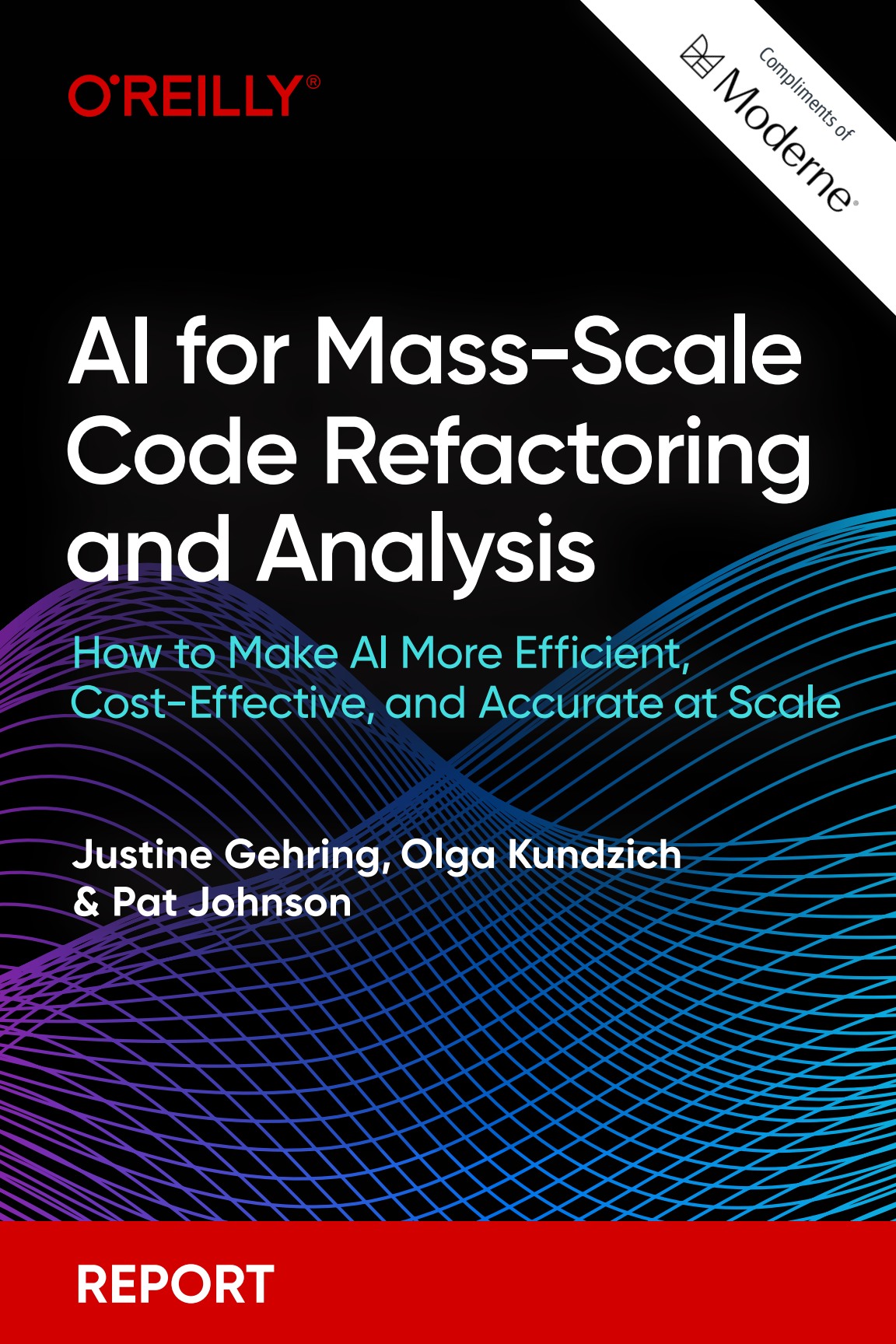
O'REILLY®
Compliments of
Moderne
AI for Mass-Scale Code Refactoring and Analysis
How to Make AI More Efficient, Cost-Effective, and Accurate at Scale
Justine Gehring, Olga Kundzich & Pat Johnson
REPORT

moderne.ai

AI for Mass-Scale Code Refactoring and Analysis

How to Make AI More Efficient, Cost-Effective, and Accurate at Scale

Justine Gehring, Olga Kundzich, and Pat Johnson

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

AI for Mass-Scale Code Refactoring and Analysis
by Justine Gehring, Olga Kundzich, and Pat Johnson

Printed in the United States of America.

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

Acquisitions Editor: Louise Corrigan
Development Editor: Gary O'Brien
Production Editor: Christopher Faucher
Copyeditor: nSight, Inc.
Proofreader: O'Reilly Media, Inc.
Interior Designer: David Futato
Cover Designer: Susan Brown
Illustrator: Kate Dullea

September 2024: First Edition

Revision History for the First Edition
2024-09-09: First Release

978-1-098-17581-8

[LSI]

Table of Contents

CHAPTER 1

AI for Code: Ready for the Masses?

Just as integrated development environments (IDEs) revolutionized software development in the 1990s, AI-driven code generation and refactoring tools promise a step-function improvement to the way developers write, maintain, and optimize code today. Practically every software engineering organization right now is evaluating and implementing AI solutions with various goals in mind, such as to help their teams deliver more features faster, fill skill gaps, gain efficiencies, improve code quality, reduce technical debt, and save costs.

The shining star of AI for code thus far has been generative AI (GenAI) assistants or chatbots. These assistants, readily accessible on the market, are making it easier than ever for developers to write code. We're already seeing the huge impact of tools such as GitHub Copilot, reporting faster developer task completion—55% faster—as well as code quality improvements. What an exciting time for developers!

Meanwhile, at Moderne, we have been focused on helping enterprise companies manage their large and growing codebases—including their proprietary source code along with thousands of open source software components. Our platform enables accurate, automated code refactoring and analysis across these massive codebases using deterministic recipes and a novel dataset for code. We have also been excited by the surge of AI models and techniques available to us now and have readily jumped in to discover what value they can bring to our platform and our customers working at scale.

During our AI explorations, we have faced some distinct challenges using GenAI alone to maintain and secure existing codebases. Every single edit with AI is based on immediate context and is suggestive, requiring a "human in the loop" to review. This system obviously doesn't scale if you are refactoring and analyzing across multiple code repositories at once. To refactor at scale, you need accuracy and trust in your system. We see a number of ways to leverage the value of AI while improving accuracy for mass-scale code change.

AI is a term that covers algorithms that mimic intelligence. Some earlier AI, such as A.L.I.C.E, were able to mimic, to some extent, a conversation by applying heuristic pattern matching rules to a human's input. Machine learning (ML) is a subtype of AI that uses past data to identify patterns to make predictions or decisions, and deep learning is a subtype of ML, which is what we mostly refer to when we refer to AI and large language models (LLMs) today.

We'll start this report by sharing the impact of AI assistants on software development. Then, we'll dive into the heart of the matter: maintaining and securing massive codebases—and how AI can be optimized for mass-scale code refactoring and analysis, based on Moderne's own experience.

The Rise of AI Assistants in Software Development

> *The last wave of machine learning gave you infinite interns who could read anything for you, but you had to check, and now we have infinite interns that can write anything for you, but you have to check.*
>
> —Benedict Evans[1]

GenAI is a probabilistic system incentivized to do what its name denotes: generate. By leveraging ML models trained on vast datasets, these coding assistants can predict and suggest text and code snippets. They can also supply data or context to the model at the time of query or generation as developers code.

1 Benedict Evans, "AI and the Automation of Work," *ben-evans.com* (website), July 2, 2023.

AI assistants—those infinite interns—are changing the way developers work. Instead of writing everything by hand assisted by a rules-based IDE that is 100% correct, developers are interacting with their AI assistants, evaluating suggestions that they then accept or reject. The rules-based system may still help in writing a majority of code, but AI suggestions offer another level of developer productivity improvement.

At Moderne, when our developers were just starting to use Copilot, they were often amazed that it seemed to know about some novel code they were working on. Surely it was not trained on this yet! What developers quickly learn is that Copilot doesn't just use the data it was trained on to provide suggestions; it also leverages content from the developer's IDE such as the file opened, additional files opened, and recent files.

As developers learn how to harness the power of Copilot, they learn how to guide it by staging additional data, opening similar files containing the type of code they are trying to write. This suggests to Copilot what to include in the prompt to make the large language model (LLM) output more relevant. This works really well for code generation.

The Challenges of Using AI to Refactor and Analyze Code at Scale

We all may collectively be exiting the "hype" stage of technology adoption when it comes to GenAI coding assistants and getting to the practical realities of using these tools. As we have come to experience, AI suggestions can seem prescient, intuitive, and extremely helpful, but they can also be wrong, unnecessary, buggy, and prone to hallucinations.

The downside of AI inaccuracies can be seen in the rise of churn rates for code. Churn is when incomplete or error-prone code is committed or pushed to the source code repository and then quickly reverted, removed, or updated. This is like two steps forward and one step back for development teams.

The proclivity of GenAI to "generate" is also leading to less reuse and refactoring of the code, which can exacerbate the problems of code maintenance. One study shows a 17% decline year over year of moved code since the adoption of GitHub Copilot. Experienced

developers readily see the value in reusing code because it's already battle tested in production and likely touched by other developers. But GenAI models such as Copilot like to re-create rather than reuse.

At Moderne, our developers use Copilot regularly and have experienced its shortcomings. One developer using Copilot implemented a feature that required supplying configuration properties for deployment. One of those properties was JAVA_HOME. Copilot, which uses OpenAI LLMs trained on a lot of data (including GitHub open source software (OSS) and Stack Overflow community content), suggested *JAVE_HOME* instead of *JAVA_HOME* for the configuration—a typo in its training data. We were able to track this down to the exact Stack Overflow post after the fact.

Our developer accepted the suggestion, not noticing this one error among many correct configuration properties suggested by Copilot. Deploying this feature in production, however, caused an outage, as the Java version was not found. After some panic and intense scrutiny by the team, we discovered the misspelling of JAVA_HOME in the merged pull request (PR). This illustrates that even the most senior developers can be susceptible to AI errors and hallucinations—things that in quick review may look correct but aren't.

As engineers working with AI, we often find ourselves fine-tuning our queries to achieve the precise results we need. For instance, adding explicit instructions such as "do not alter any other cases" can improve the AI's performance. Despite this, even a small margin of error—say, one incorrect output out of ten—can be problematic when scaling up. Imagine a scenario where your codebase contains 1,000 switch cases. A 1% error rate means 10 cases might have altered the logic behind the switch case, posing significant risks. In addition, asking the model multiple times does not improve this error rate, as is sometimes claimed. This mistakes the relationship between reliability and validity. A model may reliably answer inaccurately (or unreliably answer accurately).

Knowing What We Now Know, How Can We Use AI for Mass-Scale Code Refactoring?

Mass-scale code refactoring can involve affecting changes across many individual cursor positions in thousands of repositories, representing millions to billions of lines of code. It's a multipoint operation that requires coordination, accuracy, consistency, and broad code visibility. With current AI assistants, you are working file by file augmenting them with local, limited context. You can end up with incomplete, error-ridden, and very developer-time-intensive code maintenance.

Incorporating techniques such as retrieval-augmented generation (RAG), which we'll discuss in more depth in Chapter 2, "Techniques for Scaling AI for Large Codebases", ensures models remain efficient and effective without the overhead of larger systems. Additionally, employing a strategy that uses multiple models from least to most computationally demanding, along with OpenRewrite recipes or different libraries for various rules-based subtasks, can further optimize performance and capabilities.

Ultimately, for mass-scale changes, you may find that it is more time- and cost-efficient to take a rules-based recipe that's already been validated to fix the issue—then run this recipe automatically across the repos to fix the code. You can leverage GenAI to help develop the recipe (because recipes are tested before they are put to use). You can also use AI to help you identify applicable recipes to fix the problem or even to search the codebase for more detailed analysis that yields recipe recommendations.

In this report, we'll cover what mass-scale automated code refactoring with AI looks like in practice. You'll get details of:

- AI techniques and LLMs to employ
- Use cases for AI when working with large codebases
- Considerations of large enterprises when adopting AI for mass-scale refactoring

CHAPTER 2
Techniques for Scaling AI for Large Codebases

As we look at our AI world of infinite interns applied to mass-scale code refactoring and analysis, we must look at how to supervise our interns more effectively. How can we build guardrails for them to work faster with a higher accuracy rate? In addition, how can we ensure the most efficient, cost-effective AI implementations for working on massive codebases?

One thing to keep in mind is that AI models have knowledge gaps when it comes to large codebases. A model cannot code using a framework it has never been trained on. Additionally, they are not trained on private data, such as a company's codebase, out of the box. Processing codebases with millions of lines of code would overwhelm the model, leading to increased latency and noise that affects accuracy, assuming the model is even capable of managing such a large volume. New and helpful data beyond the prompt must be provided as context to the model in real time for fully understanding and refactoring large codebases.

A long context window for an LLM creates a challenge for the model to find information. The model must sift through a large amount of data to locate relevant information, which can be akin to finding a needle in a haystack. In general, as the context length increases, the model's ability to maintain coherence and relevance diminishes.[1] This is due to the model managing more distractions and irrelevant information, which can cloud the focus on the primary query. A long context window is also a challenge in terms of latency. The processing time for models to analyze and respond to input text increases quadratically. This means it may be better to query a model multiple times with shorter snippets of texts, than to feed it all at once.

What's more, because most LLMs are trained on natural language and source code as text, they do not take into consideration the differences between natural language and code. Code has a unique structure and strict grammar, as well as dependencies and type information that must be deterministically resolved by a compiler—information that could be incredibly useful for AI but is invisible in the text of the source code.

In this chapter, we cover techniques for operationalizing AI models for coding at scale, filling the knowledge gaps and enabling them to be more accurate and efficient. We'll start with the core concept of retrieval-augmented generation (RAG).

Retrieval-Augmented Generation (RAG) and Embeddings

RAG is the method for supplying an AI model with large amounts of information that the model was not trained on to improve its accuracy and reliability. RAG uses embeddings to fetch data from relevant external sources, which helps to guide the model's decision-making process. This retrieval step supplies real-time context to the model, which contains significantly more information than an LLM context window could supply.

1 Patrick Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," *Proceedings of the 34th Conference on Neural Information Processing Systems* (NeurIPS 2020), Vancouver, Canada.

Embeddings are numerical representations of data concepts that AI models can operate on. A data concept can be an image, a word, a document, chunks of a document, or even a method declaration. Since embeddings are vectors of floats, they can be used to do arithmetic. With the basis that embeddings represent entities, if the embeddings of two entities are close to each other numerically, the two entities will be similar. For example, you might think that "love" and "hate" would be far from each other, but they tend to have similar embeddings, as shown in Figure 2-1. They are both emotions that people use to define a relationship in regard to something or someone else.

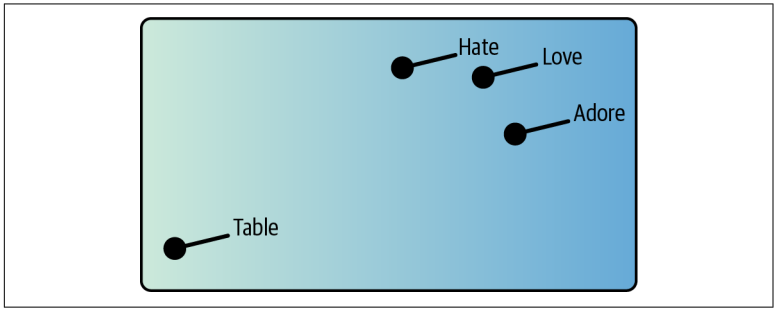

Figure 2-1. RAG embeddings illustrated

RAG operates as a pipeline where you first have a retrieval model that uses embeddings, which is a cheap, quick operation, followed by the more expensive generative model. It's a way to feed high-fidelity information to a model. The high-fidelity information can be from documentation, PDFs—really any text that contains useful information.

A typical RAG process for coding involves:

1. Partitioning files with code and generating embeddings for each partition
2. Storing the embeddings in a vector database
3. Passing relevant partitions based on their embeddings as context to the LLM for generation

Note, however, that this data may still be too broad and overwhelm the model with extraneous information, leading to mixed results containing hallucinations. For mass-scale, automated refactoring and analysis work, you need to provide context that is precise and also representative of the broader codebase. This is an area where an improved code data source, called the Lossless Semantic Tree (LST), comes in handy.

Lossless Semantic Tree (LST) as Data Source for AI

The LST code representation, invented as part of the OpenRewrite open source auto-refactoring project, offers a full-fidelity view of a codebase. It provides rich, semantic-based data, including all dependencies (transitive and direct), type information, and formatting—everything needed to search and transform code programmatically.

The LST is a more complete dataset than the common Abstract Syntax Tree (AST) representation, going beyond syntax parsing to collect more useful information. The LST is produced by guiding the compiler through both syntactic and semantic analysis stages, forcing it to produce full type attribution. This data enables type-aware semantic search, resulting in 100% accurate matching of patterns. Producing an LST also includes going back to the source code and scooping up the formatting and other syntax elements, such as comments, that are typically discarded on the AST. This ensures code changes match the style governing a project.

When using AI models with large codebases, the LST is the most comprehensive data source of a user's codebase—offering important context that can be fed to AI models. Combining RAG-like techniques with the LST means that AI models are supplied with accurate information at the time of query, retrieval, and generation. This technique zeros in on the smallest amount of and most accurate information to augment what the model was trained on, giving it the best chance of classifying the information without hallucinations.

An example of using LSTs as the retrieval part of the RAG pipeline would be where you want to understand all methods that are in use across multiple repositories of code. A program can search the LSTs for all method invocations in a codebase, then make a call to an embedding model to get a vector representation for each method

based on its signature. Figure 2-2 shows a clustering of methods using an embedding model.

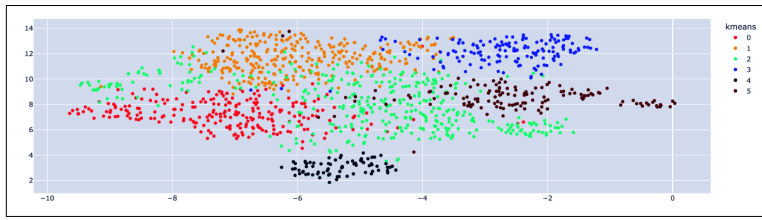

Figure 2-2. Using the LST and an AI embedding model to cluster all method types in a codebase

If a system lacks type attribution (that is, not understanding where a method is declared), it can't determine this signature. These signatures are very useful for finding a method invocation based on a query (such as "adds a content-type header"). The embeddings of the method signatures can be ordered from least to most likely to fulfill the query. This is extremely powerful, since the embeddings can be used to reduce the search space and therefore only search over the most probable methods that do such an action described by a query.

At Moderne, we can aggregate LSTs, enabling rules-based plus AI code refactoring and analysis at mass scale. We also are working with academia, such as Mila - Quebec Artificial Intelligence Institute, to research other ways that LSTs can be used for code ML applications. Mila is one of the world's largest academic research institutes dedicated to deep learning, hosting over 1,200 researchers.

NOTE

The techniques and deployments we cover in this report can be relevant to operationalizing LLMs with any data source. Just keep in mind that the more concise, complete, and accurate the data you feed to the model, the fewer hallucinations and the better the results.

Recipes for Guiding and Validating AI Actions

We use programs called *recipes* to direct AI models and data retrieval for mass-scale code refactoring and analysis work. Recipes, another aspect of OpenRewrite, are rules-based operations for

semantic code search and transformation, with over 2,700 open source recipes available in the OpenRewrite catalog.

Recipes can support AI by:

- Extracting data from the LST (only the parts that need to be evaluated or transformed)
- Feeding the data as text to the LLM as part of the query
- Inserting results of an LLM transformation back into the LST

The Moderne Platform, for example, runs OpenRewrite recipes at scale—walking a codebase (i.e., the prebuilt and cached LSTs) in a deterministic way and calling AI models when needed, as shown in Figure 2-3. When code changes are made, the Moderne Platform produces diffs for developers to review and commit back to source code management (SCM), ensuring models are doing their job with precision.

Chapter 4, "AI Use Cases For Mass-Scale Refactoring and Analysis", shows how this system is put to use.

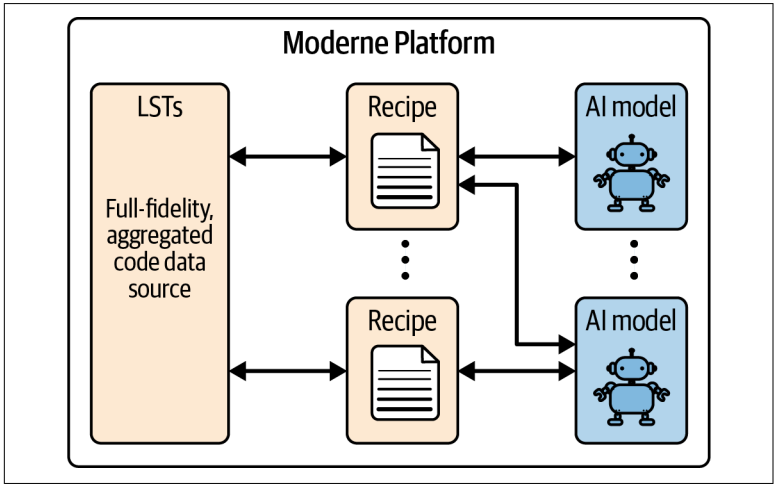

Figure 2-3. Recipes + LSTs + AI models working together on a codebase at scale

LLMs, just like most humans, are bad at math without supporting tools. LLMs generate text sequentially, one token at a time, relying on pattern recognition learned from past data. They might be able to do simple addition and multiplication, in the same way you can memorize a few key operations for some digits. However, once it comes time to do difficult multidigit operations, simply generating one token at a time is not feasible. That's why it's helpful for deployed LLMs to have access to tools, making them more agentic. For example, tools such as a calculator, or in our case the OpenRewrite rules-based recipes, can be supplied.

Now let's dive deeper into LLMs and how to assess them for the task of large scale code refactoring and analysis.

CHAPTER 3

AI Large Language Models: Beyond the Chatbot

When tackling a problem you wish to solve with AI, one of the first questions you should ask yourself is what kind of AI model you should use to solve the problem. An AI model can act as a trained, interactive knowledge base to add to your toolbox. However, that's like saying a dog is simply a four-legged animal that barks. After all, you're likely considering which breed works best with the family. Is the dog trained, and trained on things that matter to you? And what is the right sized dog for your environment?

While a chatbot can be useful for generating code, LLMs can help beyond a mere chatbot experience. There's a wide selection of AI models available today, both private and open source, that are great for coding use cases. The types of models you'll want to use are highly dependent on what your organization is trying to achieve. But you'll also want to consider the training and performance of the model for your use case.

In this chapter, we'll offer you general guidelines on model selection and some of our specific thinking about models regarding implementations for mass-scale automated code refactoring.

AI Models Pretrained and Ready for Adoption

Because of the heavy lift of AI model training, most of us will be "renting" pretrained AI models, which we'll tune with other AI tools and techniques, such as RAG, to refine results.

AI models learn by looking at previous examples, scoring their output, and updating their parameters. The larger a model is, the more data it needs to be able to tune each one of these parameters. Following the Chinchilla scaling laws,[1] it's estimated that you need approximately 20 tokens (i.e., chunks of words) per parameter for training an LLM.

A token is a chunk of text that is approximately three-quarters of a word. So, following Chinchilla scaling laws, a model of 7 billion parameters would need approximately 140 billion tokens for training. To put this in perspective, Wikipedia contains 4.5 billion words, which is approximately 6 billion tokens.

Typically, only companies with access to an extensive amount of GPUs and extremely large datasets are training LLMs today. You likely have heard of or are even using OpenAI's GPT suite, Anthropic's Claude, or Google's Gemini. There also exists a plethora of pretrained open source models with permissive licenses that can be quite effective while keeping costs down, including Google's Gemma, Meta's Llama suite, and Mistral.

Types of LLMs Useful in Coding Use Cases

When considering AI models, it's important to understand the types of models that target different use cases. Here are few types of models to consider:

Classifier models
: These models are used to categorize data into predefined classes as output.

1 Jordan Hoffmann et al., "Training Compute-Optimal Large Language Models," Google DeepMind Technologies, reprinted by *arXiv*, March 29, 2022.

Embedding models
: These models convert data into vector representations for manipulation that can be used for various tasks such as classification, information retrieval, and clustering.

Generative models
: These models create new data samples that resemble a given dataset (such as text or images).

The type of model you want to use depends on the downstream task. The size of the models will vary depending on the task. Classifier and embedding models are smaller, less expensive, and more specialized, while generative models are larger (ranging up to 400 billion parameters), more expensive, and can be considered general purpose.

For instance, a classifier model could be useful for the task of identifying technologies in use from a predefined, relatively small number of options. An embedding model task could be to find duplicates for a certain function to cluster method declarations. And a generative model could be used for the task of writing code, predicting the next token from a set approximately hundreds of thousands in size. This makes it great for prototyping a first draft of a code solution.

We recommend that you ensure that embedding models have been fine-tuned for the specific application for effective utilization. However, this doesn't mean that the model needs to be completely retrained for every new application as long as it is used for the same type of task. While generative models do use embeddings internally, these embeddings are not optimized for tasks such as classification or retrieval and, thus, won't perform well if used directly for such purposes.

Navigating Open Source LLMs

Open source is a perhaps surprisingly great place to find efficient, cost-effective, pretrained models for all use cases. It's also important to note that open source models can be run locally or on your servers without having to make an external call. If someone else is hosting the model, this can get expensive, and you'll have to pay by token or usage.

However, choosing an open source model can be a daunting task. Hugging Face, a French-American company, hosts open source models and datasets and maintains a multitude of ML libraries notably used for generation, training, and evaluation. Any research group or company can host their trained models on Hugging Face. In fact, at the time of writing, there are over 350,000 models hosted on their platform.

Open source models provide the model's weights (i.e., the trained parameters), and you may also get access to the data they were trained on—or at least the knowledge cut-off date. However, your mileage may vary, and some groups will only share the weights. The training data and the model's architecture can also heavily affect the performance of the model on any given task.

To help you navigate to the right models for your needs, Hugging Face hosts various model leaderboards. For example, you can use the Massive Text Embedding Benchmark (MTEB) leaderboard to gather a list of candidate embedding models.

While leaderboards offer valuable insights into various models, it's crucial to recognize that the scores presented might not accurately represent the true capabilities of the models. This discrepancy arises primarily from two factors:

- Benchmarking is just one metric, similar to how a student's test score may not fully capture their knowledge or skills.
- Data leakage can occur where a model may have been inadvertently trained on the test dataset, enabling the model to artificially appear to perform better.

Beyond using model benchmarks, it's also very important to assess various models for your particular domain, including both general-purpose ones and those fine-tuned—especially if you're dealing with a highly specialized area. For instance, when you're evaluating a model for a task, you can handwrite a few key examples of what you wish the model to do. The evaluation can be more open, say, if you ask a model to generate 10 images of a cat, and then you can grade them or assess them manually. Or you can do something similar to a unit test and grade the results automatically. For example, if we worked on evaluating what model to use for retrieving relevant documents based on a query, you might want to handwrite

a couple queries and check whether the model correctly retrieves the documents you would expect in this case.

Deploying AI Models on CPUs

We posit that CPUs for the *deployment* of LLMs—when models are running in production—is not only possible but can be preferred for many use cases, especially for text-based/coding settings. GPUs, with their parallel processing capabilities, have become the preferred choice for *training* LLMs, and they are often chosen as the default for LLM deployment. However, deployment typically demands less computational power than for training, making CPU deployment quite feasible.

GPUs are used for training LLMs due to their capacity to efficiently manage the computational demands of gradient tracking, a fundamental aspect of neural network learning. Even if training a smaller 7B (i.e., 7 billion) parameter model, a lot of computational power is needed. Indeed, to train Meta's Llama with 7B parameters, it took 82,432 GPU hours. Training such an LLM on a CPU would be impractical since transformers, the architecture behind LLMs, were inherently designed for parallel processing.

Choosing CPUs for LLM deployment can be advantageous, particularly since even lower-tier CPUs frequently come with more random-access memory (RAM) compared to their GPU counterparts. This is useful because more RAM means more space for a model. Moreover, CPU instances are substantially more cost-effective than GPU ones, which addresses the growing concern of cost for users and operators alike. Although a GPU-hosted model may operate more quickly in terms of tokens per second, accommodating the same model might necessitate a larger, and therefore more expensive, GPU due to RAM requirements.

In addition, you may need to consider operational requirements for running LLMs. For example, as we set out to integrate AI into the Moderne Platform, we needed an LLM implementation that could operate within the sovereignty of our platform, scale to meet the needs of multirepository refactoring work, and be very cost-effective.

In our early explorations, we evaluated whether CPUs alone could support the operational needs of the LLMs for our platform. We rigorously tested the limits of CPU processing power, focusing on generation speed, a factor potentially constrained by the choice between CPU and GPU resources. For our use cases dealing with text and code, we were pleased with the resulting speed, but we understand this might not be the case if dealing with video, audio, or graphics, which require more computation.

Some additional things to keep in mind for deploying AI models on CPUs:

Using supporting tools
: LLM libraries or frameworks make it easier to deploy models without crafting routines from scratch, offering the necessary tools to streamline AI integration and operation. Your choice of library or framework will be influenced by multiple things, including simplicity of the library, features (such as servers, chaining, batching, or metrics), speed, and which model architecture they support. A few libraries and frameworks come to mind: LangChain, vLLM, llama.cpp (and its bindings in multiple programming languages), and Hugging Face.

Improving performance
: Using a library to compile quantized models, a version of an LLM modified to use lower precision (i.e., 8-bit integers instead of floating-point numbers) for calculations and storage, can decrease the model's size and speed up its operation. Some libraries may also be built such that you can employ acceleration techniques, including Advanced Vector Extensions (AVX) to improve latency. Note that the effectiveness of these optimizations is highly dependent on the specific instances used.

Deploying models as sidecars
: You can wrap your model into an API and deploy alongside regular microservices as a sidecar service. The sidecar process can be reached by sending HTTP requests. You may want to code this using a library such as Gradio (which also provides a simple UI if needed). By having the models loaded as a sidecar, you're also certain the data in the requests stays within the virtual machine where it is used.

A great exercise to quickly discover the pain points for CPU deployment is to add a simple Python script that always predicts 1, to mimic a very "dumb" and simple model. This simple test will make you ask yourself multiple questions such as:

- Do you want the model (or script in this case) running in memory at all times?
- Do you want the model to load up each time there's a query?
- Do you want a cache?
- Can you run Python or C on your setup?

If you can't answer those questions for a simple one-line method, it will be much harder to answer them for a large multigigabyte model.

In Chapter 4, we pull together how to use the techniques and AI models together to do real automated refactoring and analysis work.

CHAPTER 4

AI Use Cases For Mass-Scale Refactoring and Analysis

At Moderne, our mission is to help our customers understand and transform their large, unwieldy codebases at mass scale. Whether it's migrating frameworks, modernizing for the cloud, remediating security vulnerabilities, or performing impact analyses, these are tasks that typically scale beyond individual developers working in single repositories.

In our platform, we have brought together AI models, LST artifacts, and OpenRewrite rules-based recipes to automate code refactoring and analysis work at mass scale. This chapter details four of our current use cases:

FIND
: Implementing an AI-based search, enabling users to quickly find recipes that can help them

ASSIST
: Using rules-based recipes to ask AI to help with simple operations, leveraging specific context from the LSTs

DIAGNOSE
: Using AI models, working with LST data, to recommend recipes based on what the code needs

BUILD
: Using GenAI to assist in OpenRewrite recipe development to enable automated code fixes at mass scale

These use cases illustrate how to put the techniques and technologies we've explained earlier into practice—from building RAG-like pipelines to improving AI assistant responses.

Using AI to Search a Recipe Catalog (FIND)

The objective of human-computer interaction (HCI) is to enhance usability, which encompasses the development of an intuitive platform, as well as features that assist users in maximizing their interaction with an interface. The latest AI-based interfaces enable users to interact with computers using natural language, further simplifying and enhancing the ease of use.

This section will walk through how we upgraded the search function of the Moderne Platform with an AI-based interface to ensure that our users could more easily find the OpenRewrite recipe they were looking for, as well as discover other relevant search and refactoring recipes to use across their large codebases. This use case demonstrates a multistep pipeline using two different embedding models.

Problem: Failing Exact Match Recipe Searches

The first-generation search in the Moderne Platform used the Apache Lucene library (i.e., string-matching) and required exact match keywords, which often prevented users from finding the desired recipe. For example, a search for "Find method invocations" wouldn't return the recipe titled "Find method usages," despite the terms "invocations" and "usages" being synonymous in this context.

We needed a way for our users to search the recipe catalog of thousands of recipes based on concepts instead of exact words, which led us to an AI-based solution. This function needed to be fast and operationally inexpensive (both in time and financially), allowing us to scale as the recipe catalog grows. We also wanted to use an open source solution where we had more control over security for air-gapped and restricted environments, so using OpenAI's API or other providers was out of the question.

Solution: Build an AI-Powered Search Engine

We built an efficient, semantic-based AI-powered search engine that allows users to quickly find the recipe they were looking for by using word representations and concepts. This is made possible through

the use of embeddings, those vector representations of concepts that allow for searching based on the distance between vectors.

The reality is that most recipes will be poor overall matches for a given query. Our solution incorporates two embedding models to get to a relevant subset. The first model conducts a preliminary sweep of all recipes, while the second, more sophisticated but slower model, meticulously refines the results of the initial pass, as shown in Figure 4-1.

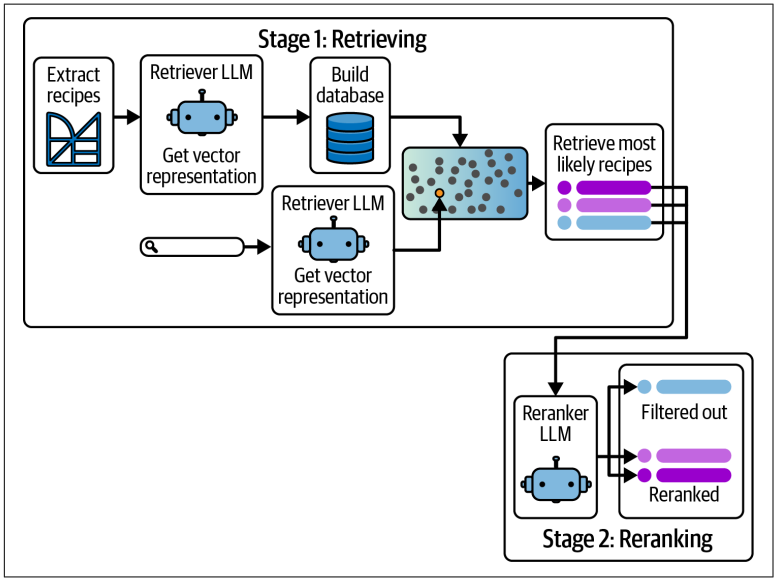

Figure 4-1. Two-stage AI pipeline for OpenRewrite recipe search

Stage 1: Retrieving initial search results

There are many different options of retrievers you can use for returning a response from a query. We evaluated three different ones for our use case:

Regular retriever
: Native to the vector database, this simple retriever is typically based on the distance between embeddings, and the results are based on the closest elements to the query's embedding.

Multiquery retriever
: This uses a generation model (such as OpenAI GPT or Code Llama) to produce a range of similar queries, rather than just

the one user-provided query, enhancing the diversity and comprehensiveness of the set of results retrieved.

Ensemble retriever
: This technique fetches search results using multiple retrievers, bringing the best of both worlds of embedding-based search and keyword matching.

We found that the regular retriever worked fine for our use case and did not add overhead that was unsupportable. We ended up using a BAAI retrieval model, specifically bge-small-en-v1.5.

The multiquery retriever was not fit for many reasons. It comes with a significant computational cost, which in turn leads to higher latency. To get useful queries, you need an LLM that is good at generating, such as using an OpenAI model or an OSS model such as Mistral and Code Llama. With the ensemble retriever, we did not see a significant performance difference for our use case beyond what we were getting with the regular retriever.

Retrieval is quick at search time because all the embeddings, except for the query, are already computed when initializing the recipe database at startup. This means that all you have to do is compare the distance between a recipe's embedding and the query's embedding.

Stage 2: Reranking to finesse AI search results

While the retriever has to be efficient for large sets of recipes, it might return irrelevant candidates. Also, the order in which even the top few recipes are returned in practice can be counterintuitive.

Reranking is a technique to finesse your search results, up-leveling the more likely candidates and providing a more intuitive ordering. Reranking passes the query and a recipe retrieved in the first stage and passes them simultaneously to the transformer neural network, which then outputs a single score between 0 and 1, indicating how relevant the recipe is for the query. Using a reranking model that uses a cross encoder enables the model to have access to the query and recipe text together, meaning it can better grasp the subtleties of the query and recipe together.

But this also means that reranker scores can't be calculated in advance of the query coming in. While the qualitative performance of the reranker is better than a regular retriever, scoring thousands

of query/recipe pairs (i.e., for every recipe in the catalog) would be prohibitively expensive. In addition to the lack of cacheability, reranker models are simply larger than retriever models. In our case, we use another model from BAAI called bge-reranker-base, which is 1.11 GB. This is significantly larger than the 134 MB for our retrieval model.

Combining the stages into a pipeline

Now let's look at the pipeline in operation, as shown in Figure 4-1. By taking just the top-k from the retriever step, the obvious poor recipe matches are swept away in Stage 1. We then pass these top-k recipes to the reranker model in Stage 2. The reranker scores the recipes, discards the recipes that don't meet a predetermined threshold, and orders the remaining set of recipes based on their respective scores—without any consideration given to the order from the original retrieval step.

This two-step pipeline, using two models of varying sizes, yields the best results both for performance and accuracy.

Using AI to Support Rules-Based Refactoring at Scale (ASSIST)

We talked about using AI as another tool in your toolbox. To that end, LLMs can become a useful assistant to OpenRewrite recipes, adding additional analysis and transformation options for codebases. Connecting rules-based techniques and AI-based techniques are a powerful combination that can help reconcile the need for "human in the loop" for GenAI.

This section details how the Moderne Platform provides the framework for a recipe to walk through a large codebase (aggregated LSTs) in a deterministic way, calling the AI model only when needed. This not only safeguards and focuses the model to precise places in the code, but also makes models more efficient, as they are only used when needed.

Problem: Misencoded French Characters in Code

One of our customers came to us with a problem ripe for an automated fix that leverages AI. Their older code had gone through

multiple stages of character encoding transformation through the years, leading to misencoded French characters being unrenderable.

French characters can have accents such as *é* or *è* or might be even *ç* or *œ*. These special French characters could be found in comments, Javadocs, and basically anywhere you would have textual data in their codebase. ASCII is a 7-bit character encoding standard that was designed primarily for the English alphabet, supporting a total of 128 characters (0–127). This set includes letters, digits, punctuation marks, and control characters but does not include accented characters such as *é* or other non-English characters. When a character has an encoding issue, then it will be replaced by ? or .

Other ramifications for this customer were that the misencoded French characters in Javadoc comments caused the Javadoc compiler itself to fail, which means consumers of that code did not have ready access to documentation on the APIs they were using. It also hindered automatic test generation.

Solution: Find and Fix Misencoded Characters

With the Moderne Platform and a little help from AI, we were able to quickly solve this problem. We decided to use AI to figure out what the words were supposed to be and to fill in the appropriate modern UTF-8 characters.

We wrote an OpenRewrite recipe to use AI to fix the misencoded comments and Javadocs. The recipe walks through the codebase (i.e., the LSTs) until it finds either a comment or a Javadoc. It then sends the text in the comment or Javadoc to one LLM sidecar that determines whether it's French text. If it is, that text is then sent to another sidecar to generate a predicted fix for the misencoded text. We were able to integrate two models for a transformation that can be used at scale. By having a recipe that guides the changes, our customer could be certain that the changes would only be on comments or Javadocs, essentially safeguarding their code from any unnecessary change.

We focused on OSS-specialized LLMs that can run on CPUs for maximum operational security and efficiency within the Moderne Platform. This enabled us to provision the models on the same CPU-based worker nodes where the recipes were manipulating the LST code representations.

On the Moderne Platform, models run as a Python-based sidecar using the Gradio Python library. Each Python sidecar hosts a different model, allowing for a variety of tasks to be performed. A recipe then has several tools in its toolbox, and different recipes can also use the same sidecar.

When a recipe is running on a worker, it can search LSTs for the necessary data, pass it to the appropriate LLM, and receive a response. The LST only sends to the model the parts that need to be evaluated or transformed. The recipe then inserts the LLM response back into LST. The Moderne Platform produces diffs for developers to review and commit back to source code management (SCM), ensuring models are doing their job with precision. See the process in Figure 4-2.

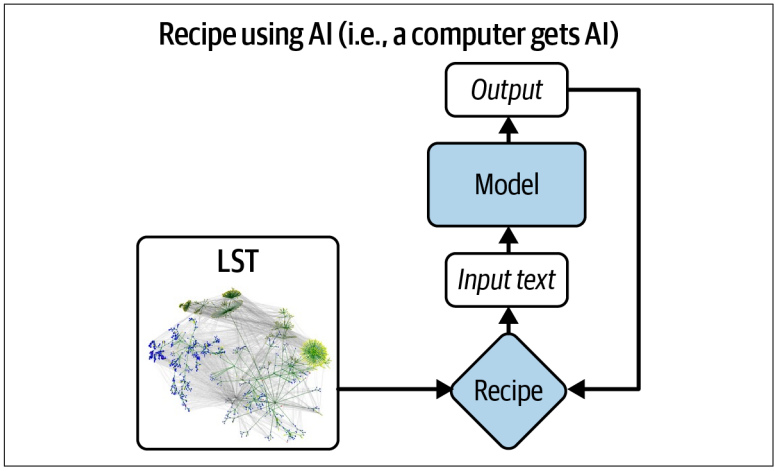

Figure 4-2. Recipes searching LSTs to feed the AI model useful context

Fixing misencoded French text in a codebase is challenging for both purely rules-based systems and LLMs alone. Rules-based systems can't identify the natural language in comments, and LLMs struggle to identify comments or other code syntax/semantic data. By using recipes to guide and focus the LLM, we achieve more predictable and reliable results.

We tested it out on ChatGPT alone and found too many instances where the LLM failed to fix the misencoded comments, as shown in Figure 4-3. For example, it fails to understand that "this" represents the keyword instead of the determinant. It also fixes "class" to "classe," which could be frustrating for developers.

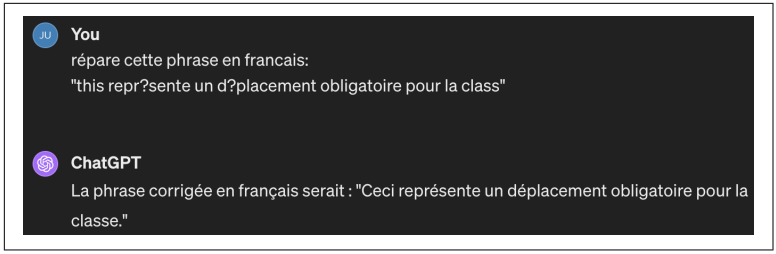

Figure 4-3. Chatbots can't repair misencoded French

Using AI to Discover Problems in a Codebase (DIAGNOSE)

We've found that it's really useful for AI chatbots to have access to tools, something we call "AI gets a computer." These tools, such as a calculator, enable AI models to behave more autonomously, mimicking traits of an agent that can interact with its environment, by limiting hallucinations and significantly improving the quality of their output.

This section details the use case of how the Moderne Platform helps LLMs sample our customers' large codebases, diagnose issues in the code, and recommend applicable fixes (i.e., recipes).

Problem: What's Wrong with My Code?

In large enterprise codebases spanning millions if not billions of lines of code, the old adage of "you don't know what you don't know" is so true. Our customers may not be aware of repositories with older framework versions that are reaching their end of life or whether they've missed a security vulnerability, such as Log4Shell, somewhere in the code.

How could we help our customers cut through the noise to discover issues in their codebase, as well as fix the issues for them?

Solution: AI Sampling a Codebase and Recommending Recipes That Can Fix Problems

We developed a recommendations tool designed to diagnose issues in a codebase and recommend recipe fixes specific to that codebase. This solution has three main stages (shown in Figure 4-4):

A. A recipe extracts methods or classes from the codebase and feeds them to a model to produce embeddings. We cluster embeddings using k-means, then select samples from each cluster to give to a generative model to make recommendations.

B. Using these recommendations, our in-house recipe search (see the FIND use case) discovers recipes that can do the modernization, fix, or migration required. This can validate the AI recommendation.

C. Prove the efficacy of the AI recommendation by automatically running the safe and tested OpenRewrite recipes on the code; only recipes that actually produce changes are shown.

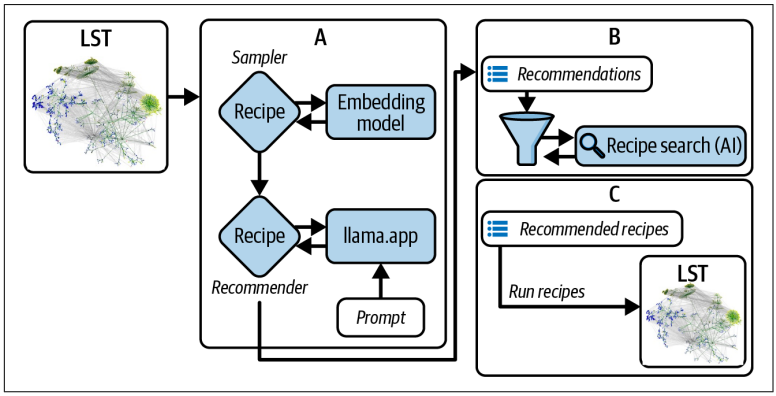

Figure 4-4. AI pipeline used for recommending recipes

It's important to use recipes for validation as the last step to making such huge transformations. When you look at the second stage of the recommendation pipeline, hallucinations were something to be worried about. An example of a hallucination we saw when prompting the generative model to generate recommendations for modernization was when the model said to upgrade Java version because "the `Boolean` class was introduced in Java 5, but it has been deprecated in favor of the `boolean` primitive type." This is absolutely false. If you were not a Java coder or very knowledgeable about versions, you might not know that both the `Boolean` and the primitive `boolean` are both still accurate ways to represent a boolean in Java 8. So what do we do with recommendations that may contain hallucinations?

Thankfully, at Moderne, we do have a set of recipes that we trust and know are accurate. The next step for us is simple: search for recipes based on the recommendation. Therefore, if there are any recipes that deal with the `Boolean` class or Java migrations, we can run them to see if changes exist. The accurate recipe is the only thing that can make a change on the source code.

Using AI to Develop OpenRewrite Recipes (BUILD)

Given the nondeterministic nature of GenAI, as well as the fact that the coding assistant can only change a single file at a time, large-scale code remediation efforts are not feasible with AI assistants alone. Even a slight risk of error is unacceptable, and it doesn't scale to the human review necessary.

However, AI assistants can be helpful in writing the OpenRewrite recipes that can then be run across a large codebase to stamp out the change—cookie-cutter style. Recipes and their unit tests are highly structured. They can be an imperative program or a declarative composite recipe expressed in YAML, or a declarative template style recipe ("refaster style"), specifying before and after templates for code transformation.

Recipes are tested and reviewed by a human first and then can be used to produce perfect cookies (i.e., code changes) at scale. Involving a human in the recipe creation process ensures reliability and predictability in the changes implemented by the recipe.

This section walks through how we can use AI during the writing process for OpenRewrite recipes. LLMs are great at writing repetitive but necessary parts of code or sketching a possible solution (proposed code) to a problem, such as a large-scale migration. But the actual application of changes must remain deterministic to be effective at scale.

Problem: Generating OpenRewrite Migration Recipes Faster

Modern applications are assembled with as much as 90% of their code coming from dependencies. These dependencies are outside of business control and evolve at their own pace, remediating

vulnerabilities and developing new capabilities—in effect changing API signatures or deprecating and deleting APIs.

OpenRewrite makes it easy to provide a refactoring recipe to lift consumers of APIs to a new library version. This enables framework and library authors to freely make changes without worrying about affecting their consumers because upgrades are now automated. However, given that the ecosystem of OSS and third-party vendor libraries is so broad, how do we as a community catch up and keep producing refactoring recipes faster?

Solution: Leveraging GenAI to Write Recipes

At Moderne, our developers write recipes with assistance from Copilot, just like any other program, with the human reviewing each suggestion and supervising the production of the correct recipe for refactoring or remediation. But it is still the human who needs to go read release notes and decide what recipes to build.

Determining what has changed frequently is not straightforward, relying on the quality of release notes, security reports, and other documentation. However, when such information is available, it can be leveraged by the AI assistant to full effect: inserted into an IDE as a comment prefixing a specific recipe declaration or in a ChatGPT prompt with a query to create a declarative OpenRewrite migration from the release notes.

OpenRewrite declarative YAML format is perfect for capturing the majority of migration changes, such as API and dependency version changes. We still rely on developers to correct the definition, but in our experiments, the time saved in having AI write out the boilerplate code is significant. We recommend preprocessing the release notes removing bug fixes, contributors, and other extraneous information, focusing only on what has changed prior to feeding it to the AI assistant—focusing AI on just the relevant information.

We're also working to leverage AI in improving the recipe feedback cycle. Determining what has changed between versions of libraries is not a straightforward task. For example, understanding the high-level Spring framework release notes is not enough. If you use Kafka, you will likely also need to migrate Kafka libraries, and if you use AWS cloud APIs, you will also need to migrate that. Unfortunately, the lack of recipe coverage usually shows up during the compilation of repos after applying a recipe. With AI, we can

help people understand the additional recipe coverage they need and fill the gaps more quickly—such as finding existing recipes or understanding the changes necessary to accomplish an upgrade.

The Single File Problem

Given that the AI assistant can create suggestions in only one file at a time, in order to help it generate the whole migration from release notes, we are changing the structure of OpenRewrite recipes, allowing multiples of them (declarative and imperative) to be defined in a single file.

Before AI assistants came into the picture, framework authors had a stylistic choice to make. Either they encouraged decentralized use of their API across many files (e.g., Spring @RequestMapping annotations spread across many files) or encouraged centralizing use in one or a small set of files (e.g., Play Framework's Routing definitions being centrally positioned). After the advent of AI assistants, there is a qualitative difference. As a framework author, if you want engineers to harvest the power of AI while working with your framework, it's best to provide at least the option of a centralized approach. This allows the AI, editing a single file, to implement larger swaths of the target application quickly.

Recognizing this shift, we've made changes to OpenRewrite to support writing multiple-step recipes, not just in YAML, but in source code as well. This allows an AI code assistant to help us write more of the recipe at one time, faster.

CHAPTER 5
Considerations Using AI with Enterprise Codebases

We recognize that we are all in a world of experimentation with AI. Many development teams are gravitating toward using AI assistants or chatbots to help generate code. Even with the hallucinations, this is an important application of AI for code.

However, AI assistants are not built for mass-scale code changes—many individual developers with their AI assistants operating on one file, one cursor position at a time. This by definition limits the problem space and requires each developer to review the output. To apply AI to maintaining and securing large-scale, proprietary codebases, you need some different approaches as we've shared in this book, with the end goal of making accurate changes to codebases at scale.

LLMs need to be augmented with the code data via various techniques. RAG is one that's popular today, but others may appear in the future. In addition, instead of embedding code as text like any other business document, you can do one better if you take into consideration the structured nature of the code. When you understand the code and are able to extract relevant methods or classes from code, the better you can guide the model with just enough relevant information. These approaches are useful whether you decide to use LLMs via API or plan to deploy and operationalize them from OSS.

As you take your next steps with AI in your software development organizations, we want to leave you with our top considerations for using it at scale:

Trust
: It's hard to trust a system of infinite interns (i.e., AI assistants) working on your codebase, especially for making code changes at mass scale. Implementing guardrails, such as using Open Rewrite recipes with LSTs, can improve the accuracy of models and facilitate the adoption of AI for working on large, proprietary codebases.

Security of your IP
: Companies with tight security restrictions must decide between self-hosting an LLM or relying on third-party hosting. A third-party shared instance can be a deal-breaker due to security concerns with their code IP, while a dedicated, private instance can provide a more secure option. For example, Moderne self-hosts LLMs in order to provide a single, private instance of AI models for use in mass scale auto-refactoring.

Cost
: Whether it is because of computational demands or from using third parties to host LLMs, AI can be costly. Balancing useful AI outputs while limiting costs involves techniques such as using recipes and RAG, which reduce overhead and reliance on expensive models.

Right tool for the right job
: Don't use a chainsaw when you need a butter knife. Remember that bigger generative models are not good for embeddings or classification tasks! Start with popular benchmarks to find the best models for the job, then try out the models yourself and see the outputs. Given the variety and the power of OSS LLMs, they are a viable option for large enterprises.

At Moderne, we believe that the combination of AI models, rules-based recipes, and the full-fidelity LST code representation can deliver more accurate, efficient, and cost-effective mass-scale code changes for enterprise organizations.

About the Authors

Justine Gehring is a researcher in the field of machine learning for code (ML4Code) and Graph Neural Networks (GNNs). She obtained her master's from McGill and Mila where her research focused on generating code under challenging circumstances such as library-specific code. Presently, Justine is a research engineer at Moderne, focusing on leveraging AI for large-scale code refactoring and impact analysis. She also oversees the partnership with Mila.

Olga Kundzich is cofounder and CTO at Moderne, delivering automated code refactoring and analysis solutions that operate at mass scale. She has extensive experience building enterprise software solutions. Previously, she worked as a technical product manager at Pivotal focused on application delivery and management solutions (e.g., Spinnaker), and has been a lead software engineer and manager at Dell EMC, working closely with enterprise users on implementing data protection practices. Olga is coauthor of the O'Reilly report *Automated Code Remediation*.

Pat Johnson is head of marketing for Moderne. She has published and presented on software development, DevOps, and cloud computing topics, with a focus on improving the developer and operator experience alike. Previously, she has worked as portfolio marketing lead for the Tanzu platform at VMware and product marketing manager for CI/CD solutions at both Pivotal and CA Technologies. Pat is coauthor of the O'Reilly report *Automated Code Remediation*.