O'REILLY®
Report

SECOND EDITION

Automated Code Remediation at Scale

The Role of AI in Application Modernization and Security

Pat Johnson, Olga Kundzich & Jonathan Schneider

moderne.ai

Automated Code Remediation at Scale

The Role of AI in Application Modernization and Security

Pat Johnson, Olga Kundzich, and Jonathan Schneider

O'REILLY®

Automated Code Remediation at Scale
by Pat Johnson, Olga Kundzich, and Jonathan Schneider

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Louise Corrigan
Development Editor: Melissa Potter
Production Editor: Ashley Stussy
Copyeditor: nSight, Inc.

Interior Designer: David Futato
Cover Designer: Ellie Volckhausen
Illustrator: Kate Dullea

May 2023: First Edition
May 2025: Second Edition

Revision History for the Second Edition
2025-05-05: First Release

This work is part of a collaboration between O'Reilly and Moderne. See our statement of editorial independence.

979-8-341-62760-4

[LSI]

Table of Contents

CHAPTER 1

The Software Industrial Revolution Has Arrived

We asked developers from a wide range of organizations and industries a simple question: "How long will your application continue to work if you are not allowed to touch its code?" The answer was striking—the more modern the application, the sooner it would be in jeopardy.

Why? Software used to be built from scratch. Today, it is assembled from off-the-shelf components like open source software (OSS) and third-party APIs. Unlike physical components that remain static once integrated, software components evolve independently at their own rate. Enterprises have no control over this evolution. And what happens when you fail to keep up?

Software becomes less secure, gets harder to maintain, and eventually stops working.

And now, with the surge in AI-assisted development, code is being generated faster than ever before. Every suggestion from an AI assistant becomes another line of code to maintain. New features arrive faster—but so does entropy. As a result, development teams are facing an unprecedented scale of technical debt, often through no fault of their own.

Development teams have relied on code remediation and refactoring to chip away at technical debt—whether it's fixing security vulnerabilities, migrating frameworks, updating dependencies, or

improving code quality. This is tedious and error-prone work that requires line-by-line, repository-by-repository changes. It consumes valuable time that could be spent on business-critical initiatives, which means this work is often deferred.

Now, let's imagine a world where code remediation is automated across your entire codebase. It's a world where eliminating technical debt is no longer a time-consuming chore but a continuous, automated process. Your teams could address vulnerabilities more quickly and holistically, and developers could focus on the work that truly drives business value. Let's find out how you get there from here.

In this chapter, we'll explore the growing complexity of modern software, emerging trends in managing technical debt—and why mass-scale automated code remediation is required.

The Assembled Software Supply Chain

We are in the midst of the software industrial revolution, with more and more software rapidly assembled from third-party components—commonly known as the software supply chain. Custom software is integrated with components provided by vendors, cloud providers, and OSS, with as much as 90% of code coming from such dependencies.

In a recent Sonatype industry report,[1] it is estimated that there are 180 third-party components on average per each application. This can include everything that goes into or touches software as it is being produced: developer tools (IDEs, CI/CD), third-party software (dependencies), language runtimes and frameworks, and monitoring and testing frameworks. It allows us to build applications much more quickly, but it also makes code maintenance and security much more difficult as it is no longer under our control.

These composed applications have a life of their own. Third-party dependencies change and evolve at their own pace. Anyone at any time can unintentionally introduce software vulnerabilities. Developers add, deprecate, and delete APIs all the time. For example, a third-party vendor or OSS maintainer can make a change to their

1 Sonatype team, *10th Annual State of the Software Supply Chain*, Sonatype, 2025.

API, and then your organization is on the hook to update your applications before they break.

Additionally, these assembled applications create a larger attack surface, with many vulnerabilities lying dormant until they are exploited. The same industry report cites a 156% year-over-year increase of malicious packages. Additional data from a Veracode report[2] analyzing 1.3 million unique applications shows that 64% of those apps had flaws in first-party code and 70% had flaws in third-party code.

The complexity of the software supply chain requires development teams to be ever vigilant and constantly update their code to keep it all secure and working. However, it's an impossible task. There are too many dependencies, too many repositories, and too many vulnerabilities. Manual developer labor, even with AI assistance, simply cannot keep up.

A Note About Open Source Software Maintainers

The majority of OSS maintainers are dedicated to their communities. When they discover a vulnerability, they typically issue a patch release that fixes the issue without introducing other changes. However, keeping up with these patch updates remains a daunting task for consumers, given the sheer volume of dependencies in modern software.

Maintainers can realistically only support a few back versions of their libraries. When they discover a vulnerability in an unsupported version, it becomes the enterprise organization's responsibility to undertake a major migration to a supported version.

Automated remediation offers a solution, enabling maintainers and vendors to help their users continuously migrate to new versions with minimal friction.

2 Niels Tanis et al., *2025 State of Software Security*, Veracode, 2025.

How Much Code Are We Really Talking About?

The growth in code is explosive as companies face pressure to deliver value to their customers at an increasingly rapid pace. It's not uncommon for large organizations to have billions of lines of proprietary source code. Even small organizations can have 20 million lines of their own code. It really doesn't take very long to accrue, especially with the help of AI assistants. A company we work with has gone from 40 to 1,200 source code repositories in the last eight years.

One financial organization in particular has a comparatively small codebase, totaling 500 million lines of proprietary source code. This is still a mind-boggling number. To put this in perspective, consider an O'Reilly book, with roughly 60 lines to a page. If we were to simply print the 500 million lines of source code in paperbacks and stack them end to end, they would stretch for 75 miles. The amazing thing is that the lion's share of this code—the third-party assembled code—is not even included in this count. Including third-party software would extend this line of books from Miami to Montréal.

Many organizations are building these massive codebases with little to no increase in engineering head count and instead are aided by the sophistication of frameworks, third-party libraries, IDEs, and now AI. In fact, it's never been easier for developers to create new code, with one study indicating that 61.8% of developers say they use AI tools in the development process. Another report[3] cites that GitHub Copilot increased the speed of task completion by 55%.

It's no wonder that the thought of maintaining these massive (and growing) codebases can feel so intractable!

DevSecOps Shifts Security Burden onto Every Developer

DevSecOps has pushed more security responsibility onto developers—tasking them not just with building features but also with addressing vulnerabilities. But in practice, most developer time goes toward preventing vulnerabilities and bugs from getting introduced

3 Eirini Kalliamvakou, "Research: Quantifying GitHub Copilot's Impact on Developer Productivity and Happiness," *GitHub* (blog), updated May 21, 2024.

to new or modified code. Older, untouched code is often overlooked until a vulnerability is discovered—and even then, only the most critical defects in the most critical apps may be prioritized.

The scale of the problem is significant. The Veracode report found that nearly half of the applications studied contained OWASP Top Ten flaws, and over 56% had high-severity vulnerabilities. Many of these go unpatched, especially in third-party dependencies, turning into security debt that quietly accumulates. For example, the Sonatype report shows that three years after the Log4Shell exposure, 13% of Log4j downloads remain vulnerable. Even dormant applications can pose risk.

While static application security testing (SAST) and software composition analysis (SCA) tools can help surface issues across codebases, the burden of remediation still falls on developers, which can be a complex, time-consuming, and labor-intensive process.

When faced with a vulnerability, developers must first determine if their codebase is affected. This involves tracing repositories, identifying vulnerable code paths, and assessing the issue's scope—all of which require a deep understanding of the code and its dependencies. The process can require detailed project management to coordinate between security and often several development teams, with back-and-forth communication to prioritize risks, refine strategies, and manage timelines. The overhead of managing a mass-scale remediation project can quickly slow and even stall upgrade projects.

The reality is that only the most severe, exploitable vulnerabilities get immediate attention, while others contribute to accumulating security debt and leave organizations exposed to risk. It's like having 200 holes in your raft versus 2,000. You're still sinking.

Can we do more to relieve developers of security friction? Instead of reacting to vulnerabilities one by one, imagine a codebase that's secure by default. By consistently updating libraries and dependencies, teams can avoid whack-a-mole patching and stay ahead of emerging threats. Automating these upgrades at scale keeps software on the latest, safest versions—proactively minimizing risk and creating a resilient, forward-looking security posture.

Migration Engineering: A Required Discipline for Modern Software

Code migration is labor-intensive, chaotic, and often clerical—even with AI tools in the mix. While development teams can address many vulnerabilities by upgrading dependency versions, others require direct changes to the application source code. Dependency upgrades themselves can lead to broken code, introducing new complexities. Some fixes are straightforward, like changing an API signature. Others require coordinated, large-scale refactoring across dozens or thousands of repositories.

And here's the truism: the longer you put off migrations, the more complex—and painful—they become.

Migration engineering is a relatively new but critical discipline focused on the complex work of coordinating and updating software across large codebases. The goal is not only to harness new features and improve security but also to avoid software obsolescence altogether. As businesses strive to adapt quickly to new technologies and avoid lock-in, migration engineering plays a pivotal role.

Migrations can cover a broad range of technology transformations, including:

Framework upgrades
: Moving to modern frameworks (e.g., Java 8 to Java 24)

Tech consolidation
: Standardizing on a single technology stack across an organization, often after mergers or acquisitions

Cloud migrations
: Moving to cloud-native technologies from on-prem (e.g., Sybase to PostgreSQL)

Dependency updates
: Keeping up with version changes to maintain support and security compliance (no more reliance on end-of-life components)

These are often risky, error-prone endeavors that require complex coordination across many repositories—with potential for

production outages and unexpected costs. As a result, organizations tend to delay them until they're absolutely necessary, like when a critical vulnerability is discovered and remediation becomes a race against time.

To make matters worse, migrations and dependency updates are typically deprioritized as technical debt—left for "when there's time." One of our customers calls this the "tyranny of agile," which does not allow sufficient space for tech debt remediation. Take, for example, a financial services company managing over 20,000 applications built on a variation of Spring Boot. When a breaking change is introduced in the underlying Spring Framework, that potentially means changes to 20,000 repositories. This kind of effort is not just daunting—it feels impossible.

To meet the speed and scale of modern software, migration engineering must evolve—away from manual, brittle projects and toward automation-led discipline. Platform teams and developer productivity groups are well suited to lead this work, enabling safe, systematic change. Their job isn't to judge existing code but to support the developers responsible for it—with empathy and automation. After all, the current state has built up over decades in many cases. Current developers on a project often aren't the originating developers. That doesn't make them any less responsible for its quality but suggests that they are best approached with a heightened empathy to their predicament.

Ultimately, the goal of migration engineering is not merely to keep the lights on but to enable organizations to adapt quickly, reduce risk, and provide the modern foundation for developers to drive the business forward.

The Business Problem of Tech Debt

Technical debt from third-party dependencies—and now from the explosion of AI-generated code—is becoming a serious business risk. Security vulnerabilities, outdated libraries, and rising maintenance costs are compounding daily.

Manual remediation doesn't scale, and AI assistants weren't designed for this kind of work. Even with alerts and suggestions, teams are left doing repetitive, error-prone tasks across thousands of repositories—one repo at a time.

The only way forward is automation at scale. By codifying fixes and applying them across entire codebases, organizations can reduce risk, stay current, and let developers focus on innovation. As Jason Simpson, VP of Engineering at Choice Hotels International, put it: "As we continue to upgrade these libraries and get to the latest and the fully patched versions, we're much, much safer."

You might be wondering—won't AI just fix all of this soon? It's a fair question, and one we hear often. But the solution isn't another year of model training. As you'll see in the chapters ahead, what AI is missing isn't capability—it's context. To remediate and modernize large codebases, AI needs rich data about your entire codebase.

Keep reading. In the following chapters, we'll explore how automation and AI can transform your approach to technical debt and software modernization.

CHAPTER 2

Leveling Up Your Automated Code Remediation Journey

Developers can now generate new code at unprecedented speed, but with that velocity comes an even greater burden: maintaining, securing, and modernizing your vast, interconnected codebases. Identifying issues is no longer the hard part. Resolving them at scale is.

Refactoring or modernizing across these systems is not a simple grep-and-replace operation—it's a multirepository, multiteam, and multisystem challenge involving millions to billions of lines of code. This is the reality for many enterprise organizations today. Attempting to remediate at this scale using traditional manual approaches and even with AI assistance is not only time-consuming and error-prone—it's not scalable.

In this chapter, we'll explore why today's tools can't scale, what a scalable remediation system requires, and how organizations can build confidence in mass-scale change through automation—moving from manual firefighting to continuous modernization.

Too Many Alerts, Not Enough Action

Scanning and search tools were once the backbone of software maintenance—helping teams surface vulnerabilities, outdated dependencies, and code quality issues. But in today's world of fast-growing, interconnected codebases, these tools are no longer

enough. They reveal problems, but the burden of fixing them still falls to overextended development teams.

SAST and SCA tools highlight vulnerabilities but leave developers to investigate and fix each issue manually. CI/CD gates block code from shipping if it doesn't meet quality standards, but they don't help teams fix the problems blocking delivery. Dependency bots flood repositories with pull requests (PRs) for version bumps—many of which break builds or require additional code changes to be viable. Code search tools help developers locate problems, but they stop short of transforming code, relying on manual scripts and human judgment.

We're seeing a shift: visibility without action no longer creates value—it slows progress and compounds risk. Code search may assess impact, but when a breaking change hits 20,000 repositories, what's the plan? Without scalable remediation, change becomes paralyzing. Technical debt piles up, decisions ossify, and the organization slows down.

But here's the key insight: if you can remediate, you can search. Automated remediation requires the ability to find, understand, and transform code—so search becomes a byproduct, no longer a standalone burden. The result? Developers spend less time firefighting and more time building. Organizations move faster—with greater confidence, consistency, and control.

Role of AI Assistants in Code Authorship and Remediation

AI developer assistants like GitHub Copilot have transformed how developers write code, especially when it comes to creating net-new functionality. Operating directly on code that's open in a local workspace, these tools leverage the IDE's internal code model and provide contextual suggestions that can speed up coding tasks.

While the rise of AI-assisted development has made it easier to write new code, it's also flooding codebases with duplication and complexity. A recent GitClear report found that, for the first time, duplicated lines of code now outpace refactored ones—making codebases increasingly bloated and harder to maintain. As GitClear warns, "Developer energy may soon shift from building new features to defect remediation as the primary day-to-day task."

While contributing to the maintenance burden, it's also clear that these AI assistants are not built for large-scale code remediation. Their nondeterministic suggestions require human review, as outputs can vary, sometimes hallucinate, and always demand manual validation. They also lack the ability to coordinate changes across repositories or services—making them impractical for systemic, enterprise-level change.

For example, for a solution like GitHub Copilot Autofix that leverages CodeQL to identify security vulnerabilities and large language models (LLMs) to suggest fixes, the process is linear, alert-by-alert, repo-by-repo, and developer-intensive. A CodeQL scan flags a vulnerability and alerts developers. Copilot Autofix translates the description and location of an alert into code changes that may fix the alert. Once the developer reviews and accepts or modifies the fix, they create a PR. The code is then merged, compiled, and rescanned (repeating the process until the scan is clean), which can be very time-consuming in practice. This process happens for every alert across potentially hundreds of developers, over and over again.

Recognizing these limitations, GitHub Copilot Java upgrade assistant and Amazon Q Developer both rely on the open source OpenRewrite deterministic framework behind the scenes to execute accurate framework migrations. They invoke OpenRewrite recipes—safe, rules-based programs—to make precise, repeatable code transformations that far exceed the reliability of AI-generated fixes. However, these tools can only support a limited set of migrations because of the computational load of running recipes on their infrastructure. Running an OpenRewrite recipe is essentially the same as clean-compiling a repository.

GitHub Copilot and similar assistants are fundamentally developer-driven automation. For organizations that need security, compliance, and modernization at scale, an industrialized approach that can work across multiple repositories at once is required.

The Emergence of Automated Remediation at Scale

To manage today's large, complex codebases, teams need a systematic approach to remediation—one that combines semantic code understanding, deterministic transformation, and scalable execution.

Automated code remediation initially emerged with OpenRewrite, a tool created to tackle the problem of reliable, repeatable refactoring. Developed at Netflix by Jonathan Schneider (one of this book's authors), OpenRewrite was born out of the challenges of a "freedom and responsibility" culture, where developers were free to choose their tools and frameworks—but still had to maintain consistency and security across the codebase.

OpenRewrite is optimized for single-repo execution, enabling developers to apply precise transformations—one repository at a time. It provides semantic code analysis and automated refactoring through standalone operations called *recipes*. These recipes are deterministic programs that identify and transform specific code patterns, preserving code style and ensuring correctness. The OpenRewrite core framework is Apache licensed, community driven, and continues to power migrations, security fixes, and code cleanups for teams around the world.

Identifying the bigger challenge of empowering organizations to tackle systemic issues holistically, with the ability to analyze, transform, and validate their entire codebase in real time, Jonathan Schneider and Olga Kundzich (also an author of this book) cofounded Moderne, the first commercial platform designed to scale OpenRewrite across entire organizations.

Moderne automates distributed, real-time analysis and change across codebases—without disrupting existing development workflows. With its unique IP that horizontally scales codebases, Moderne enables teams to apply recipes simultaneously across thousands of repositories, with real-time validation (including fast compilation checks), integration into CI/CD pipelines, and visibility into adoption and impact. Moderne's new AI agent, Moddy, delivers a conversational experience on platform, enabling codebase explorations, standardization, and remediation at scale. Moddy can also

help onboard developers to codebases quicker—all developers have to do is ask.

To power automated remediation and refactoring at scale, there is a vibrant marketplace of OpenRewrite recipes—including community-contributed recipes from framework authors and ecosystem maintainers, as well as proprietary, enterprise-grade recipes developed and curated by Moderne. This growing library spans everything from security patches to complex framework migrations, enabling organizations to apply proven, automated transformations across their entire codebase with confidence.

Beyond the Tools: The Cultural and Operational Shift

When organizations embark on mass-scale software remediation, the first instinct is often to centralize automation and issue mass PRs for product teams to review and merge. On the surface, it feels scalable: run recipes centrally, push out fixes, and let developers approve.

But in practice, mass PRs from central teams often fall flat. Product teams may view them as intrusive or misaligned, rejecting changes over style preferences, timing conflicts, or lack of clarity. Even valid migrations can stall without shared ownership or context. Merge rates tell the story: our enterprise customers have noted only a 20%–30% success rate for centrally issued PRs, depending on the type of change.

To operationalize change at mass scale, teams must begin to think in the context of the change itself—recognizing that not all changes are equal.

There are *push-based changes*, like security vulnerability fixes, which are small, urgent, and surgical. These are ideal for centralized mass PR issuance, where teams expect and accept rapid remediation with minimal disruption.

Then there are *pull-based changes*, like framework migrations or large-scale refactors, which are complex, context-sensitive, and often staged. These require collaboration with product teams, who need to understand what's changing, why it matters, and how to test effectively.

Understanding and embracing this push versus pull dynamic is key to scaling remediation in organizations with minimal friction. Both types of change are essential. Both require different operational paths. In this model, central systems play a key role—curating, testing, and validating recipes, and even issuing PRs—but product teams control issuing, merging, and integrating changes when it aligns with their workflow.

This is the mindset shift: from isolated, reactive fixes to a continuous, context-aware model for code change—where automation supports developers, not overwhelms them, and software evolves without disruption.

How Product Teams Gain Confidence in Mass-Scale Remediation

One of the most common questions we hear from engineering leaders is: "How do we introduce automated remediation without overwhelming our developers?" It's a fair concern. For many teams, the prospect of tools making automated changes to existing code can feel unnerving, even risky. Developers take pride in their code, and rightly so—they want to understand every change, review it for quality, and ensure it doesn't introduce regressions.

Shifting to a continuous, automated model for code change requires more than new tooling—it requires trust in the process. But confidence doesn't happen overnight. It's built over time, through measured adoption and clear results. The key is not just to introduce automation but to introduce it in ways that feel safe, manageable, and valuable to those doing the work.

Let's explore how teams can build trust in automated remediation—starting small, proving value, and evolving toward a model where code maintenance becomes continuous, proactive, and no longer a burden.

Let Developers Stay in Control

It's crucial that developers remain in control of when and how automated changes are introduced. Automation should augment their workflow, not disrupt it. Developers should be able to trigger remediation when it fits within their sprint or release cycle, review

diffs as they normally would, and either accept or reject PRs as needed.

Teams can also integrate auto-remediation into CI gates, aligning with existing shift-left practices. For example, instead of simply breaking a build when a vulnerability is detected, the pipeline could offer an automated fix as a PR or inline diff. Developers still review and approve the change—but they save time by not having to investigate or fix the issue manually.

With automation, teams can also codify and share best practices rather than re-explain them repeatedly. One principal developer put it this way: "I'm tired of going around telling my kids to pick up socks—to use this pattern instead of another." Automated remediation enables the pattern to become the practice—no reminders needed.

Central Teams as Force Multipliers

In large organizations, central platform or productivity teams often manage modern tooling and recipe catalogs aligned to business priorities. These teams act as force multipliers by:

- Curating, testing, and developing custom OpenRewrite recipes for organization-wide use
- Running change campaigns aligned with security, compliance, or modernization goals
- Issuing and tracking mass PRs, reporting on adoption, risk reduction, and ROI
- Partnering with product teams to help them understand different migrations and the recipes available to them—acting as a human expert necessary to complete large migrations

The migration discipline is becoming real, and you can develop it internally in your central engineering or productivity teams. It creates a paved path for safe, scalable change, while leaving developers in control of their repositories—reviewing and merging PRs at the right time, in the right context.

A Side Effect: High Impact Productivity

As auto-remediation gains traction, one unexpected outcome may arise: the number of maintenance or technical debt tickets closed increases dramatically. For instance, in one case, an insurance company closed 1,100% more maintenance stories in a single quarter. Initially, business stakeholders were concerned—did this mean the team was focusing on maintenance at the expense of features?

In reality, the team was not only closing more maintenance work faster—it was also delivering more features (30% more) because of the overall reduced maintenance overhead for the team. Check out the full case study in Chapter 5.

Over time, teams discover that as application code becomes more modern and standardized, it becomes easier to maintain, test, and evolve. Auto-remediation contributes to this by progressively replacing legacy patterns with modern ones, making the codebase more resilient and more approachable.

Now, in the next chapter, let's look at the technology operating behind the scenes to enable mass-scale, automated code remediation.

CHAPTER 3

Technology Behind the Scenes of Auto-Remediation at Scale

To manage continuous, automated code remediation at scale, you need technology that crosses boundaries of teams and repositories. This technology must inspire confidence that code changes via automation are comprehensive and correct. It also must evolve with the software ecosystems it supports.

This chapter dives into the following technology concepts in more detail:

- Rules-based automated code refactoring engine
- Lossless Semantic Tree, a new, structured representation of a codebase
- Recipes, which are programs for performing code search and transformation
- Preservation of code style to avoid rejection when auto-remediating
- Coordination of multirepository code analysis and change
- Data tables output from code examination for analysis and action

The examples we will use throughout this chapter are given as Java refactorings, but the same technology and ideas apply equally to other languages.

A Refactoring Engine Drives Auto Remediation

Auto-remediation and refactoring are closely aligned concepts. Refactoring is the act of changing the internal structure of the software to make it more efficient and maintainable, which includes making it easier to understand and remediate—all without changing the software's observable behavior. It's how we've traditionally addressed technical debt, automated within the IDE and now aided by AI assistants, working repository by repository.

For managing organization-wide software assets, we must coordinate changes across multiple places within the same repository or across multiple repositories. Whether it's updating a dependency version, replacing one API call with another, or adding arguments, the touchpoints in the code can be extensive (and even unknown by developers). For example, changes need to be coordinated across repository boundaries if we want to change APIs and their consumers are in different organizations. When a major framework migration is necessary, the types of changes and level of coordination can be overwhelming for a development team.

Fortunately, refactoring work has three characteristics that make it ripe for mass automation:

- Changes are enumerable and can be expressed as rules.
- The code is in a working state to begin with (i.e., this is not authorship).
- There is a one-to-one correspondence between code before and after fixes.

Based on these assumptions, it is possible to develop a rules-based refactoring engine for code transformations—which is the heart of automated remediation. Basically, we are broadening the capabilities of refactoring and taking them out of the IDE.

With this engine, we are able to assemble rules (in the form of recipes) that are much more complex, progressively encapsulating lower building blocks while still being 100% accurate. For example, there aren't multiple valid branches to implement remediations like upgrading from JUnit 4 to 5. The requirements are immediately enumerable and can be built into rules that are automated. Where there are multiple valid branches, recipes generally hew toward compatibility over optimization. A subsequent recipe can then optimize in a separate action where the scope of change is reduced.

Next-Generation Code Structure for Mass Auto-Remediation

Automated code refactoring at scale required existing scanning technologies to be put aside for a breakthrough technology to emerge. Scanning is based on abstraction or an index of code. Transformation requires full fidelity of code, including all dependencies (transitive and direct), type information, and formatting.

To achieve this, the refactoring engine produces and manipulates a new representation of code called the *Lossless Semantic Tree* (LST). Think of the LST like an IDE's understanding of your code but built to scale. Generated at compilation time, the LST captures the full semantic context of a codebase: syntax, types, transitive dependencies, build tools, and even formatting. Programmatic changes can be made to the LST, and then the modified trees can be printed back to source code. With LSTs, any dynamically formed query or change of code can be performed without an additional database or indexing of code.

The LST artifact is produced by guiding the compiler through the first two phases of compilation to generate a type-attributed abstract syntax tree (AST). The AST is then mapped to the LST that resolves type attribution more deeply than the compiler strictly requires to produce bytecode, and also preserves formatting in the LST. Figure 3-1 shows a comparison of AST versus LST code representations.

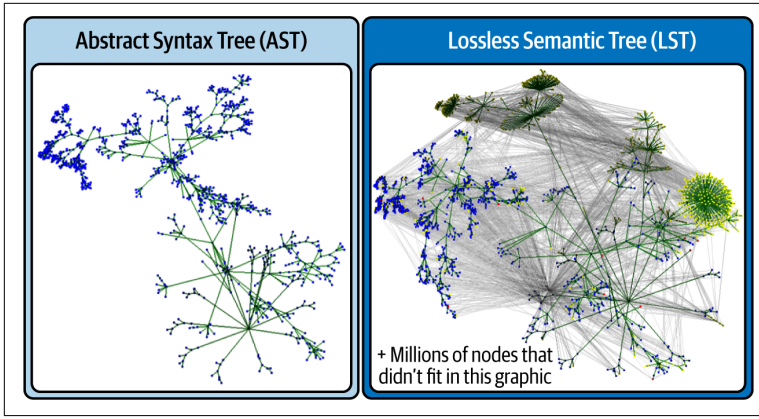

Figure 3-1. Simplified view of a Java AST compared to a Java LST

Because the AST is an abstraction of code, it lacks formatting, semantic awareness, and resolution of dependencies. For example, when identifying an issue such as Log4Shell or applying logging best practices, a scanning tool will be looking for patterns like logger (named destinations). But the AST does not have enough metadata to be able to identify this logger coming from Log4J or other popular logging libraries. So it will overreport and have false positives identifying other logger types, resulting in extra work for developers to examine.

The LST is an evolved data structure in that it:

- Preserves formatting—even inferring the local and global formatting preferences from observation.
- It preserves the entire type awareness that the compiler possesses in its intermediate representation before producing its final output. We call this *type attribution*.
- The LST handles incomplete type information (i.e., is not optimized for grammatical error tolerance).
- It can be searched quickly at scale (i.e., is designed for round-trip serialization).

As you can see in Figure 3-1, even a pruned representation of the LST creates a very dense tree. This very simple snippet of Java code without many binary dependencies yields an LST containing 8.1 million vertices!

With this many nodes and vertices, efficient deduplication is key to the performance of the refactoring engine using the LST. Deduplication is itself a complex process. Even two method invocations against the same method name with the same fully qualified receiver type name and the same arguments could represent two entirely different trees. An example would be when two method invocations are pointed at two different versions of the same binary dependency.

Having full type attribution is essential to making many remediation decisions. Without it, we will have false positives.

The rich LST artifact is also proving to be an optimal code data source for AI large language models (LLMs). We've found that the more concise, complete, and accurate the data you feed to the model, the fewer hallucinations and better the results, which is critically important for making mass-scale code changes you can trust. We'll cover this more in the next chapter.

Recipes Encapsulate Transformations and Searches

Recipes are actual programs for performing specific search and transformation actions on source code. They operate on the LST, visiting and transforming different parts of the tree, and then the refactoring engine can print the LST back as text for humans to review and accept changes.

There are many building-block recipes, such as find method, change method, find transitive dependency, upgrade dependency, and exclude dependency. These recipes in turn can be composed into more complex recipes by grouping them into a composite recipe. When the building blocks are not enough, a recipe can be written as a program in the same language as the code we want to transform—encapsulating complex logic with the full expressiveness of the language already familiar to developers.

These building blocks abstract many of the details to ensure that edits that we make to source code obey the original style of the project. Recipe authors focus on transforming code, and the refactoring engine tracks the style of projects and application of recipes to codebases in a style-preserving manner.

To understand the power of recipes, let's look at a major version framework migration in the Java ecosystem: upgrading to Spring Boot 3.4 from Spring Boot 2.3 and Java 8. You have a lot of work ahead:

- Spring Boot 3.x migration requires at minimum Java 17.
- Java 17 requires first migrating to Java 11 and also includes migration to Jakarta EE 9.
- You also have to migrate from Spring Boot 2.3 to 2.4, which requires updating JUnit 4 to 5, which requires updating Mockito 1 to 3.

Multiple, chained recipes can complete this complex migration (shown in Figure 3-2) by addressing all dependency upgrades needed to achieve functioning software at the end of the remediation. For complex framework migrations, we have found that recipes automate the largest portion of the work (90% or more), but some manual changes will need to be made because of architectural requirements specific to an organization. The good news is that organizations can build their own custom recipes to address specialized cases at scale.

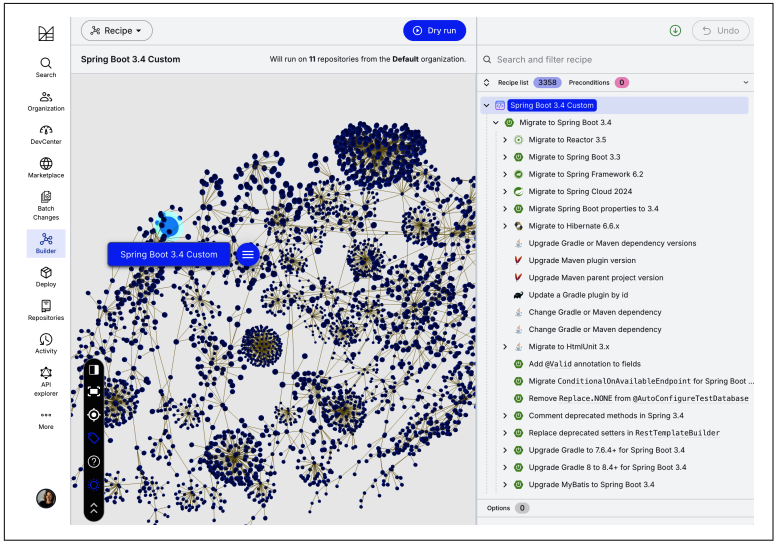

Figure 3-2. Visual of Spring Boot recipe with over 3,000 migration steps

There are also recipes designed to only search the code—treating search like a special case of transformation by adding markers on LST elements. We can query anything that the LST contains. The recipe's results then can be printed as text with special representation for markers (usually as comments).

In the world of assembled software components, you can see how recipes built by framework and library authors can become the migration experts and drive the auto-remediation action. Over time, OpenRewrite recipes can cover the whole OSS ecosystem, supporting automated remediation for the menagerie of interconnected software and all the users. Enterprises that consume those recipes just need to apply auto-remediation to fix the issues.

Why Recipes Need to Be Programs

It is important that recipes themselves are programs to provide the flexibility for encapsulating custom logic and building integrations. For illustration, consider these real-world examples not possible to achieve with a declarative syntax:

- A large fintech company uses LaunchDarkly feature flagging and wanted to clean up its code by removing unused flags. The developers wrote a recipe that starts by making an API call against LaunchDarkly to list unused feature flags. Armed with this real-time knowledge, the recipe then operates on the codebase to remove code branches associated with those unused flags. In a second phase, a follow-up recipe lists unused feature flags and automatically deletes the unused flags.
- Organizations can build variants of OSS dependency vulnerability remediation recipes to interrogate their own internal and proprietary vulnerability data sources. This enables them to use information about vulnerabilities present in the transitive dependency closure of a project so that the recipe can make progress on vulnerability remediation, informed by the same data source that the organization already reports against.

It is easy to layer declarative syntax on top of an imperative implementation. Most building-block recipes can be assembled with simple YAML syntax, but to express complex transformations, the power of a programming language is necessary.

The Importance of Style Preservation for Auto-Remediation

We can think of an automated fix as a kind of transplant that we are inserting into someone's body of code (actually, many such bodies of code when applying it en masse). A body that does not recognize a transplant as its own rejects it. If you have ever witnessed the ongoing debate of tabs versus spaces in the past decades, you will understand why this is important.

Adopting the attitude that a fix must be carefully crafted to not be rejected is a hard task. In modern enterprises outside of a select few high-tech companies, stylistic inconsistency is the norm—even if an organization has every intention of agreeing on and enforcing a particular style. It is the side effect of many years of evolving best practices (and sometimes new language features) that somehow make older code look different, even if we are perfectly consistent in how we write all the new code.

When the LST artifact compiles, it captures formatting and associated syntax elements by going back to the source code as text and scooping it up. From the formatting present on the LST, a style governing the project can be derived. For example:

- Does the project use tabs or spaces?
- Does the project use star import formatting?
- How many types need to be present in a package before it is star-folded?
- How are import statements grouped together?
- What are the project's standards around the use of blank lines between methods, between classes, before the first method, at the end of the file, and so on?

The styles themselves are captured on the LST in the form of markers implementing the Style interface. Then, as recipes make edits to the LST, they are able to use autoformat helpers to shape the edit in a way that looks idiomatically consistent in the context of the codebase that is being inserted.

When a recipe transforms the code, the style of the project captured on the LST dictates how the code is printed back as text. The same change applied to different projects can look very different. A recipe author doesn't need to care and anticipate all the different styles.

Coordinating Automated Code Remediation at Multi-Repo Scale

The first question when people encounter automated remediation is whether it is really possible. And the second is, How do I coordinate the distribution of these recipes and changes at scale?

Our answer is that it's not just a technical problem; it's also a cross-team collaboration problem. In the past, tracking changes and understanding the impact on codebases was also manual. It required a project manager with a spreadsheet gathering data from engineers.

To assist in the second problem, you need a horizontally scalable system operating on LSTs as a data structure, using recipes to enact any query or code transformation. Recipes can run and complete in minutes across 100 million lines of code. Code changes are coordinated in real time. When a new vulnerability comes in, you can quickly ask where it is in your codebase. You can implement change campaigns to migrate code or perform weekly bumps of all dependencies.

Doing this at scale across a codebase with a platform like Moderne helps engineers validate the quality of recipes and builds an organization's practice of proactive, continuous improvement to the codebase. By being able to perform batch changes and issue mass pull requests (PRs), organizations can stay aligned all the time. In addition, recipes can output data about code, and you can leverage existing data analytics tools to study your code and the impact of the transformations.

Having a center of code intelligence for your organization enables you to have full, strategic visibility to reason about and enact change to your codebase. Data tables, discussed next, are a great example of this.

Impact Analysis with Structured Data Derived from a Codebase

In addition to recipes making edits to the LST that result in code transformations, at the same time we have an opportunity to emit structured data as a side effect. A system like Moderne, which is operating these recipes across hundreds of millions of lines of code, can aggregate that data across a large set of code assets and present the data in a tabular form (e.g., comma-separated values (CSV), Excel, Apache Parquet, Apache ORC, etc.) that is suitable for use in big data and analytics tools that exist in some form in every enterprise.

Given how much data is actually on the LST, the possibilities for impact analysis are practically endless. As an example, data tables already exist in different recipes that produce a variety of outcomes. Here are a few exciting ones:

- The source code of every method call matching a pattern
- Every license in use by direct and transitive third-party dependencies
- Every security vulnerability detected in direct and transitive dependencies, as matched against the GitHub Advisory Database, including the Common Vulnerabilities and Exposures (CVE) number, description, actual version of the dependency in use, minimum fixed version, severity, and depth of dependency
- The relationship between individual source files and relational database management system (RDBMS) table columns and whether that column is selected, updated, inserted, or part of a deletion
- Every Spring API endpoint found and which component it was defined in
- Every outbound API call made by RestTemplate or WebClient
- The relationship between Spring components through dependency injection relationships, which effectively allows us to build an architecture diagram on the fly for any given application
- The definition of every Jakarta Enterprise Beans (EJB) 1, 2, and 3 component

The exported data also can direct action by an OpenRewrite transformation recipe, and AI agents can also use it for knowledge.

Emitting data tables in structured form like this also permits us to use analytics tooling to great effect to summarize what we are seeing and take action. One such integration already built into Moderne is Jupyter Notebooks, which can source the data directly from the API and produce visualizations. For example, Figure 3-3 shows a graphical representation of dependency vulnerabilities. AI agents also have access to tools that can build graphs when provided with the right structured data.

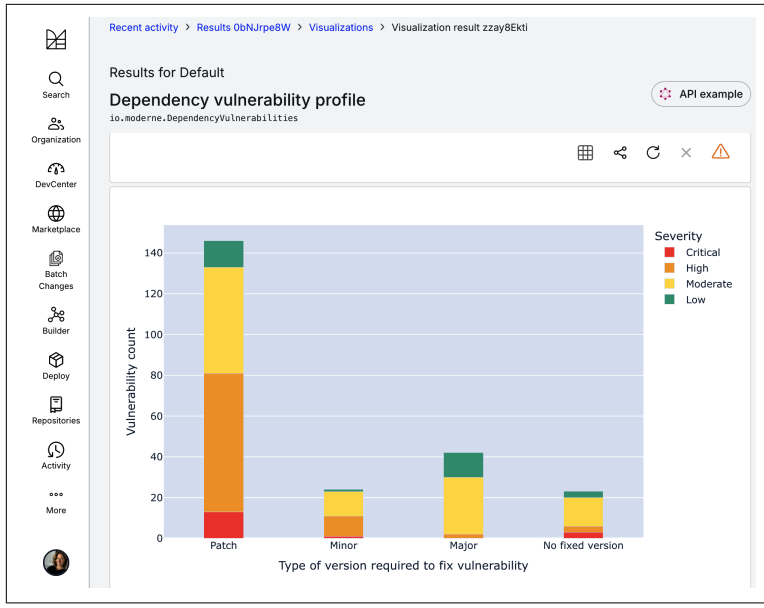

Figure 3-3. A code visualization showing severity of vulnerable dependencies

When the recipe is written as a program, it is easy to pluck pieces of data from random parts of the LST and stitch them together to build insights into a wide body of code.

With the power of the LST and rules-based recipes, code analysis and changes are possible at massive scale. Now, let's move on to understand the role that AI can play to take things a step further.

CHAPTER 4

The Role of AI Agents in Mass-Scale Auto-Remediation

The rise of AI-assisted development has opened up new possibilities in software creation. Developers now have tools that can generate code faster, explore frameworks more easily, and accelerate feature delivery. But with this increased velocity comes an equally fast-growing burden of maintenance. Every line of code, every dependency, every library introduced today becomes a long-term responsibility—one that must be kept secure, compliant, and compatible.

For enterprises managing thousands of repositories, many of which are deeply interconnected through shared libraries and legacy integrations, the challenge isn't just about writing code—it's about maintaining and evolving code at scale. When a security vulnerability arises or a framework update is required, it's no longer sufficient to address one repository at a time. Organizations must be able to analyze, refactor, and validate changes across their entire codebase, all while ensuring consistency and minimizing disruption.

The next evolution in AI isn't about writing more code—it's about making existing code better, intelligently and at scale.

This chapter explores how AI agents, when paired with deterministic tools and structured code data, can deliver safe, reliable, and scalable code remediation. For this discussion, we'll be referencing how the Moderne AI agent Moddy works to remediate at scale. We'll examine how AI shifts from suggestion engines to orchestrators of

accurate change, capable of transforming millions of lines of code across thousands of repositories—with precision and confidence.

AI Agents and Tools: A New Era of Scalable Remediation

To truly scale code modernization, AI needs more than pattern recognition or text generation—it needs structured, precise execution paths. That's where agentic AI comes in, moving beyond passive code suggestions into the realm of autonomous action. An AI agent doesn't just propose code changes—it plans, executes, and adapts, operating through feedback loops and data-driven decisions.

The advent of tool-function calling, through tool-calling paradigms such as Model Context Protocol (MCP), OpenAI agent framework, etc., unlocked a hybrid approach, combining probabilistic language understanding with deterministic execution. This dramatically expanded what AI agents could do reliably. Thousands of OpenRewrite recipes—each a deterministic program capable of safely and predictably refactoring code—suddenly became callable tools for agents. Instead of generating fixes from scratch, agents could now invoke trusted recipes to act across large codebases.

AI agents now have a "toolbox" of capabilities, each registered with structured inputs and outputs. LLMs can call specialized tools, such as code transformation recipes, compilation verifiers, and documentation updaters, selecting and executing based on the task at hand. Crucially, this technique allows for repeatability and safety, which are nonnegotiable when making wide-scale changes to business-critical code.

For instance, if the goal is to understand how the Guava library is used across a codebase, Moderne's agent, Moddy, can locate and run the appropriate OpenRewrite recipes (e.g., FindMethods), configure recipes correctly to look for usages of com.google.common package, gather data from across repositories, and synthesize a coherent response—all without requiring human involvement. LLMs understand what a human means when they ask about Guava, able to interpret the request and invoke the computer to do the accurate calculation. Figure 4-1 depicts this flow.

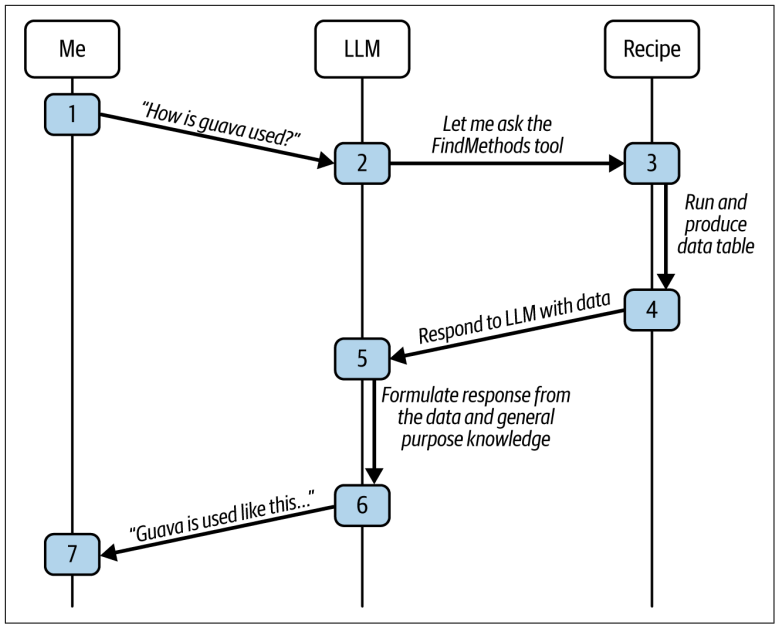

Figure 4-1. AI agentic tool calling and flow

Through Moderne's MCP client, any agent, such as custom-built AI systems you might find in enterprise environments, can call Moderne tools to perform transformations, extract insights, or validate changes at scale.

The Importance of Massive Structured Code Data for AI

For AI agents to operate effectively on code at scale, they need more than just access to source files. They need access to the right data—a deep understanding of code structure, dependencies, and semantics, as well as visibility across repositories—understanding how shared libraries, services, and APIs connect throughout the broader codebase.

This is where Lossless Semantic Trees (LSTs), which we introduced in Chapter 3, come into play. An LST artifact provides a fully resolved view of a repo of code, capturing type hierarchies, method definitions, and dependency relationships. The Moderne Platform brings scale to the single-repo LST data with its unique IP that serializes LSTs to disk, enabling the platform to horizontally traverse

the LSTs for near-instant recipe execution and real-time, multi-repo analysis.

This infrastructure makes it possible for Moderne's multi-repo agent, Moddy, to act on entire codebases with context, speed, and confidence. With serialized LSTs, an agent doesn't just see that a method exists—it understands how that method connects to the rest of the system, what libraries it depends on, and what downstream effects might result from changing it.

From serialized LSTs, knowledge graphs can be automatically constructed to further enhance AI reasoning. These graphs can include data like:

- Architectural patterns (e.g., usage of singleton, Builder.io, or reactive paradigms)
- Business logic flows (e.g., all code paths related to authentication or payment processing)
- Versioned dependency usage across internal and external libraries

Armed with this data via tools, AI agents like Moddy can intelligently answer questions, recommend safe changes, and guide remediations with architectural and domain context—not just surface-level pattern recognition.

In short, without the rich, structured data of LSTs—and the infrastructure to leverage them at scale—AI for code is flying blind.

Real-Time Validation: Scaling with Confidence

One of the greatest challenges in large-scale code remediation is validation. With many AI assistants today, the sequence of code change is *AI makes change*, then *Run build tool to verify compilation*, then *AI makes change to fix compilation failures*, and so on. Imagine executing the build for every change for every repository at once. This is a massive, computationally intensive, time-consuming process. Yet developers need confidence that automated changes won't introduce regressions or break builds.

To address this, AI agents can use compilation verification tools (i.e., a recipe designed to verify compilation). These tools offer fast,

lightweight checks that validate whether the transformed code will compile without executing a full build.

When leveraging AI for large-scale code transformation, those early, instant signals of change quality are essential for managing change at scale. Immediate feedback enables the agent to fail fast, iterate quickly, and avoid pushing flawed changes downstream. It's a crucial part of scaling remediation while minimizing the risk of errors or disruptions.

The AI Agent Workflow in Practice

An AI agentic system (see Figure 4-2) leveraging deterministic recipes and LSTs as the code data source via tool calling could answer a query about a large codebase by:

- Guiding users through relevant OpenRewrite recipes
- Configuring and executing recipes
- Providing recommendations
- Summarizing the discoveries or deterministic changes
- Discussing discoveries with the user
- Validating changes from recipes

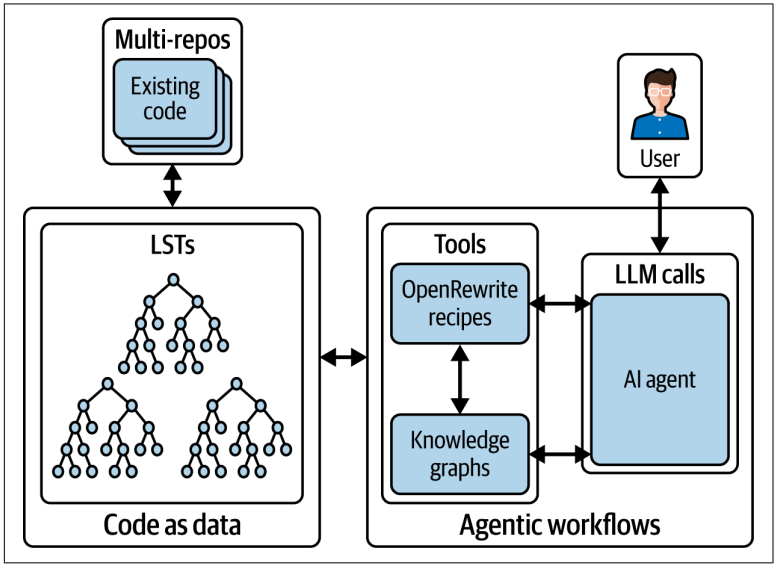

Figure 4-2. Agentic system leveraging tool calling and LSTs

Let's walk through how this all comes together in a real-world scenario. For example, if a developer wants to migrate legacy database usage to PostgreSQL, they can ask the agent how SQL is used across the organization's entire codebase. The agent will identify the appropriate recipe to locate SQL usage, then execute it across all repositories, gathering the results in a structured format that the LLM can analyze and summarize for the developer.

This end-to-end automation, working across multiple repositories at once, enables developers to understand and evolve entire codebases like never before.

CHAPTER 5

Case Studies in Automated Remediation at Scale

Now that we've covered the background and technology for automated code remediation, we want to share some real-world case studies exploring the practice. You'll see what leads organizations to automate code remediation and its impact on the way they work.

Case Study: Improving Enterprise Productivity

Our first case study takes us back to the problem of technical debt. A midsize insurance company, with over 20 million lines of code across 1,200 repositories, was struggling to close code maintenance stories without impacting the development team's productivity. The company's code maintenance work, which covered vulnerability patching, code migration, and dependency upgrades, was folded under technical debt in its systems.

The organization was averaging one large maintenance project per quarter, and each of those projects could consume the entire development team. For example, in Q2, the team had to prioritize a Spring Boot version upgrade to secure its code from the Spring4Shell vulnerability. This became 32 different stories for the development team—one per repository. It also required one developer per repository acting as the "migration expert" for the update. It was all-consuming and reduced the organization's business output, as you can see in the Q2 results shown in Figure 5-1.

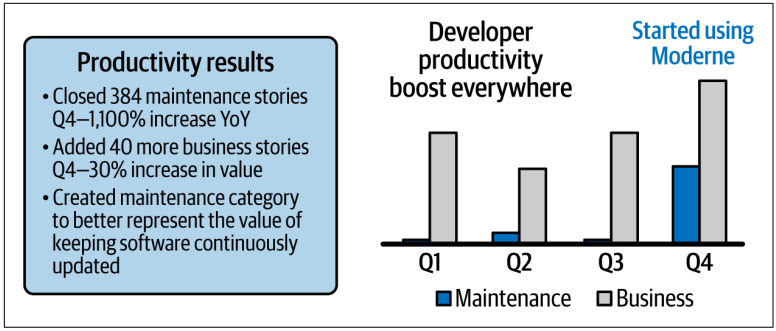

Figure 5-1. Insurance company boosts developer productivity with auto-remediation

Then the organization started using automated code remediation with Moderne. Instead of a whole development team touching multiple repositories to maintain code, a single developer in less than a day could have the Moderne Platform update code and issue mass pull requests on the team's behalf across repositories. Then the code would follow the normal workflow to test and deploy. Much of the overhead and work of code migration was handled by the platform.

The team gained comprehensive, accurate visibility into its codebase—a better understanding of code and transitive code health—that is actionable through automation, enabling the team to make significant, constant progress in code maintenance. This is demonstrated through the team's productivity results, as shown in Figure 5-1. In the quarter when the team was fully using auto-remediation, not only did it close 384 maintenance stories (a 1,100% increase), but the team also added 40 additional business stories (a 30% increase).

The company is also separating true "technical debt" from "maintenance" stories. Now upgrades in software versions and their cascading dependencies—something that is beyond the development team's control—have been recategorized as maintenance. This will significantly reduce the black hole of technical debt and the perception of bad coding practices for the team.

Case Study: "Freedom and Responsibility"

This case study features Jonathan Schneider and his experience with the Netflix "freedom and responsibility" culture and driving change. This was a company that relied on a culture of high-performance, creative, self-disciplined workers, not process adherence, to achieve corporate goals. The only "good process" was something that helped talented people get more things done.

Regarding engineering (specifically, the source code), Jonathan had only ever seen centrally initiated efforts to manage developer workflow backed by executive sponsorship defeating successive waves of organizational resistance through sheer force of will. What he soon learned is that the best solutions can emerge when centrally forced change does not meet the culture.

"Freedom and responsibility" at Netflix allowed platform teams to test the value of solutions without the effectiveness bias that being able to force a solution causes. Jonathan saw this firsthand when he unwittingly stepped into a migration engineering role. At the time, he was wrestling with how to get teams from Java 6 to 8, Gradle 2 to 5, and Framework version 1 to 2.

The diffusion of responsibility created challenges for these migration projects. Mass communication to development teams turned out to not be the answer. Providing code search and reporting didn't help. Some of his colleagues went so far as to create a hack day project that identified when an engineer made a breaking API change as well as the teams that would be impacted. They automated the production of one-off videos that began with the title *Here's Who You Hurt Today*, followed by a series of profile pictures of engineers on impacted teams, with background music by Sarah McLachlan singing about angels.

All the communication in the world generated minimal action. Why? The diffusion of responsibility means no one has a sense of urgency, which is why we experience internal resistance, even in top-down organizations.

Being required to negotiate (*beg* might be a better word) with product engineers for a change, Jonathan heard the same refrain: "Do it for me, and I'm happy to accept the change." But how could he, as a member of a small team, "do the work" for everyone else?

It led him to an entirely different solution, one he would not have imagined without "freedom and responsibility." He invented an open source automated refactoring solution called OpenRewrite. As a tool that would essentially fix the source code for developers, it fit in nicely with the Netflix culture and "good process" notions.

Case Study: Remediating Security Vulnerabilities Proactively

When Interactions, a leader in creating conversational AI assistants for large-scale enterprises, embarked on a journey to modernize its tech stack, the company faced many challenges. Like many technology companies, over time, Interactions had accumulated natural technical debt across many areas of its stack due to the manual nature of remediation along with running on legacy JDKs. The company needed a solution that could not only address these issues but also keep them compliant with strict security standards.

Because Interactions operates in a billable work model, this added to the challenge of executing code updates and upgrades. The need to estimate, plan, and gain approval for every nonbillable task added layers of complexity that could slow down essential modernization efforts, making it difficult to maintain the agility needed to keep the company's technology stack current and secure.

Moderne offered a comprehensive solution that automated code migration, security remediation, and code cleanup, enabling Interactions to streamline its code maintenance, reduce manual effort, and focus on delivering high-quality, customized solutions to its clients.

The Interactions security tool at the time, Veracode, was primarily focused on basic vulnerability scanning. While it served its purpose, it lacked the remediation capabilities needed to address the growing technical debt and perform large-scale code remediations. This meant the process of remediation was extremely manual and time-consuming, particularly given the scale of their codebase.

Interactions implemented a custom security recipe that automated the process of identifying and fixing vulnerable dependencies. As one of the company's engineers commented, "Historically, we wouldn't have been able to do this on any reasonable timeline. Now, we run the vulnerable dependency fix at a bimonthly interval, or it can be run on-demand, and it's made a world of difference."

This proactive approach to security has allowed Interactions to stay ahead of potential threats, ensuring its applications are secure without the need for extensive manual intervention. The regular automated checks provided by Moderne have become an integral part of its security strategy, delivering peace of mind and allowing the team to focus on more strategic initiatives.

All manner of companies are already discovering the power of automated code remediation to drive their codebases forward faster. Let's see what the future looks like as we conclude this report.

CHAPTER 6

Beyond the Prompt: How Codebases Evolve at Scale

We are in a new era of software engineering—one defined by automation and AI.

Enterprise software isn't a greenfield environment; it's inherited, interconnected, and constantly evolving. Most teams manage large volumes of older code and technical debt spread across hundreds or thousands of repositories. These systems must adapt to a software supply chain in constant motion: vulnerabilities appear, libraries update, and frameworks deprecate APIs. Meanwhile, AI-assisted development is accelerating the creation—and duplication—of new code, adding even more surface area to maintain. In this environment, any meaningful code change must be applied consistently, at scale, and in compliance with an organization's security, architectural, and quality standards.

The challenge is no longer the efficiency of writing code—it's keeping up with it.

The Limits of AI Developer Assistants

Without a doubt, AI will play a significant role in the industrialization of software. For example, tools like GitHub Copilot have become powerful assistants for developers—helping them write net-new code faster, experiment with unfamiliar APIs, and speed up task completion in the IDE. But developer assistants alone aren't built for

the kind of large-scale, coordinated work enterprises must do to stay modern and secure.

They offer suggestive, nondeterministic outputs that are limited to a single file or repository. Their understanding of broader architectural context is shallow, and their outputs often require manual validation. Refactoring and large-scale remediation aren't just weaknesses of developer assistants—they're outside the realm of what these tools are built to do.

Even AI-enhanced security workflows, like those that combine CodeQL scans with LLM-generated fixes, fall into slow and repetitive cycles: scan → suggest → test → repeat. They remain developer-intensive and don't solve for systemic coordination.

In short: AI developer assistants are great at writing code but poorly suited to maintaining it at the required scale.

The Future: Scalable, Trustworthy, Agent-Driven Code Modernization

Code remediation at scale requires accuracy, consistency, coordination, and trust. It means changing code across thousands of repositories, without breaking builds or disrupting business priorities. That's where agentic AI—paired with the right deterministic tools and architecture—changes the game.

For example, the multi-repo AI agent Moddy operating within Moderne's platform, doesn't just suggest changes; it leverages:

- Structured code data like serialized LSTs to understand system-wide impact
- Deterministic tools like OpenRewrite recipes to apply safe, rules-based transformations
- Real-time validation and feedback loops so changes can be applied with confidence

It's a new model for maintaining modern software: interactive, intelligent, and industrialized.

With this model, engineering leaders can finally move beyond the endless backlog of technical debt. They can reduce developer

burnout, accelerate modernization, and ensure that their codebases remain secure, consistent, and ready for what's next.

Join the Software AI Revolution

None of us can imagine a future for software that doesn't involve AI. But what that future looks like—and how AI best supports development teams—is still unfolding.

What we do know is this: AI needs data. Today's IDE-based AI assistants and chatbots are quite good at suggesting code during authorship when a developer is in the loop to review and guide the outcome. But to truly operate at scale—across massive, interdependent codebases—AI must go beyond suggestion. It must be supplied with accurate, consistent data equal to the scale of our modern codebases.

Enter AI agents brandishing powerful tools, such as Moderne's platform. These solutions will help you maintain and evolve your codebases at a pace and scale never before possible—becoming a force multiplier for businesses that can open the door to billions of lines of code.

With this expansive capability comes change. As agents gain the ability to analyze and act on entire codebases, engineering culture and processes will adapt. Teams will shift from firefighting to proactive remediation, from reacting to change to shaping it.

Across the organization, every role has a part to play: developers transforming legacy code, security engineers patching CVEs, team leads enforcing coding standards, software architects driving structural consistency, and framework authors guiding migrations. With the help of AI assistants, these experts can infuse their knowledge into recipes—codifying once-manual work into reusable programs.

Those recipes can then be executed by agents, consistently and safely, across thousands of repositories. From dependency upgrades and security patches to modernization and compliance, this combination of assistant-augmented creation and agent-driven execution transforms how teams manage code at scale.

Your next steps:

- Explore OpenRewrite to understand the power of semantic, rules-based code transformation.
- Try Moderne and Moddy to experience remediation at enterprise scale.

What you'll find is that modernizing at scale isn't just maintenance—it's momentum. It's how you create the space to adapt, invent, and build without limits.

About the Authors

Pat Johnson is head of product marketing for Moderne, which automates software remediation at scale. She has published and presented on DevOps and cloud computing topics, with a focus on improving the developer and operator experience alike. Previously, she was the portfolio marketing lead for the Tanzu cloud native application platform at VMware and product marketing manager for CI/CD solutions at both Pivotal and CA Technologies.

Olga Kundzich is cofounder and CTO at Moderne, which automates software remediation at scale. She has extensive experience building enterprise software solutions. Previously, she worked as a technical product manager at Pivotal, focusing on application delivery and management solutions (e.g., Spinnaker); and she was a lead software engineer and manager at Dell EMC, working closely with enterprise users on implementing data protection practices.

Jonathan Schneider is cofounder and CEO at Moderne, which automates software remediation at scale. He founded OpenRewrite, an auto-refactoring tool, at Netflix and went on to found the Micrometer project as a member of the Spring Team. Jonathan is a Java Champion and the author of *SRE with Java Microservices* (O'Reilly, 2020).