dependency from the repository configuration only when it was no longer required by first-party source code, tests, or repository tooling. Accordingly, successful regeneration did not always require complete elimination of the package from every part of the repository. Our primary criterion was use-case-oriented regeneration: whether first-party repository behavior could be preserved without continued runtime reliance on the original external dependency.

The prompt template used in the study is shown below. The following prompt template was instantiated separately for each repository–dependency pair by replacing `[REPOSITORY_PATH]` and `[TARGET_DEPENDENCY]` with the corresponding repository path and target dependency. The installation and validation commands were adapted to the package manager and validation artifacts used by each repository.

```
You are working in [REPOSITORY_PATH].

Goal:
Replace first-party runtime reliance on [TARGET_DEPENDENCY] with
```

local helper implementations while preserving repository behavior and baseline validation outcomes.

Scope:
- Focus on first-party repository code.
- Prioritize runtime/application code over dev-only or tooling-only usage.
- Do not attempt to recreate [TARGET_DEPENDENCY] broadly; implement only the repository-observed behavior that is actually needed.
- Minimize change surface and preserve existing coding style and TypeScript compatibility where applicable.

Tasks:
1. Identify all first-party imports of [TARGET_DEPENDENCY] or [TARGET_DEPENDENCY]/* and determine which ones are exercised by the repository.
2. Establish the current baseline by running the repository's existing validation steps and recording the passing result.
3. Create local helper modules implementing only the required behavior.
4. Replace eligible first-party imports of [TARGET_DEPENDENCY] with local imports.
5. Remove [TARGET_DEPENDENCY] from the repository configuration only if it is no longer required by first-party source, tests, or repository tooling.

Validation steps (must run and report):
- Run the repository's installation command, such as npm install, yarn install, or pnpm install, depending on the repository.
- Run the repository's baseline validation command, such as npm test, yarn test, pnpm test, or the project-specific validation command

```
  identified during baseline preparation.

Acceptance criteria:
- The baseline validation command passes with no new failures relative
  to the baseline.
- No first-party runtime source file imports [TARGET_DEPENDENCY] or
  [TARGET_DEPENDENCY]/*.
- Public behavior remains unchanged for repository functionality
  exercised by the validation artifacts.
- Changes are localized and minimal.

Report:
- Summary of edits
- Exact files changed
- Commands run and their results
- Whether [TARGET_DEPENDENCY] remained in the repository configuration
  and why
- Dependency diff
- Risk / compatibility notes
- Final verdict: safe replacement / not safe replacement
```

4.5 Evaluation Methodology

We evaluate use-case-oriented regeneration along three dimensions corresponding to our research questions: regeneration feasibility, behavioral preservation, and differences between generated local implementations and the original dependency in terms of size and dependency footprint. Together, these dimensions allow us to study not only whether repository-specific regeneration is possible, but also how well it works in practice and how generated implementations compare with the external artifacts they are intended to replace. Table 4.3 summarizes these metrics and how they are measured.

Table 4.3. Metrics used to evaluate use-case-oriented regeneration.

Metric	Intuition	Operationalization
Regeneration feasibility	Whether the agent can produce an integrated local replacement.	For repository r and dependency d, define $F(r,d) = 1$ if the target dependency is removed from first-party runtime imports and a local replacement exists; otherwise $F(r,d) = 0$. The feasibility rate is $FR(d) = \frac{1}{\lvert R_d \rvert} \sum_{r \in R_d} F(r,d)$.
Behavioral preservation	Whether the regenerated repository still behaves correctly under available repository evidence.	The validation pass rate is $PR(d) = \frac{\sum_{r \in R_d} P_a(r,d)}{\sum_{r \in R_d} P_b(r,d)}$, where P_b is the number of baseline validation checks passed and P_a is the number of validation checks passed after regeneration. New validation failures are measured as $F_{\text{new}}(d) = \sum_{r \in R_d} \max(0, P_b(r,d) - P_a(r,d))$, corresponding to baseline-passing validation checks that fail after regeneration.
Focused dependency-specific tests	Whether the generated replacement preserves the behavior of dependency APIs used by the repository.	Focused test preservation is $FPR(r,d) = \frac{FP(r,d)}{FT(r,d)}$, where FP is the number of focused tests passed and FT is the number of focused tests executed. The value $FT(r,d)$ also records the number of focused assertions or checks.
Size reduction	Whether the replacement is narrower than the original dependency.	Byte reduction is $BRed(r,d) = 1 - \frac{B_g(r,d)}{B_o(d)}$, where B_g is generated replacement bytes and B_o is original dependency bytes. LOC reduction is $LRed(r,d) = 1 - \frac{LOC_g(r,d)}{LOC_o(d)}$.
Dependency-footprint reduction	Whether regeneration reduces reliance on external packages.	Direct-dependency reduction is $DRed(r,d) = D_o(d) - D_g(r,d)$, where D_o and D_g are original and generated direct dependency counts. Installed-tree reduction is $TRed(r,d) = 1 - \frac{B_g(r,d)}{T_o(d)}$, where T_o is the installed tree size of the original dependency.
API surface utilization	How narrowly the client repository used the original dependency.	API utilization is $U(r,d) = \frac{A_u(r,d)}{A_e(d)}$, where A_u is the number of API elements used by the repository and A_e is the number of API elements exported by the dependency.

Note. Here, r denotes a repository, d denotes a dependency, and R_d is the set of repositories evaluated for dependency d.

In addition to repository-level validation artifacts, we collected two auxiliary measurements: focused dependency-specific tests and API surface utilization. Focused dependency-specific tests were generated for each repository–dependency pair to exercise the dependency APIs identified as used by that repository. These tests were not the full upstream dependency test suite. Instead, each focused test was a standalone `.test.ts` file, run with Jest or Vitest, targeting the repository-used API functions of the generated replacement. We recorded pass/fail outcomes and assertion counts.

API surface utilization measures the fraction of the original dependency's exported API surface used by a repository. For each dependency, we defined dependency-specific patterns for API calls and import forms, scanned first-party source files while excluding directories such as `node_modules/`, test directories, declaration files, `dist/`, and `build/`, and collected the unique exported API names used by each repository. We then divided the number of unique used APIs by the total exported API count for the corresponding dependency.

4.5.1 RQ1: Regeneration Feasibility

Our first question asks whether an agent can regenerate the exercised functionality of a dependency as a local implementation that preserves repository behavior. We therefore evaluate feasibility at the level of the repository–dependency pair. A regeneration attempt is considered successful when the agent eliminates first-party runtime reliance on the target dependency and the repository's validation outcomes remain unchanged relative to the baseline.

We distinguish several outcomes. A *successful regeneration* eliminates first-party runtime dependence on the target package while matching baseline validation behavior. A *partial regeneration* removes some first-party reliance or preserves only part of the observed behavior, but does not fully satisfy the success criterion. An *unsuccessful regeneration* fails to preserve repository behavior or fails to eliminate the relevant dependency usage. Finally, we separately note *environmentally blocked* cases in which evaluation cannot be completed reliably because of issues such as missing system dependencies, incompatible package-manager requirements, or broken repository infrastructure unrelated to the regeneration attempt.

4.5.2 RQ2: Behavioral Preservation

Our second question asks to what extent generated implementations preserve repository behavior across repository–dependency pairs. We answer this by comparing post-regeneration validation outcomes against the baseline established for each repository before any regeneration attempt was made.

The primary signals are the number of baseline passing validation checks, the number of checks passed after regeneration, and the number of newly failing checks introduced by the generated implementation. In the strongest cases, post-regeneration validation matches the baseline exactly. However, we also distinguish near-matches, partial preservation, and cases in which the generated implementation appears directionally correct but still introduces regressions. This evaluation focuses on repository-level behavior rather than semantic equivalence of the dependency in the abstract. Accordingly, a generated implementation is judged by whether it preserves the behavior that the repository actually exercises under its existing validation artifacts.

4.5.3 RQ3: Differences from the Original Dependency

Our third question asks how generated implementations compare with the original dependency in terms of size and dependency footprint. To answer this, we compare the generated local implementation against the original carried dependency as it existed in the repository before regeneration.

For size, we consider measures such as lines of code, file size, and, where appropriate, bundle size or other proxies for carried functionality. For dependency footprint, we examine whether regeneration reduces or eliminates direct and transitive reliance on the original external package, especially in first-party runtime code.

These comparisons are not intended to establish full semantic superiority of the generated implementation over the original dependency. Rather, they help characterize what kind of artifact the agent produces when regeneration succeeds. In particular, they help us assess whether successful regenerated implementations tend to be narrower, more localized, and less dependency-heavy than the original package as carried by the repository.

4.5.4 Interpretive Scope

These evaluation dimensions are intentionally repository-centered. Passing validation does not guarantee full semantic equivalence, and failures do not always imply that the regeneration strategy is fundamentally unsound. Results depend on the coverage and quality of the repository's existing validation artifacts, as well as on environmental stability during evaluation. Accordingly, our methodology is designed to assess practical repository-level regeneration, not to certify that a generated implementation is universally interchangeable with the original dependency across all possible clients.

4.6 Results

This section presents the results of our use-case-oriented regeneration experiments across the evaluated repository–dependency pairs. We organize the results around the three research questions introduced earlier in the chapter. We first examine whether regeneration is feasible across the benchmark, then evaluate how well generated implementations preserve repository behavior, and finally compare successful implementations against the original dependencies in terms of size and dependency footprint.

4.6.1 RQ1: Regeneration Feasibility

We first examine regeneration feasibility at the repository–dependency-pair level. A regeneration attempt is feasible when the agent can remove first-party runtime reliance on the target dependency, integrate a local replacement, and preserve the repository's baseline validation results. Because this outcome depends on both the target dependency and the way a repository uses it, we begin with representative examples before reporting aggregate results.

Table 4.4. Representative regeneration outcomes for three dependencies each tested against a small, medium, and large repository.

Dependency	Size	Repository	GitHub Repo	Stars	Tests Before	Tests After	Tests Failed
nanoid	Small	schemio	ishubin/schemio	557	252	252	0
	Medium	shortid	dylang/shortid	5,723	18	18	0
	Large	postcss	postcss/postcss	28,975	649	649	0
chalk	Small	generator	yeoman/generator	1,262	335	335	0
	Medium	bower	bower/bower	14,930	445	444	1
	Large	ink	vadimdemedes/ink	37,554	907	907	0
lodash	Small	consolidate.js	tj/consolidate.js	3,475	340	340	0
	Medium	nightwatch	nightwatchjs/nightwatch	11,946	1960	1960	0
	Large	redux-saga	redux-saga/redux-saga	22,478	153	153	0

Use-case-oriented regeneration is feasible across repositories of different sizes. Table 4.4 presents representative outcomes for three dependencies evaluated against small, medium, and large client repositories. These examples illustrate a broader pattern in the benchmark: in many cases, regeneration preserves repository behavior across varied repository settings. Among the cases shown here, only one test case fails after regeneration, while the others preserve all baseline tests. Complete per-dependency results for all evaluated repository–dependency pairs are provided in Appendix A. Additional details on failed regeneration outcomes and their likely root causes are provided in Appendix A.4.

Table 4.5. Summary of regeneration results across all 180 repositories.

Dependency	Repos	Baseline	After	Failed	Pass Rate	Perfect
`axios`	20	7,513	7,509	4	99.9%	18/20
`chalk`	20	4,716	4,715	1	100.0%	19/20
`change-case`	20	20,917	20,880	37	99.8%	19/20
`express`	20	1,170	1,164	6	99.5%	18/20
`lodash`	20	12,804	12,700	104	99.2%	16/20
`nanoid`	20	2,118	2,118	0	100.0%	20/20
`postcss`	20	2,292	2,289	3	99.9%	18/20
`semver`	20	5,111	5,109	2	100.0%	18/20
`zod`	20	7,993	7,993	0	100.0%	20/20
Total	**180**	**64,634**	**64,477**	**157**	**99.8%**	**166/180**

Table 4.5 summarizes aggregate outcomes across all 180 repository–dependency pairs. Overall, regeneration is highly feasible as an integrated replacement strategy in this benchmark, though perfect behavioral preservation is not uniform across all repository–dependency pairs. Across all dependencies, 64,477 of 64,634 baseline checks still pass after regeneration, and 166 of 180 repository–dependency pairs achieve perfect preservation. Several dependencies, including `nanoid` and `zod`, achieve perfect preservation in all 20 client repositories, while others such as `axios`, `chalk`, `express`, `postcss`, and `semver` also perform strongly. `lodash` is more challenging than most of the other dependencies in the benchmark, but still preserves 99.2% of baseline checks and achieves perfect preservation in 16 of 20 repositories.

Taken together, these results indicate that use-case-oriented regeneration is feasible for a substantial fraction of repository–dependency pairs. At the same time, success is not uniform across all dependencies, suggesting that regeneration difficulty depends both on the target dependency and on how the client repository uses it.

4.6.2 RQ2: Behavioral Preservation

We next evaluate whether the generated replacements preserve the dependency-specific behavior exercised by the client repositories. In addition to the repositories' baseline validation artifacts, we use focused dependency-specific tests to measure whether the regenerated implementation preserves the particular APIs and behaviors that were used by each repository. This analysis provides a narrower view of behavioral preservation than full repository validation, but it more directly targets the dependency functionality that regeneration attempted to replace.

Table 4.6. Results on dependency-focused test cases before and after regeneration, including new failed tests, pass rate, and number of repositories with perfect preservation.

Dependency	Repos	Focused Base	Focused After	Failed	Pass Rate	Perfect
`axios`	20	3,387	3,384	3	99.9%	19/20
`chalk`	20	1,964	1,964	0	100.0%	20/20
`change-case`	20	425	425	0	100.0%	20/20
`express`	20	631	627	4	99.4%	19/20
`lodash`	20	3,985	3,962	23	99.4%	17/20
`nanoid`	20	1,288	1,288	0	100.0%	20/20
`postcss`	20	1,427	1,425	2	99.9%	18/20
`semver`	20	2,386	2,385	1	100.0%	19/20
`zod`	20	522	522	0	100.0%	20/20
Total	180	16,015	15,982	33	99.8%	172/180

Generated implementations usually preserve the dependency-focused behavior that client repositories actually exercise. Table 4.6 reports results on dependency-focused tests and shows that, across all repository–dependency pairs, the generated implementations preserve 15,982 of 16,015 focused checks, with 172 of 180 pairs achieving perfect focused-test preser-

vation. This suggests that, in many cases, the generated implementation captures the subset of dependency behavior that the client repository actually exercises.

This result should be interpreted at the repository level rather than as evidence of full semantic equivalence with the original package. Passing the available tests provides evidence that the generated implementation preserves repository-observed behavior, but it does not imply that it would behave identically for all possible downstream clients or all possible uses of the original dependency.

For example, one successful replacement involved `webc`'s use of `nanoid`. The original `nanoid` package provides several entry points and capabilities, including the default `nanoid()` function, custom alphabets, random-byte helpers, browser and non-secure variants, and command-line functionality. However, `webc` only exercised the default identifier-generation behavior. Accordingly, the generated replacement preserved the slice of behavior needed by `webc`, rather than recreating the full upstream package.

Listing 4.1: Example local replacement for the `nanoid` functionality used by `webc`.

```
const urlAlphabet =
  'useandom-26T198340PX75pxJACKVERYMINDBUSHWOLF'
  + '_GQZbfghjklqvwyzrict';

export function nanoid(size = 21) {
  const bytes = new Uint8Array(size);
  crypto.getRandomValues(bytes);

  let id = '';
  for (let i = 0; i < size; i++) {
    id += urlAlphabet[bytes[i] & 63];
  }

  return id;
}
```

The replacement keeps the same core behavior needed by the client repository: it obtains random bytes, bitwise-ANDs each byte with `63`, and maps the result into the same URL-safe alphabet. At the same time, it omits functionality that `webc` did not use, such as `random()`, `customRandom()`, `customAlphabet()`, browser-specific entry points, the non-secure variant, and the command-line interface. It also removes the upstream package's buffer-pooling optimization, instead allocating a fresh `Uint8Array` for each call.

This example illustrates the scope of the equivalence claim in this chapter. For `webc`'s observed use case, the replacement preserves the behavior exercised by the repository's validation artifacts. It is not a drop-in replacement for arbitrary downstream clients of `nanoid`: another project that uses custom alphabets, imports helper functions, or depends on the original package's performance characteristics would require a larger replacement. Thus, the reduction from the original package to the generated local implementation should be understood as regeneration of the repository-used slice, not as a claim that the entire dependency has been reimplemented.

4.6.3 RQ3: Size and Dependency Footprint

Finally, we compare successful generated replacements with the original dependencies they replace. This analysis asks whether regeneration merely recreates a dependency in another location, or whether it produces a narrower repository-specific implementation. We therefore examine size, lines of code, dependency structure, edit scope, and API surface utilization.

Table 4.7. Average original and regenerated implementation sizes, with percentage reductions in size and lines of code across dependencies.

Dependency	Avg Original	Avg Regenerated	Size Reduction	LOC Reduction
`zod`	3,524 KB	26 KB	99.3%	99.4%
`axios`	1,999 KB	15 KB	99.3%	98.1%
`lodash`	1,150 KB	26 KB	97.8%	99.3%
`change-case`	21 KB	0.1 KB	99.6%	97.7%
`nanoid`	26 KB	0.5 KB	98.2%	97.0%
`express`	196 KB	16 KB	91.9%	90.3%
`semver`	89 KB	13 KB	85.8%	82.1%
`chalk`	31 KB	10 KB	69.1%	60.5%
`postcss`	103 KB	56 KB	45.9%	69.6%

Table 4.7 compares the size of successful regenerated implementations against the original dependencies. In most cases, the generated implementations are substantially smaller than the packages they replace. For several dependencies, including `zod`, `axios`, `lodash`, `change-case`, and `nanoid`, average size reduction exceeds 97%, with similarly large reductions in lines of code. Even for dependencies with smaller relative reductions, such as `chalk` and `postcss`, the regenerated implementation remains materially smaller than the original dependency.

This pattern is consistent with the use-case-oriented framing of the task. The goal is not to recreate the original dependency in full, but to regenerate only the subset of functionality that the client repository actually uses.

Table 4.8. Average transitive dependencies, dependency tree size, and number of files changed per repository for each dependency.

Dependency	Avg Transitive Deps	Avg Dep Tree Size	Avg Files Changed Per Repo
`express`	57.0	2,002 KB	5.0
`axios`	20.2	2,595 KB	5.9
`change-case`	8.6	113 KB	2.6
`postcss`	5.1	807 KB	6.0
`chalk`	2.4	55 KB	2.2
`semver`	0.3	134 KB	3.8
`lodash`	0.0	1,150 KB	10.4
`nanoid`	0.0	26 KB	3.0
`zod`	0.0	3,524 KB	23.0
Overall	10.4	1,156 KB	6.9

Regeneration difficulty varies not only with dependency size, but also with dependency structure and integration scope. Table 4.8 provides additional context by summarizing transitive dependency counts, carried dependency-tree size, and the number of files changed per repository during integration. The evaluated dependencies differ substantially along these dimensions. Some dependencies, such as `express` and `axios`, carry relatively large transitive structures, while others such as `nanoid`, `semver`, and `lodash` have few or no transitive dependencies of their own. The average number of files changed per repository also varies substantially, from 2.2 for `chalk` to 23.0 for `zod`.

These differences help explain why regeneration success is uneven across dependencies. Outcomes are shaped not only by the size of the original package, but also by how broadly the repository depends on it and how invasive the resulting integration changes must be.

Table 4.9. Original exported API surface, average regenerated API surface, and resulting reduction and utilization across dependencies.

Dependency	Original Exports	Avg Regenerated	Reduction	Utilization
`zod`	236	4.1	98.3%	1.7%
`lodash`	306	19.4	93.7%	6.3%
`chalk`	64	5.0	92.2%	7.8%
`postcss`	25	1.0	96.0%	4.0%
`change-case`	14	0.4	97.1%	2.9%
`axios`	33	6.4	80.6%	19.4%
`semver`	45	9.8	78.1%	21.9%
`nanoid`	5	1.4	73.0%	27.0%
`express`	11	3.2	70.9%	29.1%
Overall	82.1	5.6	93.1%	6.9%

Generated implementations are not only smaller in size, but also substantially narrower in the interface they expose. Table 4.9 shows that, across dependencies, the generated implementations reduce exposed API surface by an average of 93.1%. Put differently, client repositories typically use only a small fraction of the original dependency interface, and the generated implementations reflect that narrower repository-specific usage. For example, the average regenerated interface exposes only 4.1 exports for `zod` and 19.4 for `lodash`, compared with original exported surfaces of 236 and 306, respectively.

This result supports the central intuition behind use-case-oriented regeneration. In many repository–dependency pairs, the relevant regeneration target is not the full dependency, but the bounded slice of functionality that the repository actually exercises. Taken together, the results in this chapter show that use-case-oriented regeneration is often feasible, usually preserves repository-observed behavior under existing validation artifacts, and frequently yields artifacts that are much smaller and narrower than the original dependencies. These findings do not suggest that generated local implementations should displace conventional

dependency reuse in general. They do, however, suggest that use-case-oriented regeneration is a plausible sourcing strategy for at least some repository–dependency pairs.

4.6.4 Validation Coverage and Behavioral Confidence

Line coverage was measured over the generated replacement code, rather than over the entire client repository. For each repository–dependency pair, we ran the repository's existing validation artifacts with coverage enabled and measured what percentage of the newly generated local replacement code was exercised by those tests. For each dependency, Table 4.10 reports the average generated-code line coverage across the evaluated repositories. Coverage is used only as an interpretive measure of behavioral confidence; it is not part of the regeneration feasibility criterion.

Table 4.10. Average percentage of generated replacement code covered by each repository's existing validation artifacts.

Dependency	Avg Line Coverage
`change-case`	92.5%
`nanoid`	85.7%
`postcss`	72.6%
`axios`	54.0%
`express`	44.1%
`lodash`	40.8%
`chalk`	35.3%
`zod`	28.5%
`semver`	25.5%

Behavioral confidence depends on how strongly the repository validation artifacts exercise the generated replacement code. Table 4.10 reports the average generated-code line coverage across dependencies in our study and shows substantial variation, ranging from over 90% for some dependencies to below 30% for others. This variation is important for interpreting the

results of use-case-oriented regeneration because our evaluation relies on existing repository validation artifacts to assess behavioral preservation. Accordingly, passing tests provides evidence that the generated implementation preserves repository-observed behavior, but it does not guarantee full semantic equivalence with the original dependency.

When generated-code coverage is limited, untested replacement logic, functionality, or edge cases may remain unexercised. As a result, some regeneration outcomes that pass repository validation may overestimate true behavioral equivalence. Conversely, some failures may reflect gaps or brittleness in the validation artifacts rather than fundamental limitations of the regeneration approach. Taken together, these results show that use-case-oriented regeneration should be understood as a repository-level transformation validated against available tests, rather than as a guarantee of full functional replacement of the original dependency.

4.7 Discussion and Limitations

The results of this chapter suggest that use-case-oriented regeneration is often feasible for a substantial subset of repository–dependency pairs. Across the evaluated benchmark, generated implementations frequently preserve repository-observed behavior under existing validation artifacts and tend to be significantly smaller and narrower than the original dependencies they replace. These findings support the view that, in many cases, downstream repositories rely on only a small slice of dependency functionality, and that this slice can be regenerated locally.

At the same time, regeneration success is not uniform across all dependencies. Some dependencies, such as `nanoid` and `zod`, exhibit consistently strong outcomes, while others such as `lodash` are more challenging. This variation reflects differences not only in dependency size, but also in how broadly and deeply repositories rely on them, and in how invasive the resulting integration changes must be. Accordingly, use-case-oriented regeneration should be understood as a conditional sourcing strategy that is more effective when dependency usage is narrow, well-exercised, and structurally localized within the repository.

An important factor in interpreting these results is the quality of the validation artifacts used to assess behavioral preservation. As shown in Table 4.10, test coverage varies substan-

tially across dependencies, ranging from over 90% to below 30%. Because our evaluation relies on existing repository validation artifacts, passing tests provides evidence that generated implementations preserve repository-observed behavior under the exercised inputs, but it does not guarantee full semantic equivalence with the original dependency across all possible inputs or usage contexts. In cases with limited coverage, untested functionality or edge cases may remain unexercised, and successful regeneration outcomes may overestimate true behavioral equivalence.

At the same time, the combination of high regeneration success rates and relatively low coverage in some cases reinforces a central observation of this chapter: repositories often exercise only a small portion of the functionality exposed by their dependencies. This is further supported by the substantial reduction in API surface observed across regenerated implementations, suggesting that the effective dependency interface for a given repository is often much smaller than the full package interface.

Several additional limitations should be considered. First, the study is restricted to a specific set of JavaScript/TypeScript dependencies and client repositories with runnable validation artifacts, and results may not generalize directly to other ecosystems, programming languages, dependency types, or repositories with weaker validation infrastructure. Second, the regeneration process depends on the capabilities of a single agentic workflow, the prompt design, and the available tooling, all of which may influence outcomes. Third, successful regeneration does not establish long-term maintainability, performance characteristics, or security properties of the generated implementations. More broadly, regeneration may change the bug and risk profile of the resulting system in ways that repository-level validation does not fully capture. A generated replacement may plausibly introduce faults through underspecified requirements, missed edge cases, or behavioral differences in infrequently exercised conditions, consistent with broader findings that LLM-generated code and coding-agent behavior can exhibit non-syntactic mistakes and repository-grounded behavioral anomalies [56, 57]. At the same time, it may also remove risks by eliminating unused code paths, narrowing the effective API surface, or omitting problematic behavior present in the original dependency but irrelevant to the client repository, which is closely related to motivations seen in software debloating research [51, 58]. Accordingly, regeneration should

not be understood only in terms of whether behavior is preserved, but also in terms of how defects and risks may be redistributed when a broad upstream artifact is replaced by a narrower local implementation.

Taken together, these findings suggest that use-case-oriented regeneration is best understood as a repository-level transformation validated against available evidence, rather than as a guarantee of full functional replacement of an upstream dependency. While it does not replace conventional dependency reuse in general, it highlights an alternative sourcing strategy that may be viable in settings where dependency usage is narrow, validation artifacts are available, and local control over functionality is desirable.

5. CONCLUSION

5.1 Summary of Findings

This thesis argued that generative systems are beginning to reshape software supply chains in two related ways. First, AI coding agents already participate in software reuse within existing package ecosystems by making dependency decisions in real repositories. Second, increasingly capable large language models and agentic workflows may make use-case-oriented regeneration feasible in some settings, allowing projects to regenerate repository-used slices of dependency functionality locally rather than continuing to rely on a full upstream package. Taken together, these developments suggest that software supply chains should be understood not only in terms of persistent upstream artifacts, but also in terms of software that can increasingly be generated, regenerated, validated, and maintained through generative workflows.

Chapter 3 examined the first and more immediate form of change: how AI coding agents already affect software reuse from within existing software ecosystems. Using agent-authored and human-authored pull requests across 2,807 GitHub repositories spanning seven ecosystems, that chapter showed that dependency modification is a routine part of agent-authored repository work rather than an edge case. Agents do not only generate source code; they also add, remove, and update third-party dependencies as part of completing software engineering tasks. The results further showed that agents perform version updates more often than humans and, conditional on introducing or updating a dependency, select PR-time known-vulnerable versions more frequently. These findings demonstrate that generative systems already influence software supply chains through dependency decisions that matter for security, maintenance, and evaluation, even within today's package-centric model of reuse.

Chapter 4 studied a second and more fundamental possibility: whether generative systems can change the unit of software sourcing itself through use-case-oriented regeneration. Rather than asking whether a dependency can be recreated in full as a general-purpose package, that chapter examined whether the subset of functionality actually exercised by a client repository can be regenerated as a local implementation while preserving repository behavior under repository-level validation. The results suggest that this possibility is best

understood at the level of the repository–dependency pair rather than the package in isolation. In particular, the viability of use-case-oriented regeneration depends on factors such as how narrowly the repository uses the dependency, how well that behavior is captured by existing validation artifacts, and how much of the dependency's broader functionality is actually required in practice. These findings do not imply that whole-package reuse is obsolete. They do suggest, however, that in some settings the traditional assumption that reusable functionality must always be sourced through a full upstream artifact is less complete than it once was.

Taken together, these findings suggest that use-case-oriented regeneration is best understood as a repository-level transformation validated against available evidence, rather than as a guarantee of full functional replacement of an upstream dependency. More broadly, full semantic preservation is not always the right objective. If the original dependency contains bugs, unsafe behavior, or even malicious functionality, then reproducing all of its behavior may be undesirable. In that sense, the practical goal of use-case-oriented regeneration is narrower: to preserve the repository-relevant behavior that the client application depends on, rather than to reproduce the entire upstream artifact in all respects. While it does not replace conventional dependency reuse in general, it highlights an alternative sourcing strategy that may be viable in settings where dependency usage is narrow, validation artifacts are available, and local control over functionality is desirable.

5.2 Future Work

This thesis opens several directions for future work. First, the empirical study of AI coding agents and dependency decision-making can be extended in both breadth and depth. Future research could examine a wider range of agent systems, private repositories, and organizational settings, as well as richer forms of dependency choice beyond manifest edits alone. It would also be valuable to study how agent dependency decisions interact with project policies, maintainer preferences, and deployment constraints that are difficult to recover from pull request data alone.

A particularly important next step is the construction of benchmarks that evaluate dependency decision-making directly rather than treating it as an incidental byproduct of task completion. This thesis and the broader research agenda around it motivate benchmark directions centered on behaviors such as safe version selection, reuse discipline, and restraint against unnecessary dependency additions. Developing those ideas into a broader and more mature benchmark suite would make it possible to evaluate coding agents not only on functional success, but also on whether they make sound dependency choices under realistic security and policy constraints.

Second, the evaluation of use-case-oriented regeneration should be expanded beyond the initial benchmark used in this thesis. Future work could study more ecosystems, more dependency classes, and a larger variety of repository–dependency pairs. It would also be useful to examine regeneration under weaker validation settings, where tests provide only partial behavioral coverage, and under stronger validation settings, where semantic equivalence, performance characteristics, or compatibility guarantees matter in addition to test passing. Such work would help clarify when local regeneration is genuinely viable and when apparent success is mainly an artifact of limited validation.

Third, future work should study the security and maintainability properties of generated local implementations more directly. A locally generated implementation may reduce dependency footprint and avoid carrying unused upstream functionality, but it may also introduce brittle logic, subtle behavioral errors, or new maintenance obligations for downstream developers. It would therefore be valuable to evaluate generated implementations not only for functional preservation, but also for auditability, long-term maintainability, performance, and resistance to malicious or inefficient behaviors. This is especially important if generative systems are to be considered as part of real software sourcing practice rather than only as experimental tools.

Finally, future work should continue to investigate how generative systems alter the trust model of software supply chains. Traditional supply chain security mechanisms such as provenance, signing, and SBOMs were designed primarily for persistent upstream artifacts. If software can increasingly be generated, regenerated, or selectively produced locally, then new assurance mechanisms may be needed to help downstream projects reason about the origin,

validation, and evolution of generated functionality. Understanding how these mechanisms should interact with existing software supply chain practices is an important open problem.

REFERENCES

[1] Tamanna et al., "Research directions in software supply chain security," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 5, Article 146, 2025. DOI: 10.1145/3714464.

[2] B. Gokkaya, L. Aniello, and B. Halak, "Software supply chain: A taxonomy of attacks, mitigations and risk assessment strategies," *Journal of Information Security and Applications*, vol. 97, p. 104 324, 2026. DOI: 10.1016/j.jisa.2025.104324.

[3] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *28th USENIX Security symposium (USENIX security 19)*, 2019, pp. 995–1010.

[4] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proceedings of the 15th international conference on mining software repositories*, 2018, pp. 181–191.

[5] J. Yang, Y. Zhang, Z. Chen, et al., "Swe-agent: Agent-computer interfaces enable automated software engineering," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

[6] Q. Wang et al., *Openhands: An open platform for ai software developers as generalist agents*, https://github.com/OpenHands/OpenHands, 2024.

[7] T. Singla, B. Çakar, P. C. Amusuo, and J. C. Davis, *Towards a benchmark for dependency decision-making*, 2026. arXiv: 2601.00205 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2601.00205.

[8] T. Singla, D. Anandayuvaraj, K. G. Kalu, T. R. Schorlemmer, and J. C. Davis, "An empirical study on using large language models to analyze software supply chain security failures," in *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, ser. SCORED '23, Copenhagen, Denmark: Association for Computing Machinery, 2023, pp. 5–15, ISBN: 9798400702631. DOI: 10.1145/3605770.3625214. [Online]. Available: https://doi.org/10.1145/3605770.3625214.

[9] K. G. Kalu, T. Singla, C. Okafor, S. Torres-Arias, and J. C. Davis, "An industry interview study of software signing for supply chain security," in *2025 Proceedings of the 34th USENIX Security Symposium*, 2025, pp. 81–100, ISBN: 978-1-939133-52-6.

[10] E. Pautsch et al., "Agenthub: A registry for discoverable, verifiable, and reproducible ai agents," *arXiv preprint arXiv:2510.03495*, 2025. [Online]. Available: https://arxiv.org/abs/2510.03495.

[11] D. Anandayuvaraj, T. Singla, Z. A. H. Hammadeh, A. Lund, A. Holloway, and J. C. Davis, "Learning from software failures: A case study at a national space research center," in *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering (ICSE '26)*, Rio de Janeiro, Brazil: ACM, 2026. DOI: 10.1145/3744916.3773149.

[12] P. Amusuo et al., "Ztd_ {java}: Mitigating software supply chain vulnerabilities via zero-trust dependencies," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, IEEE Computer Society, 2025, pp. 685–685.

[13] K. G. Kalu, S. Okorafor, T. Singla, S. Torres-Arias, and J. C. Davis, "Why johnny signs with sigstore: Examining tooling as a factor in software signing adoption in the sigstore ecosystem," *arXiv preprint arXiv:2503.00271*, 2025. [Online]. Available: https://arxiv.org/abs/2503.00271.

[14] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the javascript package ecosystem," in *Proceedings of the 13th international conference on mining software repositories*, 2016, pp. 351–361.

[15] F. Hou and S. Jansen, "A systematic literature review on trust in the software ecosystem," *Empirical Softw. Engg.*, vol. 28, no. 1, Jan. 2023, ISSN: 1382-3256. DOI: 10.1007/s10664-022-10238-y. [Online]. Available: https://doi.org/10.1007/s10664-022-10238-y.

[16] Synopsys Cybersecurity Research Center, "Open source security and risk analysis (OSSRA) report," Synopsys, Inc., Tech. Rep., 2024, Ninth edition.

[17] Black Duck Software, Inc., "Open source security and risk analysis (OSSRA) report," Black Duck, Tech. Rep., 2025, Tenth edition.

[18] B. W. Boehm, "Software engineering economics," *IEEE transactions on Software Engineering*, no. 1, pp. 4–21, 1984.

[19] B. Boehm, C. Abts, and S. Chulani, "Software development cost estimation approaches—a survey," *Annals of software engineering*, vol. 10, no. 1, pp. 177–205, 2000.

[20] B. W. Boehm et al., *Software cost estimation with COCOMO II*. Prentice Hall Press, 2000.

[21] J. E. Gaffney Jr and R. Cruickshank, "A general economics model of software reuse," in *Proceedings of the 14th international conference on Software engineering*, 1992, pp. 327–337.

[22] A. Mili, S. F. Chmiel, R. Gottumukkala, and L. Zhang, "An integrated cost model for software reuse," in *Proceedings of the 22nd international conference on Software engineering*, 2000, pp. 157–166.

[23] T. Oberndorf, L. Brownsword, and C. A. Sledge, "An activity framework for cots-based systems," Tech. Rep., 2000.

[24] S. A. Slaughter, D. E. Harter, and M. S. Krishnan, "Evaluating the cost of software quality," *Communications of the ACM*, vol. 41, no. 8, pp. 67–73, 1998.

[25] S. Hamer, N. Imtiaz, M. Tamanna, P. Shabrina, and L. Williams, "Trusting code in the wild: Exploring contributor reputation measures to review dependencies in the rust ecosystem," *IEEE Transactions on Software Engineering*, 2025.

[26] SLSA Contributors, *Supply-chain levels for software artifacts (slsa) provenance specification*, https://slsa.dev/spec/v1.0/provenance, Accessed: 2026-03-24, 2023.

[27] National Telecommunications and Information Administration, *The minimum elements for a software bill of materials (sbom)*, https://www.ntia.gov/report/2021/minimum-elements-software-bill-materials-sbom, Accessed: 2026-03-24, 2021.

[28] J. Saayman. "Post mortem: Axios npm supply chain compromise." GitHub issue #10636, accessed April 10, 2026. [Online]. Available: https://github.com/axios/axios/issues/10636.

[29] Aqua Security. "Trivy ecosystem supply chain temporarily compromised." GitHub Security Advisory GHSA-69fq-xp46-6x23, accessed April 10, 2026. [Online]. Available: https://github.com/aquasecurity/trivy/security/advisories/GHSA-69fq-xp46-6x23.

[30] npm, Inc. "Kik, left-pad, and npm." npm Blog, accessed April 10, 2026. [Online]. Available: https://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm.

[31] C. E. Jimenez et al., "Swe-bench: Can language models resolve real-world github issues?" *arXiv preprint arXiv:2310.06770*, 2023.

[32] T. Liu, C. Xu, and J. McAuley, *Repobench: Benchmarking repository-level code autocompletion systems*, 2023. arXiv: 2306.03091 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2306.03091.

[33] J. Du et al., *Dependeval: Benchmarking llms for repository dependency understanding*, 2025. arXiv: 2503.06689 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2503.06689.

[34] H. Li, H. Zhang, and A. E. Hassan, *The rise of ai teammates in software engineering (se) 3.0: How autonomous coding agents are reshaping software engineering*, 2025. arXiv: 2507.15003 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2507.15003.

[35] E. Larios Vargas, M. Aniche, C. Treude, M. Bruntink, and G. Gousios, "Selecting third-party libraries: The practitioners' perspective," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 245–256, ISBN: 9781450370431. DOI: 10.1145/3368089.3409711. [Online]. Available: https://doi.org/10.1145/3368089.3409711.

[36] J. Anugerah and E. Martin-Jones, "The surprise of multiple dependency graphs: Dependency resolution is not deterministic.," *Queue*, vol. 23, no. 1, pp. 64–84, 2025.

[37] A. J. Jafari, D. E. Costa, E. Shihab, and R. Abdalkareem, *Dependency update strategies and package characteristics*, 2023. arXiv: 2305.15675 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2305.15675.

[38] B. Rombaut, F. R. Cogo, and A. E. Hassan, *Leveraging the crowd for dependency management: An empirical study on the dependabot compatibility score*, 2024. arXiv: 2403.09012 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2403.09012.

[39] Y. Wang et al., "An empirical study of usages, updates and risks of third-party libraries in java projects," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2020, pp. 35–45.

[40] F. Reyes, B. Baudry, and M. Monperrus, "Breaking-good: Explaining breaking dependency updates with build analysis," in *2024 IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE, 2024, pp. 36–46.

[41] S. Comella-Dorda, J. Dean, G. Lewis, E. J. Morris, T. Oberndorf, and E. Harper, "A process for cots software product evaluation," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-2003-TR-017, 2004. DOI: 10.1184/R1/6571721.v1.

[42] R. He, T. Saarinen, N. Saarinen, A. van Deursen, and M. Mäntylä, "Automating dependency updates in practice: An exploratory study on security vulnerabilities and other dependency update issues," *IEEE Transactions on Software Engineering*, vol. 49, no. 8, pp. 3928–3949, 2023.

[43] N. Vasilakis, A. Benetopoulos, S. Handa, A. Schoen, J. Shen, and M. C. Rinard, "Supply-chain vulnerability elimination via active learning and regeneration," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21, Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 1755–1770, ISBN: 9781450384544. DOI: 10.1145/3460120.3484736. [Online]. Available: https://doi.org/10.1145/3460120.3484736.

[44] E. Lamprou, J. Dai, G. Ntousakis, M. C. Rinard, and N. Vasilakis, *Lexo: Eliminating stealthy supply-chain attacks via llm-assisted program regeneration*, 2025. arXiv: 2510.14522 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2510.14522.

[45] R. Abdalkareem, V. Oda, S. Mujahid, and E. Shihab, "On the impact of using trivial packages: An empirical case study on npm and pypi," *Empirical Software Engineering*, vol. 25, pp. 1168–1204, 2020. DOI: 10.1007/s10664-019-09792-9.

[46] X. Chen, R. Abdalkareem, S. Mujahid, E. Shihab, and X. Xia, "Helping or not helping? why and how trivial packages impact the npm ecosystem," *Empirical Software Engineering*, vol. 26, no. 27, 2021. DOI: 10.1007/s10664-020-09904-w.

[47] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "Why do developers use trivial packages? an empirical case study on npm," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, 2017, pp. 385–395. DOI: 10.1145/3106237.3106267.

[48] I. Malavolta et al., "Javascript dead code identification, elimination, and empirical assessment," *IEEE Transactions on Software Engineering*, vol. 49, no. 7, pp. 3692–3714, Jul. 2023. DOI: 10.1109/TSE.2023.3267848. [Online]. Available: http://dx.doi.org/10.1109/TSE.2023.3267848.

[49] webpack, *Tree shaking*, https://webpack.js.org/guides/tree-shaking/, Accessed 2026-04-22, 2025.

[50] S. E. Ponta, W. Fischer, H. Plate, and A. Sabetta, "The used, the bloated, and the vulnerable: Reducing the attack surface of an industrial application," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2021, pp. 555–558. DOI: 10.1109/ICSME52107.2021.00056.

[51] Y. Liu, D. Tiwari, C. Bogdan, and B. Baudry, "Detecting and removing bloated dependencies in commonjs packages," *Journal of Systems and Software*, vol. 230, p. 112 509, 2025, ISSN: 0164-1212. DOI: 10.1016/j.jss.2025.112509.

[52] The Go Authors, *Go modules reference*, https://go.dev/ref/mod, Accessed: 2026-04-27.

[53] pip developers, *Vendoring policy*, https://pip.pypa.io/en/latest/development/vendoring-policy/, Accessed: 2026-04-27.

[54] Anthropic, *Claude Code Overview*, https://docs.anthropic.com/claude-code, Accessed: 2026-04-27, 2026.

[55] Anthropic, *Introducing Claude Opus 4.6*, https://www.anthropic.com/news/claude-opus-4-6, Accessed: 2026-04-27, Feb. 2026.

[56] Q. Chen, J. Yu, J. Li, J. Deng, J. T. J. Chen, and I. Ahmed, "A deep dive into large language model code generation mistakes: What and why?" *arXiv preprint arXiv:2411.01601*, 2024. [Online]. Available: https://arxiv.org/abs/2411.01601.

[57] W. Dai, M. Openja, H. V. Pham, G. Uddin, J. Yang, and S. Wang, "Abtest: Behavior-driven testing for ai coding agents," *arXiv preprint arXiv:2604.03362*, 2026. [Online]. Available: https://arxiv.org/abs/2604.03362.

[58] A. Quach et al., "Debloating software through piece-wise compilation and loading," in *27th USENIX Security Symposium (USENIX Security '18)*, USENIX Association, 2018, pp. 341–356.

A. EXTENDED REGENERATION RESULTS

This appendix reports per-dependency successful replacement results for the nine dependencies evaluated in chapter 4. Each table lists the repository–dependency pairs for which replacement preserved the repository's validation artifacts. To keep the main chapter focused on aggregate findings and interpretation, these extended per-dependency tables are presented here.

A.1 Small Dependencies

A.1.1 `nanoid`

Table A.1. Repository–dependency pairs for which `nanoid` was successfully replaced while the repository's validation artifacts continued to pass.

Repository	GitHub Repo	Pkg.	Before	After	Failed	%
postcss	postcss/postcss	pnpm	649	649	0	100.0%
webc	11ty/webc	npm	378	378	0	100.0%
warehouse	hexojs/warehouse	npm	295	295	0	100.0%
schemio	ishubin/schemio	npm	252	252	0	100.0%
tuneflow	tuneflow/tuneflow	npm	163	163	0	100.0%
excalidraw-storage-backend	kitsteam/excalidraw-storage-backend	npm	83	83	0	100.0%
lhast	BlackGlory/lhast	yarn	53	53	0	100.0%
algsosh	Llitvinchuk/algsosh	yarn	32	32	0	100.0%
react-markdown-editor-lite	HarryChen0506/react-markdown-editor-lite	pnpm	31	31	0	100.0%
xmindmark	xmindltd/xmindmark	yarn	31	31	0	100.0%
repositype	bit-cmdr/repositype	pnpm	30	30	0	100.0%
gumboard	antiwork/gumboard	npm	29	29	0	100.0%
nanoid-good	y-gagar1n/nanoid-good	npm	20	20	0	100.0%
shortid	dylang/shortid	yarn	18	18	0	100.0%
cpn-pdr-api	CDECatapult/cpn-pdr-api	yarn	16	16	0	100.0%
eslint-plugin-generate-test-id	matzkoh/eslint-plugin-generate-test-id	npm	11	11	0	100.0%
TLD-Save-Editor-3	FINDarkside/TLD-Save-Editor-3	npm	9	9	0	100.0%
TeleOTP	UselessStudio/TeleOTP	npm	6	6	0	100.0%
mock-nanoid	vikr01/mock-nanoid	npm	6	6	0	100.0%
sortable-nanoid-ts	jibuji/sortable-nanoid-ts	npm	6	6	0	100.0%
Total			**2118**	**2118**	**0**	**100.0%**

A.1.2 `change-case`

Table A.2. Repository–dependency pairs for which `change-case` was successfully replaced while the repository's validation artifacts continued to pass.

Repository	GitHub Repo	Pkg.	Before	After	Failed	%
eslint-stylistic	eslint-stylistic/eslint-stylistic	pnpm	18285	18285	0	100.0%
cropperjs	fengyuanchen/cropperjs	npm	236	236	0	100.0%
react-native-ui-kitten	akveo/react-native-ui-kitten	npm	569	569	0	100.0%
braid-design-system	seek-oss/braid-design-system	pnpm	596	559	37	93.8%
eslint-plugin-import-lite	9romise/eslint-plugin-import-lite	pnpm	338	338	0	100.0%
hydrate-mongodb	artifacthealth/hydrate-mongodb	npm	289	289	0	100.0%
eslint-doc-generator	eslint-community/eslint-doc-generator	npm	280	280	0	100.0%
tablemark	haltcase/tablemark	pnpm	75	75	0	100.0%
eventfan	skyhookadventure/eventfan	npm	65	65	0	100.0%
insomnia-mockbin	Kong/insomnia-mockbin	npm	52	52	0	100.0%
vue-number-input	fengyuanchen/vue-number-input	npm	50	50	0	100.0%
node-madgex	guidesmiths/node-madgex	npm	38	38	0	100.0%
safe-change-case	argodevops/safe-change-case	npm	15	15	0	100.0%
node-merge-config	Telefonica/node-merge-config	npm	12	12	0	100.0%
nuxt-custom-elements	GrabarzUndPartner/nuxt-custom-elements	npm	8	8	0	100.0%
carbon-framework	carbonjs/carbon-framework	npm	5	5	0	100.0%
game-boy-tile-tools	Calvin-LL/game-boy-tile-tools	npm	1	1	0	100.0%
generator-node-cli	olohmann/generator-node-commandline	npm	1	1	0	100.0%
omnilogic-pool-rest-api	AnsonT/omnilogic-pool-rest-api	npm	1	1	0	100.0%
hotcakes-toc	crealgo/hotcakes-rehype-table-of-contents	pnpm	1	1	0	100.0%
Total			**20917**	**20880**	**37**	**99.8%**

A.1.3 `chalk`

Table A.3. Repository–dependency pairs for which `chalk` was successfully replaced while the repository's validation artifacts continued to pass.

Repository	GitHub Repo	Pkg.	Before	After	Failed	%
danger-js	danger/danger-js	yarn	681	681	0	100.0%
concurrently	open-cli-tools/concurrently	npm	581	581	0	100.0%
ink	vadimdemedes/ink	npm	907	907	0	100.0%
bower	bower/bower	npm	445	444	1	99.8%
np	sindresorhus/np	npm	449	449	0	100.0%
generator	yeoman/generator	npm	335	335	0	100.0%
react-boilerplate	react-boilerplate/react-boilerplate	npm	293	293	0	100.0%
Caporal.js	mattallty/Caporal.js	npm	220	220	0	100.0%
clasp	google/clasp	npm	197	197	0	100.0%
npm-run-all	mysticatea/npm-run-all	npm	230	230	0	100.0%
ora	sindresorhus/ora	npm	87	87	0	100.0%
boxen	sindresorhus/boxen	npm	84	84	0	100.0%
standard-version	conventional-changelog/standard-version	npm	53	53	0	100.0%
yo	yeoman/yo	npm	37	37	0	100.0%
hygen	jondot/hygen	yarn	36	36	0	100.0%
serve	vercel/serve	npm	28	28	0	100.0%
nexe	nexe/nexe	npm	22	22	0	100.0%
update-notifier	sindresorhus/update-notifier	npm	15	15	0	100.0%
npm-name-cli	sindresorhus/npm-name-cli	npm	9	9	0	100.0%
fkill-cli	sindresorhus/fkill-cli	npm	7	7	0	100.0%
Total			**4716**	**4715**	**1**	**100.0%**

A.2 Medium Dependencies

A.2.1 `express`

Table A.4. Repository–dependency pairs for which `express` was successfully replaced while the repository's validation artifacts continued to pass.

Repository	GitHub Repo	Pkg.	Before	After	Failed	%
alfa-leetcode-api	alfaarghya/alfa-leetcode-api	npm	236	236	0	100.0%
graphql2rest	sisense/graphql2rest	npm	105	105	0	100.0%
node-template-server	baguilar6174/node-template-server	yarn	58	53	5	91.4%
calendar-json-api	Gonzagadavid/calendar-json-api	npm	40	40	0	100.0%
node-multiplayer-snake	simondiep/node-multiplayer-snake	npm	35	35	0	100.0%
node-boilerplate	santoshshinde2012/node-boilerplate	npm	30	30	0	100.0%
js-unit-testing-examples	MarcL/js-unit-testing-examples	npm	25	25	0	100.0%
express-app-testing-demo	gregjopa/express-app-testing-demo	npm	23	23	0	100.0%
express-typescript	edwinhern/express-typescript	npm	21	21	0	100.0%
json-caching-proxy	sonyseng/json-caching-proxy	npm	18	18	0	100.0%
dialetus-service	mvfsillva/dialetus-service	npm	16	16	0	100.0%
graphql-rest-proxy	acro5piano/graphql-rest-proxy	npm	14	14	0	100.0%
clamav-rest-api	benzino77/clamav-rest-api	npm	16	16	0	100.0%
timings	godaddy/timings	npm	10	10	0	100.0%
Express-REST-API-Template	rzgry/Express-REST-API-Template	npm	4	4	0	100.0%
express-babel	vmasto/express-babel	npm	4	4	0	100.0%
mostly	justinsisley/mostly	npm	3	3	0	100.0%
github-trending-api	huchenme/github-trending-api	npm	2	2	0	100.0%
gamevault-backend	Phalcode/gamevault-backend	pnpm	380	379	1	99.7%
Echo-Server	Ealenn/Echo-Server	npm	130	130	0	100.0%
Total			**1170**	**1164**	**6**	**99.5%**

A.2.2 `postcss`

Table A.5. Repository–dependency pairs for which `postcss` was successfully replaced while the repository's validation artifacts continued to pass.

Repository	GitHub Repo	Pkg.	Before	After	Failed	%
stylelint-order	hudochenkov/stylelint-order	npm	517	517	0	100.0%
rtlcss	MohammadYounes/rtlcss	npm	416	416	0	100.0%
postcss-modules-local-by-default	css-modules/postcss-modules-local-by-default	npm	249	249	0	100.0%
stylelint-processor-styled-components	styled-components/stylelint-processor-styled-components	npm	230	230	0	100.0%
postcss-clean	leodido/postcss-clean	yarn	164	164	0	100.0%
uncss	uncss/uncss	npm	121	121	0	100.0%
postcss-sorting	hudochenkov/postcss-sorting	npm	105	105	0	100.0%
postcss-conditionals	andyjansson/postcss-conditionals	npm	83	83	0	100.0%
inline-critical	bezoerb/inline-critical	npm	68	66	2	97.1%
postcss-bem-linter	davidtheclark/postcss-bem-linter	npm	67	67	0	100.0%
postcss-icss-selectors	css-modules/postcss-icss-selectors	npm	57	57	0	100.0%
postcss-flexbugs-fixes	luisrudge/postcss-flexbugs-fixes	npm	42	42	0	100.0%
postcss-copy	geut/postcss-copy	npm	33	33	0	100.0%
typed-css-modules	Quramy/typed-css-modules	npm	31	30	1	96.8%
postcss-color-function	postcss/postcss-color-function	npm	25	25	0	100.0%
postcss-selector-matches	postcss/postcss-selector-matches	npm	24	24	0	100.0%
stylelint-no-unsupported-browser-features	ismay/stylelint-no-unsupported-browser-features	pnpm	21	21	0	100.0%
postcss-icss-values	css-modules/postcss-icss-values	npm	18	18	0	100.0%
stylelint-selector-bem-pattern	simonsmith/stylelint-selector-bem-pattern	yarn	11	11	0	100.0%
doiuse	anandthakker/doiuse	npm	10	10	0	100.0%
Total			**2292**	**2289**	**3**	**99.9%**

A.2.3 `semver`

Table A.6. Repository–dependency pairs for which `semver` was successfully replaced while the repository's validation artifacts continued to pass.

Repository	GitHub Repo	Pkg.	Before	After	Failed	%
fastify	fastify/fastify	npm	2204	2203	1	100.0%
release-please	googleapis/release-please	npm	1082	1082	0	100.0%
oclif-core	oclif/core	npm	730	730	0	100.0%
release-it	release-it/release-it	npm	260	260	0	100.0%
semantic-release	semantic-release/semantic-release	npm	249	249	0	100.0%
commit-and-tag-version	absolute-version/commit-and-tag-version	npm	138	138	0	100.0%
electron-packager	electron/electron-packager	npm	118	118	0	100.0%
fiddle-core	electron/fiddle-core	yarn	76	76	0	100.0%
normalize-package-data	npm/normalize-package-data	npm	68	68	0	100.0%
standard-version	conventional-changelog/standard-version	npm	53	53	0	100.0%
node-git-describe	tvdstaaij/node-git-describe	npm	25	24	1	96.0%
bumpp	antfu/bumpp	pnpm	18	18	0	100.0%
npm-pick-manifest	zkat/npm-pick-manifest	npm	17	17	0	100.0%
gulp-bump	stephenlacy/gulp-bump	npm	16	16	0	100.0%
init-package-json	npm/init-package-json	npm	15	15	0	100.0%
release-kit-semver	release-kit/semver	pnpm	12	12	0	100.0%
npmvet	harksys/npmvet	npm	11	11	0	100.0%
npm-package-arg	npm/npm-package-arg	npm	9	9	0	100.0%
semantic-release-helm	nflaig/semantic-release-helm	yarn	6	6	0	100.0%
to-semver	sindresorhus/to-semver	npm	4	4	0	100.0%
Total			**5111**	**5109**	**2**	**100.0%**

A.3 Large Dependencies

A.3.1 `axios`

Table A.7. Repository–dependency pairs for which `axios` was successfully replaced while the repository's validation artifacts continued to pass.

Repository	GitHub Repo	Pkg.	Before	After	Failed	%
pay-selfservice	alphagov/pay-selfservice	npm	2173	2173	0	100.0%
ebay-mcp	YosefHayim/ebay-mcp	npm	981	981	0	100.0%
coinbase-sdk-nodejs	coinbase/coinbase-sdk-nodejs	npm	898	898	0	100.0%
crowdin-api-client-js	crowdin/crowdin-api-client-js	npm	501	501	0	100.0%
twilio-node	twilio/twilio-node	npm	429	428	1	99.8%
alchemy-sdk-js	alchemyplatform/alchemy-sdk-js	npm	406	406	0	100.0%
mailgun.js	mailgun/mailgun.js	npm	380	380	0	100.0%
ckan-mcp-server	ondata/ckan-mcp-server	npm	327	327	0	100.0%
apify-client-js	apify/apify-client-js	npm	311	311	0	100.0%
claude-hub	claude-did-this/claude-hub	npm	272	272	0	100.0%
nutrient-dws-client-typescript	PSPDFKit-labs/nutrient-dws-client-typescript	npm	266	266	0	100.0%
node-tado-client	mattdavis90/node-tado-client	npm	120	120	0	100.0%
contentful.js	contentful/contentful.js	npm	265	265	0	100.0%
anx-api	appnexus/anx-api	npm	104	104	0	100.0%
herocast	hero-org/herocast	pnpm	31	31	0	100.0%
client-nodejs	pipedrive/client-nodejs	npm	25	25	0	100.0%
svg2jsx	balajmarius/svg2jsx	npm	9	9	0	100.0%
buymeacoffee.js	warengonzaga/buymeacoffee.js	npm	7	7	0	100.0%
shortcut-client-js	useshortcut/shortcut-client-js	npm	6	3	3	50.0%
clash-royale-api	AndreVarandas/clash-royale-api	npm	2	2	0	100.0%
Total			**7513**	**7509**	**4**	**99.9%**

A.3.2 `lodash`

Table A.8. Repository–dependency pairs for which `lodash` was successfully replaced while the repository's validation artifacts continued to pass.

Repository	GitHub Repo	Pkg.	Before	After	Failed	%
moleculer	moleculerjs/moleculer	npm	2334	2331	3	99.9%
nightwatch	nightwatchjs/nightwatch	npm	1960	1960	0	100.0%
jshint	jshint/jshint	npm	1656	1656	0	100.0%
he	mathiasbynens/he	npm	1597	1597	0	100.0%
eslint-plugin-flowtype	gajus/eslint-plugin-flowtype	npm	1548	1548	0	100.0%
karma	karma-runner/karma	npm	588	588	0	100.0%
react-styleguidist	styleguidist/react-styleguidist	npm	571	570	1	99.8%
xlsx-populate	dtjohnson/xlsx-populate	npm	523	523	0	100.0%
consolidate.js	tj/consolidate.js	npm	340	340	0	100.0%
nodeshift	nodeshift/nodeshift	npm	279	279	0	100.0%
express-validator	express-validator/express-validator	npm	311	311	0	100.0%
node-restify	restify/node-restify	npm	190	190	0	100.0%
redux-saga	redux-saga/redux-saga	yarn	153	153	0	100.0%
critical	addyosmani/critical	npm	155	155	0	100.0%
react-color	casesandberg/react-color	npm	121	121	0	100.0%
uncss	uncss/uncss	npm	121	121	0	100.0%
documentation	documentationjs/documentation	npm	192	100	92	52.1%
keystone-classic	keystonejs/keystone-classic	npm	83	83	0	100.0%
eslint-plugin-sql	gajus/eslint-plugin-sql	npm	43	43	0	100.0%
yo	yeoman/yo	npm	39	31	8	79.5%
Total			**12804**	**12700**	**104**	**99.2%**

A.3.3 `zod`

Table A.9. Repository–dependency pairs for which `zod` was successfully replaced while the repository's validation artifacts continued to pass.

Repository	GitHub Repo	Pkg.	Before	After	Failed	%
Chorus	Chorus-AIDLC/Chorus	pnpm	1268	1268	0	100.0%
prism-mcp	dcostenco/prism-mcp	npm	1025	1025	0	100.0%
astro-editor	dannysmith/astro-editor	npm	703	703	0	100.0%
adcp	adcontextprotocol/adcp	npm	1925	1925	0	100.0%
Git-Mastery	MikaStiebitz/Git-Mastery	npm	483	483	0	100.0%
salazar	shawnpetros/salazar	npm	385	385	0	100.0%
termcanvas	blueberrycongee/termcanvas	npm	370	370	0	100.0%
Questarr	Doezer/Questarr	npm	358	358	0	100.0%
OpenReels	tsensei/OpenReels	npm	302	302	0	100.0%
mini-kode	minmaxflow/mini-kode	npm	260	260	0	100.0%
openrouter-sdk	OpenRouterTeam/typescript-sdk	npm	186	186	0	100.0%
up-fetch	L-Blondy/up-fetch	npm	156	156	0	100.0%
thinkex	ThinkEx-OSS/thinkex	pnpm	120	120	0	100.0%
skir	gepheum/skir	npm	112	112	0	100.0%
appium-mcp	appium/appium-mcp	npm	106	106	0	100.0%
parsertime	luxdotdev/parsertime	pnpm	88	88	0	100.0%
llm-x	mrdjohnson/llm-x	yarn	67	67	0	100.0%
navidash	wtfllix/navidash	npm	28	28	0	100.0%
ai-reviewer	presubmit/ai-reviewer	npm	27	27	0	100.0%
currencyinfo	Adamant-im/currencyinfo	pnpm	24	24	0	100.0%
Total			**7993**	**7993**	**0**	**100.0%**

A.4 Failed Regeneration Outcomes

Although most repository–dependency pairs preserved repository-observed behavior after regeneration, a small number of attempts introduced new validation failures. Table A.10 summarizes the failed dependent repositories and the apparent causes of failure. These failures are useful for interpreting the limits of use-case-oriented regeneration because they show where repository-used behavior extended beyond the generated replacement's implemented subset.

Table A.10. Failed regeneration outcomes and likely root causes.

Dependency	Dependent repository	Failure mode	Likely root cause
`semver`	`fastify`	Partial specification implementation	Prerelease range behavior was not fully reproduced. The local implementation treated some prerelease comparisons numerically, while npm `semver` rejects prereleases for ranges that do not explicitly include prerelease versions.
`semver`	`node-git-describe`	Class identity mismatch	A test used an `instanceof` check against the exported `semver` class. The generated shim and the value under test had different prototype identities, causing the check to fail.
`semver`	`semantic-release`	Partial specification implementation	Failures were caused by range-handling edge cases, especially prerelease behavior, similar to the `fastify` case.
`lodash`	`express-validator`	Missing chained API support	The replacement did not implement `_.chain()` and wrapper behavior. The dependent uses chained lodash calls, so chained map, filter, and value operations were not preserved.
`lodash`	`moleculer`	Long-tail object semantics	Failures involved edge cases in deep cloning, equality, and merging. The replacement handled ordinary arrays and objects but did not fully match lodash behavior for buffers, typed arrays, symbols, regular expressions, or circular structures.
`lodash`	`xlsx-populate`	Error-object detection edge case	The dependent relies on lodash-style error detection for Excel formula errors. The replacement used a simpler error check, which did not match all error-object cases handled by lodash.
`lodash`	`react-styleguidist`	Snapshot-sensitive output differences	Small differences in object filtering, path parsing, or output ordering changed serialized output and caused snapshot tests to fail.
`lodash`	`keystone-classic`	Mixed lodash edge cases	Failures reflected a combination of missing chained API behavior and deep object semantics.
`postcss`	`rtlcss`	Missing optional feature	The replacement supported parsing and stringification but did not produce source-map data. The dependent expected source-map generation.
`axios`	`twilio-node`	Adapter compatibility issue	The failure involved a test using request interception. The generated replacement used a different request path from real axios, so the mock interceptor did not observe the request.
`express`	`node-template-server`	Middleware semantics mismatch	The replacement did not fully reproduce subtle Express middleware behavior, such as error-handler arity, route skipping, or router option handling.
`express`	`gamevault-backend`	Framework integration mismatch	The dependent uses NestJS on top of Express. The failure likely reflects deeper framework expectations about Express internals, such as request fields, application fields, or router stack structure.
`zod`	`adcp`	Cross-package class identity mismatch	The dependent and one of its libraries used different Zod class identities. A schema constructed using the generated replacement failed `instanceof` checks inside a library that used its own bundled Zod copy.