Chapter 15. The AI-Powered Software Engineer

> It is not the strongest of the species that survives, nor the most intelligent, but the one most responsive to change.
>
> Charles Darwin

In 1801, Joseph Marie Jacquard invented a loom controlled by punch cards that could weave intricate patterns automatically. Professional weavers watched with alarm as this new machine replicated work that once required years of practice. Many predicted that this would be the end of their craft as they knew it.

Yet something unexpected happened. Rather than eliminating weavers, the Jacquard loom transformed them. Skilled artisans became pattern designers, machine operators, and textile engineers. The most successful weavers were those who understood both the traditional craft and the new technology. Production soared, creating entirely new roles that hadn't existed before.

Fast-forward two hundred years, and we find ourselves in a similar situation. AI is writing code, fixing bugs, and designing entire systems. Developers are having those same "Am I about to be replaced?" thoughts that weavers had back then. But here's the thing: history tells us it doesn't usually work out that way. This pattern has repeated itself with every major technological breakthrough, and each time, the people who adapted came out ahead.

The Jacquard loom didn't put skilled weavers out of work. Instead, it gave them a superpower: the ability to produce far more in the same amount of time. A weaver who once took days to create a complex pattern could now produce multiple pieces in a single day. The same fundamental shift is happening with artificial intelligence (AI) for developers. You can now build applications, debug code, and solve problems much faster than ever before.

So where does that leave developers like you today? If you can learn to master the fundamentals of software engineering and learn to leverage AI as your pair programmer, you will be in high demand, not obsolete. Your ability to learn, solve problems, and adapt are your greatest strengths, and as long as you continue developing these skills, you will remain relevant. In the next section, you will learn what AI is by breaking down some of the concepts you'll encounter as a developer.

What Is AI Really?

While AI has seemingly worked its way into every conversation we have today, it isn't new. Computer scientists have been chasing the dream of AI since the 1950s, when computers filled entire rooms and had less processing power than the phone in your pocket. Research in AI has progressed in waves of excitement followed by "AI winters" when funding for general-purpose AI dried up. Yet, practical AI never disappeared. Behind the scenes, AI quietly powered things like fraud detection, image recognition, and search engines. Then, in November 2022, OpenAI released ChatGPT to the public. Within five days, it had a million users. Within two months, 100 million. Suddenly, AI wasn't just a research curiosity but a tool anyone could harness.

When it comes to answering the question "What is AI?", there isn't a simple answer because it is such a broad discipline. *AI* is an umbrella term that covers everything from simple rule-based systems to neural networks that attempt to mimic how the human brain processes information. To keep this relevant to software developers, we'll focus on a specific set of techniques that have proven effective at solving some very interesting problems.

In this section, we'll make AI more approachable by demystifying some of the key terminology that might seem intimidating. You'll learn where AI excels and where it falls short. Finally, we'll explore what AI can specifically do for you as a software engineer.

Demystifying AI Terminology

Learning a new technology can often feel daunting, particularly when it comes to mastering all the terminology that can be found in a new domain. In AI, you'll encounter terms like machine learning (ML), deep learning, generative AI (GenAI), and large language models (LLMs), depicted in Figure 15-1. At its core, modern AI teaches computers to recognize patterns or probabilities and select the statistically most likely answer, unlike traditional programming, where you write step-by-step instructions.

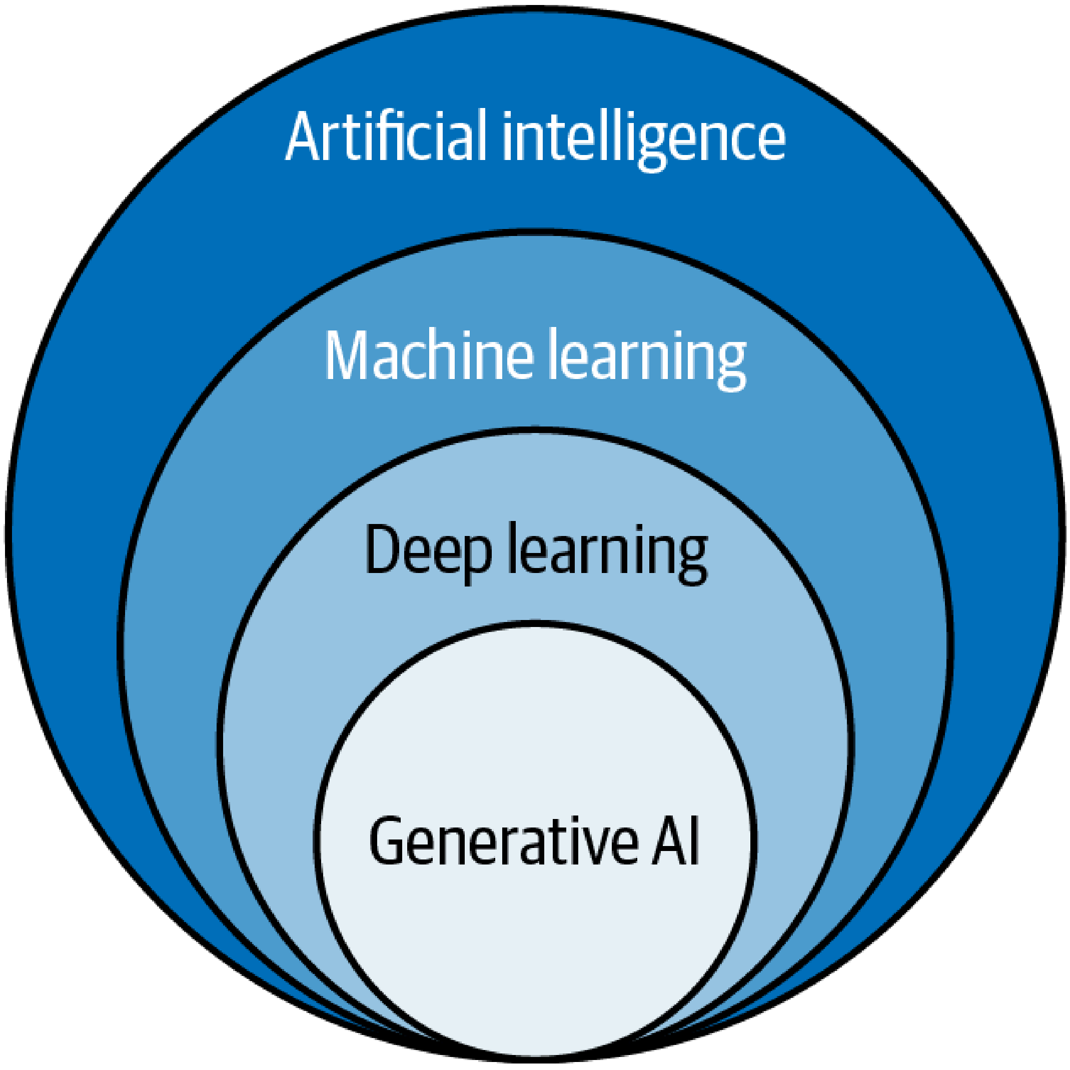

Figure 15-1. The relationships among AI, machine learning, deep learning, and generative AI

As a software engineer, you don't need to become an AI researcher, but understanding these core concepts will help you integrate AI capabilities into your applications and communicate effectively with data scientists and AI engineers on your team.

Machine learning

Machine learning (*ML*) is the foundation of modern AI. Instead of explicitly programming every decision, ML algorithms learn patterns from data. The classic example is facial recognition software. You don't write code to describe every possible face. Instead, you feed the system thousands of face images, and it learns to identify the patterns that make each face unique.

In traditional programming you follow a model of "input + program = output" by writing explicit rules. In ML you reverse this to "input + output = program" by providing examples of inputs and their correct outputs. The algorithms will then figure out the rules automatically.

This approach works out particularly well for problems that are easy for humans to reconcile but harder for us to program explicitly. Imagine you were tasked with detecting which emails should be classified as spam. Just by opening up and reading through the email, your knowledge and experience of seeing thousands of spam emails would allow you to quickly identify which ones are spam.

But how would you write code to identify these patterns? You would need to set up rules for suspicious phrases, sender patterns, formatting quirks, or thousands of other factors. Even after setting up all of these rules, spammers would constantly evolve their tactics, allowing them to bypass your rules. ML avoids this complexity by learning from the examples of spam and legitimate emails, automatically adapting as new patterns are identified.

For developers, ML typically shows up in three main types of problems:

Classification
: Determining which category something belongs to. Think of it as answering "What is this?" Examples include spam detection (spam versus legitimate), medical diagnosis (disease versus healthy), or content moderation (appropriate versus inappropriate). The algorithm learns to recognize patterns that distinguish between categories.

Regression
: Predicting numerical values. Instead of categories, you're asking, "How much?" or "How many?" Examples include predicting house prices based on location and features, estimating delivery times, or forecasting sales numbers. The algorithm learns relationships between input features and numerical outcomes.

Clustering
: Grouping similar items together without knowing the groups beforehand. This answers "What natural groupings exist?" Examples include customer segmentation (finding different buyer personas), organizing large document collections, or identifying user behavior patterns. Unlike classification, you don't tell the algorithm what groups to look for: it discovers them.

As a developer, you'll most commonly use ML through APIs and pretrained models rather than building algorithms from scratch. Here's how you might integrate ML into a typical application:

```
// Using a pre-trained ML model through an API
public class EmailClassifier {
    public boolean isSpam(String emailContent) {
        MLPrediction prediction = mlService.classify(
            emailContent,
            "spam-detection-model"
        );

        return prediction.getConfidence() > 0.85
            && prediction.getLabel().equals("spam");
    }
}
```

Deep learning

Deep learning is a subset of ML that uses neural networks with multiple layers, hence "deep." But what is a neural network? Think of it as a simplified model inspired by how the human brain processes information. Your brain has interconnected neurons (approximately 86 billion) that pass signals to one another. A neural network contains nodes that mimic this functionality (artificial neurons) and passes data through those connections.

In a neural network, the layers act as a processing layer, where each one is assigned a specific task. In a deep neural network, you might have an input layer that receives that raw data and several hidden layers that process and transform that data. When the layers are done processing the data, an output layer produces a final result. These networks contain multiple layers (typically, three or more make it "deep"), which allows it to learn complex patterns.

These networks are really good at finding complex patterns in data. If traditional ML teaches a computer to recognize patterns, deep learning gives it the ability to capture nuance and context. In facial recognition, the first layer might detect simple edges and lines, the next layer might combine those to recognize basic shapes, and deeper layers could identify more complex features like eyes or facial structures. The final layer puts it all together to classify the entire image as "Dan" or "Nate."

As a developer, you'll most often interact with deep learning through pretrained models rather than building neural networks from scratch. These are models that have already been trained on massive datasets and can be integrated into your applications. Let's look at a practical example of how you might use a pretrained image recognition model in a Java application:

```
// Using a pretrained deep learning model
public class ImageClassifier {
    public String identifyImage(byte[] imageData) {
        DeepLearningModel model = ModelLoader.load("face-recognition-v1");
        return model.classify(imageData);
    }
}
```

Generative AI

Generative AI (*GenAI*) represents a fundamental shift in what AI can do. While traditional AI analyzes and categorizes existing data, GenAI creates new content that didn't exist before. If ML is like teaching a computer to recognize what a good painting looks like, GenAI is teaching it to create new original paintings. This is possible because the AI models are trained on massive amounts of data across multiple modalities, all created by humans. The models learn the patterns, styles, and structures of how humans create, then use those patterns to generate new content that feels human like.

GenAI works across multiple types of content, each opening new possibilities for developers:

Text and code
: AI can write articles, documentation, code functions, and even entire applications based on descriptions. This includes everything from generating user-friendly error messages to creating realistic test data.

Images
: Create custom artwork, generate icons that match your app's style, produce marketing visuals, or create placeholder images that relate to your content. AI understands concepts like "a friendly robot icon in a modern flat design style."

Audio
: Generate natural-sounding speech for accessibility features, create background music for applications, or produce sound effects.

Video
: Produce demonstration videos, create animated explanations of complex concepts, or generate marketing content. AI can understand requests like "create a 30-second video showing how to use this mobile app feature."

Here's an example of how you might use GenAI in your development workflow:

```
// Using generative AI to create test data
public class TestDataGenerator {

   private final ChatClient chatClient;
```

```
    public TestDataGenerator(ChatClient.Builder builder) {
        this.chatClient = builder.build();
    }

    public List<User> generateTestUsers(int count) {
        var userList = chatClient.prompt()
                .user( (u) -> {
                    var prompt = """
                        Generate {n} realistic user profiles with
                        name, email, age, and interests""";
                    u.text(prompt).param("n", count);
                })
                .call()
                .entity(UserList.class);

        return userList.users();
    }
}
```

Note

While GenAI creates new content, remember to validate its output. Generated code should be tested, generated data should be reviewed, and any user-facing content should be checked for accuracy and appropriateness.

While most software developers aren't directly involved in developing ML algorithms or building foundational language models, you regularly interact with AI technologies through APIs and services. Understanding these basics helps you build better applications and work well with AI experts on your teams.

Large language models

Large language models (*LLMs*) combine the concepts we've discussed in this chapter so far. They use deep learning neural networks and are a leading type of GenAI. These models derive patterns from vast amounts of data to understand and generate human-like content. ChatGPT, Claude, and GitHub Copilot are all powered by LLMs.

What makes LLMs particularly powerful is their ability to understand context and nuance in human language. Unlike earlier AI that might recognize keywords, LLMs can comprehend meaning, maintain conversations, and even understand implied information. This is why ChatGPT can help you debug code, Claude can explain complex concepts, and GitHub Copilot can suggest relevant code completions.

LLMs excel at tasks involving language understanding and generation. Here's a practical example that shows how you might use an LLM to automatically generate documentation. This example takes in code as input and generates JavaDoc comments, which document the purpose and usage of code for developers:

```
// Integrating an LLM into your application to generate documentation
public class CodeDocumentationGenerator {
    private final LLMService llm;

    public String generateDocumentation(String code) {
        String prompt = String.format(
            "Generate JavaDoc comments for this method:\n%s",
            code
        );

        return llm.complete(prompt, new CompletionOptions()
            .setMaxTokens(200)
            .setTemperature(0.3)  // Lower = more focused output
        );
    }
}
```

Note

The *temperature* in LLM settings controls randomness. Lower values (0.0–0.5) produce more predictable output, while higher values (0.7–1.0) increase creativity. For code generation, stick to lower temperatures. Higher temperatures increase creativity but also raise the risk of hallucinations, which are confident-sounding but incorrect responses. For factual tasks, lower temperatures help maintain accuracy.

Understanding AI’s Capabilities and Limitations

Since its inception, AI has evolved significantly, but despite its current capabilities, it has important limitations you should understand. Being aware of these limitations up front will help you better recognize when and how to use AI effectively in your daily tasks.

This section explores AI’s strengths, helping you identify when to incorporate it into your workflows. Just as importantly, you’ll examine the limitations of LLMs, providing insights into areas where you might want to employ a little more caution.

What AI excels at for software developers

AI is improving efficiency and productivity across many professions, but how does it affect you as a software developer? Let’s look at examples of how AI can automate routine tasks that slow you down every day. Taking these tasks off your plate allows you to focus on more meaningful work and improve overall happiness:

Repetitive coding tasks

> Repetitive coding tasks are where AI can give you the most return on investment (ROI). AI can serve as an intelligent coding partner, helping you implement similar patterns across multiple files, create consistent APIs, and even assist with the development of entire features. This can help free up your time, allowing you to focus on the unique problems of your application rather than writing boilerplate code.

Automation scripts and “glue” tasks

> This is an area where AI can provide value in your day-to-day workflow. As a developer you often need to handle routine operational tasks like downloading files from servers, organizing directories, or automating deployment steps. AI is really good at generating shell scripts, bash files, and automating scripts for these “human glue” activities.

Code explanation

> Code explanation is one of AI’s most valuable features for developers learning a new language, framework, or getting onboarded to a new codebase. With access to a codebase, you can ask AI questions like: “What does this application do at a high level?” “What are the key entry points?”

"What are the major dependencies?" When learning something new, you can also ask AI to create a comprehensive learning plan for the topic.

Generating documentation

This is another area where AI excels. Let's face it, developers generally do not like writing documentation unless they are required to. AI can help generate code documentation at every level, from individual methods and classes to comprehensive project documentation. It can also convert documentation from one format (like Markdown) to another (like HTML).

Refactoring and optimization suggestions

Refactoring and optimization suggestions is like having a senior engineer review your code and provide feedback. It can identify bottlenecks, suggest cleaner implementations, and recommend modern language features or libraries that might improve your code.

Generating test cases

This is another area where AI can step in and allow you to focus on more creative and complex problems. Like documentation, tests are another area that is usually neglected, which can benefit from AI assistance. Even with a comprehensive test suite, you might miss tests for important edge cases. When given context about your application, AI can generate tests that align with your existing test suite's style and patterns.

Code conversion

Code conversion helps developers translate code between languages, frameworks, and data types. This works particularly well for higher-level languages like Python, JavaScript, or Java, but it has its limitations. It can be less reliable for low-level languages, hardware-specific code, or systems with complex dependencies. Always remember to test converted code thoroughly, as AI can miss subtle but critical details.

User interface design and mockup creation

User interface design and mockup creation allows AI to rapidly generate UI layouts, component structures, and even interactive prototypes based on your descriptions. AI can create HTML/CSS layouts, suggest responsive design patterns, and help you visualize different interface approaches before committing to implementation.

Understanding language and framework features

With the proper prompting, LLMs can save you a tremendous amount of time scouring documentation. Asking a chatbot to explain an API or the nuances of a library can be like having an infinitely patient and experienced pair at your fingertips.

AI's current limitations

While AI can help us be more productive in many areas, it isn't perfect. Understanding where AI falls short is important for using it effectively without compromising your code quality.

When reviewing the following limitations, it's crucial to distinguish between the model and the product. When you use a chatbot, you're interacting with a product that sits on top of and communicates with a specific model. These limitations primarily exist at the model level, though products often augment capabilities to address them. For example, if you ask ChatGPT for the current price of your favorite stock, it won't know the answer because it was trained on a dataset that goes up to only a certain date. However, ChatGPT can invoke a tool to search the web for the information it needs.

No real-time knowledge is a significant limitation of current AI systems. They cannot access up-to-date documentation, understand recent tools or frameworks, or process real-time events. These constraints limit their ability to provide the most accurate and relevant responses.

Hallucinations represent one of AI's most dangerous limitations. AI systems don't intentionally provide incorrect answers, but they can generate responses based only on their pretrained data and the context they are given. They may confidently provide inaccurate answers that sound plausible but are incorrect. This happens because AI generates responses based on patterns in training data, not by knowing facts.

Lack of contextual information about your codebase means AI cannot understand your project's architecture, existing conventions, or business rules. Without this context, it generates code in isolation, struggling to properly integrate with your existing system or adhere to your team's coding standards and preferences.

Embedded biases in training data can lead to AI generating problematic patterns or making assumptions that don't reflect diverse perspectives. AI models are trained on existing code repositories and documentation, which may contain outdated practices, cultural biases, or approaches that don't consider accessibility and inclusivity. This can result in generated code that may inadvertently exclude certain groups.

Inability to understand business requirements limits AI's decision-making capability. It cannot weigh trade-offs between performance and maintainability, understand regulatory compliance needs, or make decisions that require domain expertise about your specific industry or user base.

Inconsistent performance across different domains means AI excels with popular languages and common patterns but struggles with niche technologies, newer frameworks, or specialized domains where training data is limited.

Privacy and security risks happen when sharing code or data with AI tools. Many AI platforms store conversation history, use inputs to improve their models, or may inadvertently expose sensitive information. Sharing proprietary code, API keys, database schemas, or confidential business logic with AI tools can create security vulnerabilities and intellectual property concerns that require careful consideration.

Warning

Always treat AI-generated code with the same rigorous processes as code written by a human. Consider it a first draft that needs thorough testing, code reviews, and careful integration planning. The time saved during initial code generation should be reinvested in thorough validation.

By understanding both AI's strengths and limitations, you can create a plan for incorporating these tools into your development workflow while maintaining code quality. In the next section, you'll explore specific tools that can boost your day-to-day productivity.

AI as Your Pair Programmer

Imagine having a knowledgeable colleague sitting right next to you, ready to answer questions, suggest solutions, and help you think through problems. Available 24/7, never gets tired, and never judges you for asking “obvious” questions. That’s exactly what AI can be for you as a developer: the ultimate pair programming partner.

When it comes to coding, AI is really good at tasks ranging from inline assistance to full-scale project planning. In this section, we’ll explore categories of AI coding assistants. We’ll examine specific products in each category that are available at the time of writing this book. While individual products may come and go because of AI’s rapid advancement, these fundamental categories are likely to remain stable.

Standalone Chatbot Assistants

Standalone chatbot assistants like ChatGPT, Google Gemini, and Anthropic Claude are conversational AI systems you interact with through a text or voice interface, separate from your development environment. Think of them as your always-available coding mentor.

These tools excel at helping you understand concepts, debug complex problems, and plan architecture decisions. Use them when you need thoughtful code generation with explanations, complex problem-solving, or extended conversations about coding challenges.

The following are best practices for code-focused conversations:

- Provide context about what you’re trying to accomplish.
- Share relevant code snippets when asking about specific problems.
- Prompt with a persona as well as who you are. Ask for explanations, not just solutions.
- Use the AI assistant for learning new technologies or frameworks.

For example, if you have a list of objects that need to be sorted, a prompt such as “How do I sort this?” is unlikely to give you a useful response. It doesn’t tell the AI what kind of code you need. Instead, a prompt such as the following is likely to be more helpful: “I have a list of Customer objects and want to sort by registration date, most recent first, using Java 24.”

As you’ll continue to see in “Prompt Engineering Fundamentals”, the quality of the prompt you provide to an AI heavily influences the quality of the results you receive.

Inline IDE Assistants

Inline IDE assistants are AI-powered tools that work directly inside your IDE, offering real-time suggestions and auto-completions as you type. These tools integrate into your development environment through plug-ins or built-in features, helping speed up your coding by handling routine tasks.

Popular options include GitHub Copilot, JetBrains AI Assistant, and Amazon CodeWhisperer. Most modern code editors support these AI assistants, making them accessible to developers at any level.

You should use inline assistants when doing the following:

- Writing repetitive code like getters/setters or constructors
- Implementing standard patterns (singleton, builder, etc.)
- Generating unit tests based on existing code
- Creating code comments and documentation
- Needing quick API and syntax suggestions

If you want to get the most out of an AI assistant, you should keep these points in mind:

- Review generated code before accepting it.
- Use suggestions as starting points, not final solutions.
- Pay attention to patterns the AI learns from your codebase.
- Don't let AI suggestions override your architectural decisions.
- Start with simple, repetitive tasks and gradually use them for more complex patterns.

Let's see how an inline assistant can help with both code and documentation:

```
public class Calculator {
    // Start typing "public int add..." and AI suggests:
    public int addNumbers(int a, int b) {
        return a + b;
    }

    // Or start typing "/** Calculate..." for documentation:
    /**
     * Calculates the sum of two integers
     * @param a the first number
     * @param b the second number
     * @return the sum of a and b
     */
}
```

Tip

With any AI-generated code, you should avoid falling into the trap of accepting the code and moving on. Ask yourself, "Do I understand what this code does and why it works this way?" If you don't, do your own research to understand it more clearly. You should be able to explain every line during a code review. Remember: AI is a tool to maximize productivity, not a replacement for understanding. The goal is to learn faster, not to avoid learning altogether.

Agentic AI IDE Environments

The newest category of AI pair programming tools includes AI-powered development environments like Cursor, Junie, and Cline. These tools go beyond suggestions to actively participate in coding tasks, potentially writing entire functions or files based on your requirements.

These environments can understand your entire codebase context and make more sophisticated changes across multiple files. They're particularly powerful for refactoring, implementing features, and maintaining consistency across your project.

Here are examples of when to use agentic environments:

- Large-scale refactoring across multiple files
- Implementing features that require changes in several places
- Code migration or modernization tasks
- When you need AI to understand broader project context

Here are best practices when working with agentic environments:

- Start with clear, specific requirements.
- Define nonfunctional requirements up front (code style, security standards, testing approach).
- Review all changes carefully before committing.
- Use version control to track AI-generated changes.
- Break large tasks into smaller, manageable pieces.

Consider a practical example. Imagine you have a dashboard application with five existing widgets: WeatherWidget, StockPriceWidget, NewsWidget, CalendarWidget, and TaskListWidget. You ask an agentic AI to create a new StatusWidget that shows system health metrics.

Here's how an AI agent approaches this task:

Analysis phase

> The AI first examines your existing widgets to understand patterns:
>
> - Reviews the base Widget class structure
> - Identifies common methods like `render()`, `updateData()`, and `getConfig()`
> - Notices the consistent use of a data service pattern
> - Observes styling conventions and component organization

Planning phase

> The AI creates a plan:
>
> - "I need to create StatusWidget extending the base Widget class"
> - "I'll need a StatusService to fetch system metrics"
> - "The widget should follow the same refresh pattern as others"

- "I'll need to add the new widget to the dashboard registry"

Implementation

Only after this analysis does the AI begin writing code, creating the following:

- *StatusWidget.java* following established patterns
- *StatusService.java* matching the existing service architecture
- Updated dashboard configuration to include the new widget
- Consistent styling that matches the other widgets

This demonstrates the power of agentic AI: it doesn't just generate code; it understands your project's context and maintains consistency across your entire codebase.

Tip

Remember: you are the pilot, not the passenger. AI tools are powerful assistants, but you remain responsible for understanding the code, making architectural decisions, and ensuring quality. Use AI to amplify your capabilities, not replace your thinking. Ultimately, you bear responsibility for all code in your project, regardless of whether you or AI generated it.

The key to success with any AI coding tool is understanding its strengths and limitations, then choosing the right tool for each situation. As you grow as a developer, these AI assistants will evolve alongside you, becoming more sophisticated partners in your coding journey.

Prompt Engineering Fundamentals

When it comes to becoming a software developer who leverages AI, prompt engineering is one of the most valuable skills that you will need to understand. Whether you're a complete beginner or experienced user, mastering how to craft clear, effective prompts will dramatically improve your results when working with AI assistants, code generation tools, and agentic IDEs.

The reality is that many developers struggle with AI tools not because the technology is lacking, but because they haven't learned how to communicate their needs effectively. This skill often presents itself in the real world when writing clear requirements for a teammate or documenting your code. When working with AI, you need to treat it like a conversation you would have with a coworker and use clear and thoughtful communication.

What Is Prompt Engineering?

As you learned earlier in this chapter, there are some terms that make working with AI a little bit intimidating, and *prompt engineering* is another example of that. Prompt engineering is just learning how to effectively communicate with AI. Think of it like learning how to effectively use a search engine. It is a skill everyone needs to develop, not something that is reserved for specialists.

Imagine you're at the office and turn to a junior developer in the next cubicle, asking them to "fix the login bug." Without context, this simple task can become a complex problem. They might waste time searching through the ticket system for open issues. Since there are multiple login forms, which one needs fixing? They could spend hours testing the login system, finding nothing wrong. But what if instead you said, “The login form on the user dashboard isn’t validating email formats correctly, allowing users to submit invalid email addresses that are causing issues downstream.” You would get much better results from this interaction.

The same principle applies to AI. The quality of your prompt directly correlates with the quality of the AI’s response.

Essential Prompt Engineering Techniques

Getting the most out of AI isn’t just about asking questions; it’s about asking the right questions in the right way. In this section, you’ll discover techniques that will help you transform the way you interact with AI tools. Instead of getting generic responses that require extensive iterations, you’ll learn to write prompts that consistently produce the results you’re looking for.

Clear communication is key

If you want to get good results from AI, you need to be specific. This is a technique you will learn over time by iterating on your prompts when you don’t get your desired output. For example, consider the following prompts.

Bad prompt:

```
Write some Java code for sorting
```

This prompt is too vague. It doesn’t specify what to sort, how to sort it, or what data structure to use. The AI has to guess whether you want to sort numbers, strings, objects, or something else entirely.

Good prompt:

```
Write a Java method that sorts an ArrayList of Employee objects by their salary
in descending order. Include error handling for null inputs.
```

This prompt gives the AI everything it needs: the specific data structure (`ArrayList`), what you’re sorting (`Employee` objects), the sorting criteria (salary), the order (descending), and even an edge case to consider (null inputs).

Structure determines success

A well-structured prompt can help AI understand not just what you want, but how you want the output delivered. For example, compare the following two prompts.

Bad prompt:

```
Explain database indexing
```

This prompt gives no guidance about depth, perspective, or format. The AI might respond with anything from a single sentence to a graduate-level computer science explanation.

Good prompt:

```
As an experienced Java developer, explain database indexing in 3 parts:
```

```
1. What indexes are and why they matter for performance
2. When to use clustered vs non-clustered indexes
3. One practical example of creating an index in PostgreSQL

Keep each section to 2-3 sentences and include one code example.
```

This prompt provides clear structure (three numbered parts), specifies the audience perspective (Java developer), sets length expectations (two to three sentences each), and requests a concrete example.

Think of it as teaching, not commanding

If you are a good mentor for a junior developer, you wouldn't walk over to them and demand a result on a specific task. Instead, you would give them as much context as possible and teach them through examples that might already exist in your codebase. Instead of commanding results, guide the AI through your thought process. This approach, called *chain-of-thought prompting*, leads to more accurate responses. For example, compare the following two prompts.

Bad prompt:

```
Optimize this code.

[paste your code here]
```

This prompt provides no context about what "optimize" means to you. The AI doesn't know if you want faster execution, less memory usage, better readability, or something else entirely.

Good prompt:

```
I need to optimize this Java method that searches through a large dataset.
First, help me identify the current time complexity. Then suggest specific
improvements. Finally, show me the refactored code with comments explaining
the performance gains.

[paste your code here]
```

This prompt breaks the task into logical steps, specifies the optimization goal (performance for large datasets), and requests explanations alongside the code changes.

Practical tips for immediate improvement

The following are practical prompt engineering techniques you can use immediately to improve your AI-generated results:

Be specific about context
: Always provide relevant background information. Is this a greenfield project or an existing legacy one that isn't able to take advantage of modern best practices? Is this a personal project or a mission-critical application at work?

Use examples
: Show the AI what good output looks like. If you want code formatted a certain way, provide an example.

Give context about your environment
: Mention your Java version, frameworks you're using, or constraints you're working within.

Specify a role

Give the AI a specific persona or expertise to embody. For example, "Act as a senior Java architect reviewing this code" or "As a performance optimization expert, analyze this query." This helps frame the response with the appropriate level of detail and perspective.

Iterate and refine

Don't expect perfection on the first try. Use the AI's response to refine your prompt and get closer to your desired output.

Use AI to improve your prompts

When you're not getting the results you want, ask the AI to help you craft a better prompt. Try something like, "I'm trying to get you to help me debug this performance issue, but your response wasn't quite what I needed. Can you suggest how I should rephrase my request to get more specific debugging steps?"

Save good prompts

When you craft a prompt that works well, save it. You'll likely need similar requests in the future. For more detailed tips and tricks on this topic, see "Personal Knowledge Management".

Advanced Prompt Engineering Techniques

Once you've mastered the basics, advanced techniques can help you tackle more complex development tasks. We'll explore two main categories of advanced techniques: structuring techniques that help you format and frame your requests, and organizational techniques that help you manage complex, multipart prompts.

Structuring techniques

Let's take a look at some common prompting techniques and how to use them.

Zero-shot prompting is asking the AI to perform a task without providing any examples. There is nothing wrong with this approach, and at times, this will give you exactly what you are looking for. For example:

```
Create a Java class that implements the Observer pattern for a
stock price monitoring system.
```

One-shot prompting provides a single example to establish a pattern you would like the AI to follow. For example, let's say you provide AI the following:

```
// Here's an example of the coding style I prefer:

public Optional<User> findUserByEmail(String email) {
    if (email == null || email.trim().isEmpty()) {
        return Optional.empty();
    }
    return userRepository.findByEmail(email.toLowerCase());
}

// Now create a similar method called findUserById that takes a Long id parameter.
```

In the example, you taught the AI about your preferences when it comes to writing a particular type of method. Based on that, you should be able to get similar methods generated for you that follow that style.

Few-shot prompting teaches AI models through examples rather than just instructions. Instead of explaining what you want, you show the AI several examples of the desired input-output pattern. For example, say you want the AI to classify customer review sentiment:

```
Classify the sentiment of these customer reviews:

Example 1: "The software is intuitive and saves me hours of work" → Positive
Example 2: "Great documentation and excellent support team" → Positive
Example 3: "Buggy interface and crashes frequently" → Negative

Now classify: "The new features are helpful but the UI is confusing"
```

If you remember back to our AI terminology, this comes back to *classification* and giving the model enough examples so that it can correctly classify future examples.

Organizational techniques

Organizational techniques help you structure requests to get better, more accurate responses from AI systems. By clearly organizing your prompts, you reduce uncertainty and help the AI focus on what matters most.

XML tags (and other structuring formats like Markdown or JSON) help structure complex requests by clearly separating different types of information. This makes it easier for the AI to understand what each piece of information is for and how to prioritize it. For example:

```
<task>
Create a RESTful API endpoint for user management
</task>

<requirements>
- POST /users for creating users
- Include validation for email and password
- Return appropriate HTTP status codes
- Use Spring Boot annotations
</requirements>

<constraints>
- Java 17
- Spring Boot 3.0
- No external dependencies beyond Spring starter
</constraints>
```

The XML structure helps the AI distinguish between what you want built (task), what it must include (requirements), and what limitations it must work within (constraints). Without this structure, all the information gets mixed together, and the AI might miss important details or prioritize them incorrectly.

Task decomposition breaks complex problems into manageable pieces. This is something we as software engineers do all the time. Just as you would break this into smaller tasks for yourself or for a junior developer, you should do the same for the AI to improve your results. This is important because AI models can become overwhelmed by large requests and produce incorrect or incomplete responses. Breaking your request into phases lets you get a more thorough analysis of each component and adjust along the way. For example:

```
I'm building a file processing system in Java. Help me break this down:

Phase 1: Design the file reading strategy (stream vs batch)
Phase 2: Create the data validation logic
Phase 3: Implement error handling and logging
Phase 4: Add unit tests

Start with Phase 1 - analyze the pros and cons of each approach
for processing 1GB+ files.
```

The key to becoming productive with AI tools isn't about memorizing frameworks or techniques; it's about developing clear communication habits and understanding what approaches work best. Start by writing specific prompts that teach the system your desired output. Give context about yourself and your project and be willing to refine your approach through iteration.

Now that you understand how to get the most out of AI, let's explore what the future of software development might look like.

How AI Might Shape Software Engineering

What does the future of work look like for software engineers? While no one can predict with certainty what lies ahead, AI will undoubtedly play a significant role in software development. As AI continues to grow and change over time, it is important to consider what makes you stand out from other engineers. As AI becomes increasingly proficient at writing code, your unique qualities and valuable soft skills will become more important than ever, whether that is your collaboration abilities, empathy, mentoring capabilities, or other distinguishing traits.

This section explores how AI is reshaping the developer's role. Rather than replacing developers, AI serves as a powerful tool that enhances productivity and transforms how we write and review code. Learn how to leverage AI to become a more effective problem solver while highlighting your existing skills.

Will AI Take My Job?

In this chapter, you have seen how AI can automate tasks, generate code, write tests, and create documentation. As these capabilities improve, software developers at all levels are asking, "Will AI take my job?" To answer this question, we can draw some reasonable conclusions from historical patterns and current trends.

As you have learned throughout this book, being a software engineer involves much more than writing code. In Chapter 2, you learned that you will spend far more time reading code than writing it. This fundamental skill, the ability to understand and comprehend code, remains essential, whether that code comes from another developer or AI.

This is just one example of the many things you do as a software developer outside of writing code. Let's take this opportunity to review some of the tasks that make up your role. Before writing any code, you have the responsibility of turning incomplete and ambiguous requirements into working systems. As you learned in Chapter 9, you make judgment calls ("it depends") about trade-offs between features like performance and maintainability. You navigate office politics, mentor junior developers, and explain technical concepts to nontechnical stakeholders. You understand not just *how* to build something, but *why* it should be built and *who* it serves.

AI has been trained on millions of examples, both good and bad, and can generate new code based on this pretrained data. But it doesn't understand your company's specific business model, your users' unique needs, or why the CEO's "simple request" is actually a complete architectural nightmare. It can't be present in a meeting and figure out that the product manager's requirements don't match what the sales team promised. These human elements of software development aren't edge cases; they're the job.

> AI won't replace developers, but developers who use AI will replace those who don't.
>
> Jeff Atwood (attributed), software developer, author, blogger and entrepreneur

Remember the Jacquard loom from this chapter's introduction? The weavers who thrived were not the ones who fought the technology, but the ones who learned to operate it, designed patterns for it, and used it to create things that were impossible before.

There is a similar pattern playing out today for software developers. AI is eliminating the mundane tasks that programmers often find tedious. This is everything from writing boilerplate code to creating repetitive tests, creating documentation, and more. You could look at this as a threat, or you could see it for what it is: liberation. When you spend less time on unchallenging tasks, you have more time for the creative and strategic work that makes software development exciting.

If you step back and look at the evolution of programming, the industry has gone from assembly languages to high-level languages, from simple text editors to powerful development tools, sophisticated IDEs, and automated CI/CD pipelines. Through all of the technology advancements that might have made programmers obsolete, the demand for developers has grown year over year instead.

The developers who are going to succeed in the AI future are those who see AI as their coding partner—a partner who has superpowers and never gets tired, has the historical knowledge of the internet, and can do basic tasks so you can work on bigger and more complex problems. What makes you valuable is not your ability to write a for loop when called upon or how to implement a bubble sorting algorithm, but in knowing *when* to use them, *why* they matter, and *how* they fit into the larger application that you're building.

So will AI take some jobs? Possibly. But will it take the majority of jobs? Not anytime soon. Some roles will be reimagined, and that's OK. If you are able to embrace that change, you will find yourself more productive and capable of building things that wouldn't have been possible just a few years ago. The question isn't whether AI will replace you; it's whether you'll be one of the developers harnessing this powerful new tool or one of the few still hanging on to your ability to simply write code like it's 2010.

So if AI isn't replacing us, how exactly will it change our day-to-day work? Let's explore how the developer role is evolving and what new skills will set you apart.

Vibe Code Reviews

On February 5, 2025, Andrej Karpathy, one of the most influential figures in AI and deep learning, made a post on X that encapsulated his current experiences in software development and coined the term *vibe coding*. The term itself is deliberately tongue-in-cheek. Karpathy coined it to highlight problematic development practices, not to endorse them.

What is vibe coding?

Vibe coding refers to the practice of using agentic IDEs to generate entire applications through well-crafted prompts. Instead of going through the mundane task of typing out every line of code manually, developers can describe what they want to build, and AI tools generate a significant amount of the codebase autonomously.

This approach has made software development accessible to a whole new group of creators. Someone with a great idea no longer needs to hire a team of developers or spend years learning code just to build out a basic prototype. They can describe their vision to an AI assistant and watch their concept come to life right before their eyes.

Think about the implications for a moment. A restaurant owner who wants to create a simple ordering system, a teacher building an interactive quiz for their students, or an artist developing a portfolio website can now bring their ideas to reality without traditional coding experience.

The benefits and dangers

As Karpathy noted in his original tweet, vibe coding works well for "throwaway weekend projects," but the key word here is "throwaway." There is an enormous difference between building a personal project that only you will use and developing software that powers the backbone of Fortune 500 companies.

Consider these two scenarios:

Scenario 1
: You want to build a personal expense tracker to categorize your monthly spending. You use vibe coding to generate the application over a weekend. If a bug miscalculates your coffee expenses, the worst outcome is a slightly inaccurate budget.

Scenario 2
: Your company needs a payroll system that handles thousands of employees across multiple states with different tax requirements. A bug here could mean employees don't get paid correctly, tax obligations aren't met, and the company faces legal consequences.

The stakes couldn't be more different.

The enterprise reality check

When you read headlines claiming that "25% of code at major companies is now written by AI," it's important to understand what this actually looks like in practice. This AI-assisted code isn't the result of casual coding sessions where entire applications are magically generated from simple prompts.

A realistic example is the workflow that companies like Microsoft have demonstrated. A developer creates a detailed issue describing the problem statement. An AI coding agent reads that issue, generates the actual code changes, and then creates a PR with those changes. A human developer (or team of developers) reviews the code and decides whether to merge this change into the codebase. AI is handling the routine implementation work, but humans remain in control of quality assurance and decision making.

The enterprise environments that you work in have something these weekend warriors are lacking: rigorous processes. It's not to say that some level of vibe coding isn't happening within the enterprise, but here they have checks and balances. They have code reviews, testing frameworks, deployment pipelines, and quality assurance procedures. This new AI-generated code still needs to go through the same scrutiny as human-written code. In fact, these processes are more critical than ever.

Why code reviews are more important than ever

While AI tools can generate code faster than ever before, it doesn't mean that your timelines shrink exponentially or that they don't come without hidden costs. The generated code (like human-written code) can contain subtle bugs, security vulnerabilities, or architecture decisions that might not be right for your application. Expert developers like Neal Ford warn of a potential "tsunami of bad code" as teams rush to implement AI-generated solutions without proper review processes.

This is why code reviews have become more critical than ever, not less important. As the developer of this feature, you need to understand what every line of generated code does and why certain decisions were made. As a reviewer, you need to understand that regardless of who wrote it (human or machine), it needs your attention to every detail. Do you want to discover the problem on pager duty one weekend or spend the time during a code review to catch these issues?

Because code reviews are going to become a more important part of the SDLC, there is an important takeaway. If you are going to use AI to generate a method, class, or entire functionality, you need to understand and explain "your" code. Before submitting a PR for code review, you need to vigorously go through your code and be able to explain why that code exists and what it does. You can't use the excuse that AI generated that code and that it wasn't your fault. Whether you write it or AI does, you are responsible for it and accountable for the results.

The new developer mindset

As a developer in this AI-assisted era, the key is avoiding the vibe coding trap. While AI tools can accelerate development, the casual, prompt-and-pray approach that Karpathy satirized represents everything you should avoid. Instead, you need to think of yourself as both a creator and a curator of code. You should use AI thoughtfully while maintaining rigorous standards.

This means developing the following skills:

- Writing effective prompts that generate better AI code
- Quickly reviewing and understanding generated code
- Identifying potential issues in both human and AI-written code
- Identifying any potential biases or exclusivities
- Communicating effectively during code reviews

Vibe coding represents an exciting change in software development, but it doesn't eliminate the need for careful, thoughtful development practices. If anything, these processes become more important than ever. As you grow in your career, embrace the tools that make you more efficient and productive, but don't lose sight of the fundamental responsibility to ship reliable, maintainable software.

AI as Your Force Multiplier: From Writing Code to Problem-Solving

A *force multiplier* is both a military and business concept: it's a tool or capability that amplifies your existing skills. For example, most people use their smartphones for taking photos. Today's smartphone cameras are excellent and come with features to help you take high-quality photos. Yet there's a whole range of upgrades available in the form of DSLR cameras. These professional cameras can cost thousands of dollars and produce stunning photos if you know how to use them.

If you were given a $5,000 camera and handed a professional photographer with 20 years of experience just a smartphone, who would take better pictures? The professional photographer would likely win because they understand concepts like aperture, exposure, lighting, and composition, plus they have the artistic vision to capture exactly the image they want. The expensive camera is a force multiplier. It amplifies the photographer's existing skills, but it can't create skills that aren't already there.

This is similar to the current state of AI in software development. The tool doesn't make you a better developer any more than an expensive camera makes you a photographer. Your knowledge, experience, and problem-solving abilities are what set you apart. AI is your smartphone, a powerful tool that amplifies what you already know.

At the end of the day, no matter what technology you use, you are still building software. The fundamental shift that developers need to consider is not what you build, but how you spend your time building it. Traditional software development follows patterns. You might spend 70% of your time writing boilerplate code, debugging syntax errors, and implementing repetitive patterns. This leaves 30% of your time for creative problem-solving and all of the other administrative tasks that go into your job.

AI allows us to flip this equation. It can now handle the routine code generation and debugging of issues. This allows you to dedicate 70% of your energy to requirement analysis, architectural decisions, and solving complex business problems. In theory, we are shipping products at the same rate, but now they are going to production well thought out and with fewer bugs and edge cases to deal with.

Let's take a look at a problem you might be familiar with. Consider building a REST API for whatever domain you are working in. Prior to AI, you might have to spend hours creating domain objects, validation logic, controllers, repositories, and more. With AI, good prompting techniques, and a template of what REST APIs look like in your organization, you can generate these components in minutes.

With time saved on writing boilerplate code, you can now focus on some of the strategic decisions that matter. Instead of just shipping that API, you can spend some more time thinking about the design of the API and whether it's intuitive. Have you considered how this API is going to handle high throughput?. Are you taking advantage of concurrency features in the language or framework you are working with? Will this API both scale and fail gracefully under load? Yes, your application has to work first, but these are the types of complex problems that you should be spending your time on.

Note

Dan here. If you’re worried about AI reducing the lines of code you write, don’t be. Lines of code isn’t actually a meaningful measure of productivity or value. I’ve reviewed countless resumes over the years, and I’ve never seen anyone list “wrote 10,000 lines of code” as an accomplishment, nor would I be impressed If I did. Real accomplishments focus on impact: “Saved the company 20% on our cloud bill by optimizing our application architecture” tells a much more compelling story than any line count ever could.

The shift from writing code to problem-solving doesn’t diminish your role one bit; it elevates it. You’re solving bigger problems and making decisions that have a greater impact on the team and applications you work on.

Wrapping Up

AI isn’t going to replace software developers, but developers who effectively leverage AI will have a significant advantage over those who don’t. Just as the Jacquard loom transformed weavers rather than eliminating them, AI is transforming how we approach software development.

AI works best as your pair programming partner, not your replacement. Whether you’re using standalone chatbots for architectural discussions, inline IDE assistants for boilerplate code, or agentic environments for complex refactoring, the goal is the same: amplify your existing skills and free up mental energy for higher-level problem-solving.

Understanding AI’s capabilities and limitations is an important component for using it effectively. AI excels at repetitive tasks, code explanation, documentation generation, and pattern recognition. However, it struggles with real-time knowledge, can hallucinate incorrect information, lacks business context, and may introduce security risks. Treating AI-generated code with the same rigor as human-written code through thorough testing and code reviews isn’t just good practice—it’s essential.

The future of software development shifts your focus from writing code to solving problems. Instead of spending 70% of your time on boilerplate implementation, you can dedicate that energy to architectural decisions, requirement analysis, and complex business logic. This doesn’t diminish your role as a developer; it elevates it to work on more meaningful and impactful challenges.

As you begin incorporating AI into your development workflow, remember that prompt engineering is a learnable skill that dramatically improves your results. Be specific about context, provide examples, structure your requests clearly, and don’t hesitate to iterate on your prompts. Most importantly, maintain ownership of your code regardless of who or what generated it.

The developers who will thrive in this AI-enhanced future are those who embrace these tools while maintaining their commitment to code quality, continuous learning, and the fundamentals of software engineering. AI gives you superpowers, but you are responsible for wielding them correctly.

Throughout this book, we have shared the hard-earned lessons that we’ve learned throughout our careers. We feel humbled to share our experiences to help you, the next generation of software engineers, achieve your goals. We hope that you have enjoyed the book as much as we enjoyed writing it.

We can often be found at meetups and conferences, and we hope you’ll take a moment to introduce yourself if our paths cross. Thank *you* for taking the time to read this book. We appreciate it more than you will ever know!

Putting It into Practice

Implementing what you've learned requires action, not just knowledge. The following practical steps will help you integrate AI tools into your development workflow while maintaining code quality and building essential prompt engineering skills:

1. Take an existing method in your codebase and ask AI to explain what it does, identify potential improvements, and suggest test cases. Compare its analysis with your own understanding.
2. Practice prompt engineering by crafting three well-structured, detailed, contextually specific prompts using the techniques from this chapter for common coding tasks like debugging, code review, or documentation.
3. Create a personal "AI prompt library" by saving five to ten effective prompts you've crafted for common development tasks like code reviews, documentation generation, and debugging.
4. Choose one AI coding assistant and use it for one week on a personal project. Document what tasks it excels at and where it struggles.
5. Practice "vibe code reviews" by having AI generate a simple class or method, then conduct a thorough code review as if a junior developer had written it. Document what issues you find.
6. Use AI to help convert a small piece of code from one programming language to another (like Python to Java), then manually verify that the conversion is correct and follow best practices.
7. Establish your AI usage guidelines by defining when you will and won't use AI assistance, what types of code require extra scrutiny, and how you'll handle sensitive or proprietary information.

Additional Resources

- *AI Engineering* by Chip Huyen (O'Reilly, 2024)
- *Beyond Vibe Coding* by Addy Osmani (O'Reilly, 2025)
- *Prompt Engineering for LLMs* by John Berryman and Albert Ziegler (O'Reilly, 2024)
- "No Silver Bullet" by Fredrick P. Brooks Jr. (University of North Carolina at Chapel Hill)
- *Rebooting AI: Building Artificial Intelligence We Can Trust* by Gary Marcus (Vintage, 2020)
- *Co-Intelligence: Living and Working with AI* by Ethan Mollick (Portfolio, 2024)
- *Human Compatible: Artificial Intelligence and the Problem of Control* by Stuart Russell (Penguin Books, 2020)

- *Artificial Intelligence: A Modern Approach* by Stuart Russell and Peter Norvig (Pearson, 2021)
- *Deep Learning with Python* by Francois Chollet (O'Reilly, 2021)
- *Taming Silicon Valley* by Gary Marcus (Tantor Media, Inc., 2025)
- Simon Willison's Weblog
- Marcus on AI
- Practical AI podcast