1 Introducing generative AI

This chapter covers

- The ways generative AI transforms coding with context-aware help
- The evolution of AI developer tools from IDE integration to standalone assistance
- LLM fundamentals and code-generation capabilities
- AI-enhanced workflows
- Success factors for integrating AI into your development process

Robots are not going to replace humans, they are going to make their jobs much more humane. Difficult, demeaning, demanding, dangerous, dull— these are the jobs robots will be taking.

—Sabine Hauert, Co-founder of Robohub.org

What if you could use your existing Python expertise alongside AI that understands your code context, anticipates patterns, and generates implementation details while you focus on architecture and design? That's the power of generative AI tools for experienced developers. When I first encountered these tools, I approached them with healthy skepticism. But after integrating them into real production projects over the past year, I've reduced implementation time by approximately 30%, while improving code quality and test coverage.

It's likely you've already used ChatGPT or Claude for coding. You've probably seen GitHub Copilot suggestions pop up in your editor. Or maybe you're just curious about all the AI buzz. If you're interested in learning how to use these tools to make yourself super productive, you're in the right place. This book is your practical guide to using AI tools to supercharge your coding, and no AI expertise is required.

This book will show you exactly how to use these tools to write code faster, catch bugs earlier, create better documentation, create design diagrams (UML, flowcharts, etc.), and test your code more thoroughly. The best part? You don't need a PhD in math, data science, or a background in AI to benefit. I've traveled this road extensively and discovered valuable tricks along the way. Consider this your field guide to generative AI coding tools. I'll help you navigate the potholes I've encountered.

This book approaches generative AI from a developer's perspective, examining both programming-specific tools and general text generators that belong in your toolkit. We'll look at how these revolutionary tools work and how to use them efficiently. Coding will never be the same from now on.

By sharing my insights and experiences, I aim to cut through the hype and sales pitches to focus on what matters—making you a more productive Python developer. These techniques extend to many languages, empowering you to use AI as a tool for innovation and growth, and adding both fun and productivity to your daily work.

This book provides an overview of several popular tools. It includes step-by-step instructions on installing and using these tools to your advantage. You'll also learn techniques for crafting effective prompts to get the best results.

1.1 Generative AI for coders

Generative AI can benefit you, the coder, in various ways, from code generation and bug detection to documentation and testing. Let's take a look at the ways generative AI can assist you in your everyday development work.

1.1.1 Code generation and autocompletion

Autocompletion of code by software is nothing new. We've been using that for years. Smart autocompletion and code generation, however, are much newer concepts. Large language models (LLMs) can be trained to understand programming languages in depth and generate code snippets in a smart way. What do I mean by smart? They can utilize context and evaluate the code around it. They can generate code based on user inputs or requirements. By employing AI tools, developers can quickly prototype

ideas or even generate entire applications. Many AI-powered tools predict and suggest the next lines of code, as you type. This makes the development process much faster. Let's take a look at a simple example comparing traditional Python development with an AI-assisted approach.

You need to parse a CSV file, filter rows based on certain criteria, perform calculations, and output the results. In a traditional approach, you might

- Search for the Python CSV module documentation
- Write boilerplate for file opening and error handling
- Implement the parsing logic line by line
- Debug edge cases manually

In an AI-assisted approach,

- You comment, "# Parse the CSV file at `'data.csv'`, filter rows where the `'status'` column equals `'active'`, calculate the average of the `'value'` column, and write results to `'output.csv'`."
- The AI generates a complete implementation, including error handling.
- You review the response, adjust for specific requirements, and test it.

1.1.2 Bug detection and automated fixes

Generative AI analyzes existing code to identify potential bugs, security vulnerabilities, and performance problems. AI tools can evaluate context while generating suggestions for the code you're working on. Many of these tools learn as they go. Since they're based on trained models, they are refined over time to get even better. They detect problems and suggest appropriate fixes, saving you heaps of time.

1.1.3 Documentation generation

Writing clear, concise, and accurate documentation is crucial for a successful software project to thrive. Without good documentation, your users or other developers will suffer. The greatest software written can be useless without good documentation. But documentation can be boring to write. Generative AI helps with this by automatically generating human-like documentation for your software. It can provide well-structured and contextually relevant explanations for your code. Not only does it generate documentation for you, but it can help you understand your own code better as well.

1.1.4 Code refactoring and optimization

It's always good to take a second or third look through your code to ensure there aren't any errors and that it is optimized. AI tools make this process much easier as they can analyze your code and make suggestions. They can identify redundant code, inefficient algorithms, and more. By suggesting improvements, they make refactoring easier and more effective.

1.1.5 Test case generation and mock data creation

I'm one of those strange developers who loves testing and building mocking tools. Creating good tests is imperative, and I've found that many generative AI tools produce great tests and uncover things I haven't thought of. They can be used to generate test cases and create mock data for your application that meets your needs. This improves your testing systems significantly.

What generative AI tools am I talking about? Let's take a look.

1.2 Developer tools landscape

Generative AI is still new, yet AI developer tools are already making their mark on the industry. These tools utilize LLMs to generate code, provide suggestions, and automate tasks. We're going to take a look at two types of tools:

- *Integrated tools*—Tools that work within Visual Studio Code or other IDEs and function within them
- *Standalone tools*—Tools with their own interface, usually a website, that don't interact with an integrated development environment (IDE)

1.2.1 Integrated developer tools

Generally, standalone tools are meant for many types of general text generation and chat. Think of ChatGPT or Gemini, which have a web interface and are meant for general help. Integrated tools are designed for software development. They can often generate code specific to your problem within your code, using your code as context, which we'll examine in this book. Standalone tools such as ChatGPT are better for abstraction and design.

The integrated tools we'll be using are all powered by generative AI to assist you as you're writing code, boosting your productivity and revealing easier, smarter coding techniques. Your software can become more efficient, accurate, and performant with the use of AI tools. Although each tool is different, many of them operate similarly. In this book, we are going to examine their differences. And using the tools properly will make you a better developer.

GitHub Copilot

GitHub is a well-known name in the developer ecosphere. Most developers today have at least some of their code on GitHub. Microsoft released GitHub Copilot in October 2021. It's an AI-powered code completion tool developed by Microsoft and OpenAI. It uses the OpenAI Codex model. Copilot integrates with popular editors such as Visual Studio Code. It suggests code improvements, completions, comments, and even functions as you type. Copilot is context aware and provides relevant suggestions in a variety of programming languages and frameworks.

Tabnine

Tabnine is another popular AI-powered code assistant. Tabnine utilizes many popular LLM models to provide context-aware code suggestions. It integrates with popular

code editors such as Visual Studio Code and IntelliJ. It has a local version for offline use, as well as a cloud-based version for faster, more accurate suggestions. Its feature, called "Deep Completion," uses deep learning to provide more accurate suggestions. Tabnine is also contextually aware of your code as you write and will attempt to auto-suggest code in the style you use.

BLACKBOX AI

Blackbox AI is an AI-powered code assistant that works within Visual Studio Code and Jupyter Notebook. It is available for over 20 programming languages, including Python, JavaScript, TypeScript, Go, and Ruby. Blackbox AI is an integrated tool, but it also has a web interface and the ability to ask questions and interact with the backend model from your IDE.

1.2.2 Standalone tools

In addition to integrated developer tools, there are several standalone tools and platforms that employ generative AI for code generation and assistance. These tools operate outside of traditional integrated development environments (IDEs) and usually have a web interface.

CHATGPT

Surely, you've heard of ChatGPT by now. It's an awesome tool that can help software developers with developing software outlines, code generation, testing, documentation, and more. Its ability to understand and generate text specific to programming languages and frameworks is striking. The GP4x models used by ChatGPT are impressive.

ChatGPT uses a web interface for communication—you can enter a question and get an answer. Most importantly, you can have a full discussion with ChatGPT, and it keeps context in the threads. There is also a CLI (command line interface) and a full API for ChatGPT, which gives you many options for interacting with it, including the ability to build plugins.

GOOGLE GEMINI

Google Gemini is similar to ChatGPT from an interface standpoint. You can ask questions and receive answers. It generates software outlines, code, and so on, just like ChatGPT. Functionally, they are very similar; however, in my experience, Gemini isn't quite as sophisticated as ChatGPT yet. It will eventually get better and become a great contender.

One advantage to Google Gemini is the potential to integrate with other Google Services in their ecosystem, which I consider a great future advantage. Also, Google as a company has access to a lot of source code for training. This could help the model improve over time.

COPILOT CHAT

Copilot Chat has a similar interface to ChatGPT and Gemini. It uses several different models from OpenAI and Anthropic on the backend. However, there are some

differences. Although Copilot has a familiar web-based question-and-answer format, it's also integrated into Microsoft software. Both desktop and mobile versions are available.

Another difference is that the results focus more on simple requests than complex conversations. When you put in requests, it does a "search," which may perform differently than other text prediction functionality. It also has an "agent" mode so you can give it a list of tasks, and it will go through the list and attempt to perform them step by step.

1.3 How does generative AI work?

Generative AI is a kind of statistical mimicry of the real world, where algorithms learn patterns and try to create things. If we replace the child with a generative AI model, we must "train" it to create a dog. We need to show it thousands of photos of dogs as examples. The patterns gathered from these pictures help the model learn more about dogs. What shape is a dog? How many legs does it have? What are the odds it has a tail? These are all possibilities with a probability attached to them. These parameters and many more would be used for the model. The tool can use this model to assemble what it thinks a dog will most likely look like.

Similarly, when musicians learn to play an instrument, they aren't just memorizing notes. They learn the patterns, rhythms, and structures of songs. They'll listen to a particular song enough to train themselves on what it's supposed to sound like. They'll listen to so many songs that they grasp exactly what a song in general should sound like. Eventually, they can play the songs others have written until they sound like the original. Then, of course, they move on to improvisation and creating songs of their own based on this training.

Let's add another layer to this analogy: feedback. Musicians seldom work in a vacuum. How do they know they're playing the song correctly? By sharing it with others and looking for feedback. If the music teacher nods, they're doing it right. If the audience applauds, the musician knows the song is successful. If the audience throw tomatoes, they know something has gone wrong and it needs to be fixed. This is the evaluation and feedback process that contributes to their ongoing training.

Generative AI used for coding is very similar. The model evaluates hundreds of thousands of lines of code or more. It parses the code and looks for patterns used to create working software. With enough training, it develops an idea (this is what software looks like) of what new, original code should be.

First, training data is created by taking existing source code in many languages and feeding it into a model. This model is evaluated and has layers that look for specific things. One layer checks the type of syntax. Another checks for keywords and how they're used. The final layer determines whether "this is most likely to be correct and functional source code."

There is a vast array of machine learning algorithms that use the model to run through these layers and draw conclusions. Then, the AI produces output that is a

prediction of what the new software should look like. The tool says, "based on what I know, this is the most statistically likely code you're looking for." Then you, the programmer, reach the evaluation point. If you give it a thumbs up, the feedback returns to the model (in many cases, not always) as a correct prediction. If you give it a thumbs down and reject it, that is also tracked. With this continuous feedback, the tool learns what good code should look like.

Figure 1.1 illustrates the feedback loop between developer and AI that powers generative coding tools. Unlike traditional code completion, which operates on predefined rules, generative AI creates a continuous improvement cycle, which includes the following five basic steps:

1. *Developer input*—You provide source code, comments, or natural language requirements.
2. *Context analysis*—The model analyzes patterns in your existing code and requirements.
3. *Prediction*—Based on training data and your specific context, the model generates probable code.
4. *Developer feedback*—You accept, modify, or reject suggestions.
5. *Model adaptation*—The system incorporates your feedback to improve future suggestions.

This cycle creates a powerful symbiotic relationship—the AI learns your coding patterns and preferences, while you gain implementation speed and exposure to new patterns and techniques that might not have been in your toolkit.

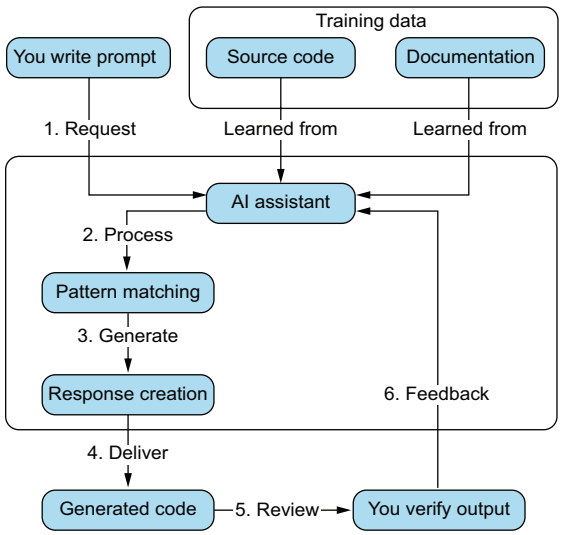

Figure 1.1 Integrated tools use a sophisticated system to generate code. The process starts with your prompt, and the assistant gathers up documentation and source code to see whether your answer can come from these sources. It makes a best guess at what you're looking for and generates a response. Your acceptance of these responses helps train the assistant in the future (unless you've blocked feedback).

This is a very high-level explanation of generative AI. It's the science of predicting what is most likely to be a correct example of something new, based on the data it was trained on. There are features in the algorithms that make things probabilistic instead of deterministic. A deterministic system will always produce the same output if given the same input—it follows fixed rules with no randomness or variation. For example, a traditional calculator always gives exactly 4 when you input 2 + 2. Generative AI models are not deterministic by design. You rarely get the same answer twice. This is intentionally done to create originality in the output. In other words, AI models strive to generate something *new* rather than regurgitate a copy of something already written.

1.4 What is an LLM, and why should I care?

Generative AI for coding and language tools is based on the LLM concept. A large language model is a type of neural network that processes and generates text in a human-like way. It does this by being trained on a massive dataset of text, which allows it to learn human language patterns, as described previously. It lets LLMs translate, write, and answer questions with text. LLMs can contain natural language, source code, and more.

An LLM is a deep learning architecture based on the Transformer model—a significant architectural advancement over previous RNNs (Recurrent Neural Networks) and LSTMs (Long Term–Short Memory Networks) for sequence processing. Transformers employ multiple layers of self-attention mechanisms that process entire sequences in parallel rather than sequentially, vastly improving training efficiency and enabling the scaling to billions of parameters.

Imagine you have a smart system that reads sentences and tries to understand and generate text; this is what a Transformer does. It's a powerful technology that underpins many advanced applications today, including chatbots, automatic translation, and content generation. Here are the steps a transformer goes through, from input data to results from your prompts, as shown in figure 1.2:

- *Starting with words*—A Transformer begins by looking at your sentence word-by-word, turning each word into numbers that it can understand. Think of each word as getting its own special ID tag.
- *Remembering word order*—The Transformer doesn't just see words; it also pays attention to their positions. For example, in the sentence "Jane helps

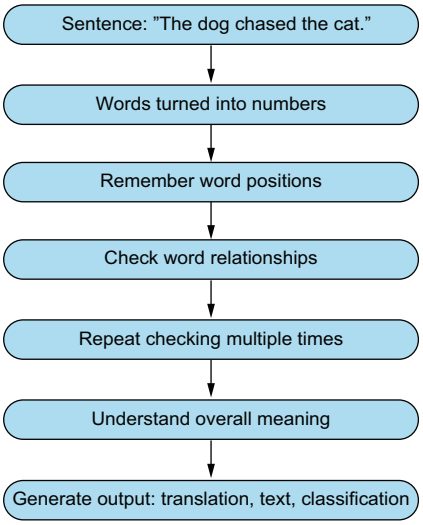

Figure 1.2 A transformer takes a sentence and analyzes word positions and relationships to try and extract meaning from the text it sees.

Joe," it knows "Jane" comes first and "Joe" comes last. It tracks this position information so it can better understand the meaning of your sentence.

- *Understanding context and meaning*—Now, the Transformer examines how words relate to each other. It checks each word against all the other words in the sentence. Imagine each word asking every other word, "Hey, how relevant are you to me?" Words that are closely related have stronger relationships, helping the Transformer understand context. For example, in the sentence "The dog chased the cat," the Transformer understands "dog" and "chased" have a strong connection.
- *Repeating this analysis multiple times*—The Transformer doesn't just do this once. It repeats this "checking relationships" step many times, each time learning something deeper about how words in the sentence connect. With each round, the system gains a clearer understanding of the sentence's overall meaning.
- *Producing the output*—After understanding the sentence, the Transformer can now use its knowledge to do different tasks:
  - *Translation*—It can convert text from English to another language.
  - *Text generation*—It can predict what words might naturally come next.
  - *Code generation (what we care about)*—It can predict the chunk of code that might come next.
  - *Classification*—It can recognize the overall meaning or sentiment behind a sentence.

Transformers excel at grasping relationships between words and concepts in sentences and documents. They greatly enhance tasks that were once challenging, such as natural language translation and understanding complex human questions.

When applied to code, these models use attention mechanisms. This feature helps them assess the importance of various parts of the existing codebase when generating suggestions. Attention mechanisms act like a spotlight, helping AI focus on what matters.

Picture yourself in a crowded room. You listen to one conversation while ignoring the noise around you. In coding, when an AI assistant suggests code, it doesn't see every line as equal. It uses attention to identify which parts are most relevant to its task.

For instance, if you're writing a function to calculate taxes, the AI will focus more on your tax rate variables. It will pay less attention to unrelated code, such as your login system. This way, it makes suggestions tailored to your project instead of generic ones.

This ability to zero in on important code is why modern AI coding assistants can offer meaningful suggestions for your specific needs. It's similar to how skilled developers know which code sections affect a new implementation the most. Each transformer layer learns about various code patterns, ranging from syntax validation to understanding the relationships among functions, classes, and modules.

The LLM is trained on vast amounts of text from sources such as books, articles, and websites. For example, GitHub Copilot learns from GitHub's public code base, which allows it to understand the semantic structures of both human language and code.

Once deployed, the LLM uses language patterns and context to create human-like text based on a prompt. It generates text and completes sentences to simulate conversation. For code tools, it aims to produce the most likely correct source code based on your input.

By adjusting parameters such as temperature (which controls randomness) and top k (which affects diversity), developers can tailor the model's output for different needs. This approach ensures the output is high quality and closely resembles human language.

1.5 Why do these tools sometimes get it wrong?

Generative AI tools for coding are sometimes inaccurate. They can produce results that look good but are wrong. This is common with LLMs. They can write code or chat like a person. And sometimes, they share information that's just plain wrong. Not just a bit off, but totally backwards or nonsense. And they say it so confidently! We call this "hallucinating," which is a funny term, but it makes sense.

So, why does this happen? Many people imagine the AI as a giant library, like the Library of Congress in a chip. You ask a question, it zooms to the right shelf, pulls out a book, and reads you the answer. That's not how it works at all. If it did, it wouldn't make things up; it would just say, "I don't know" if the book wasn't there.

So, what is it doing? Think about finishing a sentence. If I say, "The cat sat on the...," what word comes to mind? Probably "mat," right? Or maybe "chair" or "sofa." You're not looking it up; your brain just knows those words fit. You've heard them, read them, and seen them used.

LLMs do something similar, but on a huge scale. They've read a lot of the internet, tons of books, and millions of lines of code. They haven't "understood" it like we understand it, but they've learned the patterns. They know which words usually follow others and which bits of code often appear together.

When you give a prompt, the AI is not looking up the answer. It's predicting it, piece by piece. It thinks, "Okay, based on the prompt and what I've generated so far, what's the most likely next word? And the next? And the next?" It's like a super-powered prediction machine, always guessing the most statistically probable continuation based on learned patterns.

Now you see where the trouble can start. Sometimes, the most probable sequence of words, the one that fits best according to statistics, isn't true. The pattern it learned might come from faulty information online, or maybe your question was tricky. It could latch onto a pattern that seems right but leads to a wrong answer. It's just following statistical likelihood, not checking facts against a truth-database (because there isn't one!).

That's the key. It's a pattern-matching predictor, not a knowledge retriever. It's great at what it does, but since it works by prediction, it can predict nonsense just as confidently as it predicts facts. So, when you use these tools, be curious and skeptical! Don't just accept what it gives you. Ask, "Is this just a likely sounding pattern, or is it actually

right?" Understanding how generative AI works helps you know when to trust it and when to double-check. Keeping this skepticism in mind is crucial when working with these tools to produce code.

For most of us, the new concept of text *prediction* to generate answers is vague or even confusing. But we're getting the hang of it, and it's now a part of daily life.

1.5.1 How LLMs differ from databases

Another problem that can be confusing is that LLMs seldom put out the same thing twice. When starting out, many folks confuse ChatGPT and other LLMs with a large, all-knowing database. Instead, ChatGPT, and the tools we'll work with LLMs to generate new text based on mathematical probability.

Traditional databases are straightforward—you ask for something specific, and you get back exactly what was stored. Search engines work similarly, finding existing information.

LLMs work differently. They analyze massive amounts of text data to understand statistical patterns in language. The model processes information through multiple layers, each capturing different aspects—from simple word patterns to complex relationships between ideas.

When you input a question, the LLM goes through a series of steps to create a response:

- Processes your text through its mathematical model
- Calculates the relationships between different parts of your input
- Generates new text by predicting the most probable next word or character
- Repeats this process using both your input and what it just generated

The process of text prediction explains why:

- The same question can get different answers.
- Responses can be confidently incorrect.
- The model can create new combinations it hasn't seen before.

It's essentially a sophisticated prediction system. Instead of looking up stored answers, an LLM calculates probabilities to determine what text should come next. While these predictions are often accurate, they're still *predictions*—which is why it's crucial to verify any code or factual claims the model generates.

This probabilistic nature makes LLMs powerful tools for generating text and code but also means they can make mistakes, even when seeming very confident. Understanding this helps set realistic expectations about what these tools can and cannot do reliably.

1.5.2 Training phase problems

In machine learning, "training" is when we teach models to understand language and code by analyzing massive amounts of data. During training, the model learns statistical

patterns—how often certain words appear together, what code structures are common, and how different parts of text relate to each other.

The quality of training data directly affects how well the model performs. If the training data contains errors, incorrect code, or misleading information, the model will learn these flaws. Unlike humans, the model can't independently judge whether information is correct—it simply learns from everything it sees, treating all training data as equally valid.

When generating responses, the model uses probability calculations based on its training data to predict what text should come next. If it learned from flawed data during training, it may confidently generate incorrect or nonsensical output—called hallucinations—when the model produces convincing but incorrect information.

Think of it like teaching someone to code using both good and bad examples without telling them which is which. They'll learn patterns from both and might later write code that looks correct but contains hidden problems. This is why it's crucial to verify AI-generated code and not assume it's correct just because it looks good or because the model seems confident.

These training data problems are different from simple mistakes—they're systematic problems baked into the model during its learning phase. This is one reason why even advanced AI models need human oversight and verification.

For Python developers specifically, these training biases might manifest as

- A preference for common but suboptimal patterns (e.g., using lists where sets would be more efficient)
- Overuse of popular libraries even when simpler solutions exist
- Generating code that works for common cases but fails with edge cases
- Replicating outdated Python patterns from pre-3.6 codebases that don't use newer language features

This is why your expertise remains crucial. You can recognize when the AI is suggesting patterns that don't align with Python best practices or your project's architectural standards.

1.5.3 Misinterpreting context

Context is crucial for how language models understand and generate code. The model processes your input by analyzing relationships between different parts of the code and documentation to determine meaning and intent.

Common interpretation challenges in code generation are

- Variable names with multiple potential meanings
- Function overloading where context determines behavior
- API calls that could be valid but incorrect for the use case
- Syntactically correct code that violates semantic conventions
- Ambiguous requirements that could lead to different implementations

The model evaluates context by calculating mathematical relationships between elements in your input. However, it may miss important domain knowledge, coding standards, or architectural patterns that experienced developers understand implicitly.

While model training is fixed, you can improve code generation by

- Providing detailed specifications and constraints
- Including relevant code context (imports, dependencies, related functions)
- Specifying error handling requirements
- Declaring expected inputs and outputs
- Breaking complex features into smaller, focused components

The tools themselves do a great job of pulling in other parts of application into consideration, creating "context" for you. Later chapters will explore specific techniques you can apply to improve context and get more accurate and maintainable code from AI models.

1.6 The potential of LLMs

LLMs and development tools have been increasingly improving, and tens of thousands (or maybe more) of brilliant people are working every day to improve these models. They're making incredible progress, and the pace of improvement is staggering. The potential is enormous.

LLMs and development tools that derive from them greatly enhance productivity and software quality. By understanding the fundamental principles of language and communication, LLMs can provide intelligent assistance in tasks such as code generation, code analysis, and documentation. They will become a mandatory part of your toolbox. You can use these tools to assist you from the abstract design process down to the code syntax. When things break, you can find the reason faster and get assistance for fixing it.

We will write better quality software much faster. We'll test it better and provide better documentation. These tools can turn you into a super developer, and this book will help you set yourself up to reap these benefits.

1.7 Generative AI vs. code completion

It is likely that you have already encountered generative AI technologies as a software developer. When I discuss generative AI tools with other developers, they frequently mention IntelliSense, which was created by Microsoft for Visual Studio. They sometimes say "So, GitHub CoPilot is just another version of IntelliSense, right?" Having spent many years in the trenches as a C# developer with IntelliSense, I agree and disagree simultaneously. Let me explain.

IntelliSense is a code completion tool, very similar to tools in NetBeans, Eclipse, XCode, PyCharm, and many others that proliferated before AI. Generative AI is far more prescriptive. They're similar but not identical. Table 1.1. lists some of the differences.

Table 1.1 The differences between traditional code completion tools and generative AI

Function	Code completion	Generative AI
Input	Relies on predefined rules, syntax information, and available libraries	Utilizes machine learning models (LLMs) to analyze vast amounts of code from the internet and other sources
Output	Context-aware suggestions based on keywords, functions, and libraries	Suggestions based on patterns and relationships in code, and your coding style or patterns
Learning	Rule-based approach that relies on pre-defined knowledge of the language	Deep learning techniques from neural networks; the tool learns from the data it processes.
Scope	Limited to keywords, functions, and libraries available to the language; limited context of surrounding code	Code files, code blocks, functions, and libraries, as well as previously written code publicly available from the internet

Both types of tools are intended to help the developer be more productive in a similar way. How they work under the covers is different. Let's dig deeper into what makes generative AI the next stage of developer productivity tools.

1.7.1 Other types of generative AI

As discussed, generative AI is a type of artificial intelligence that can create new content. You've seen it sprouting up all over the place. AI-generated text, images, videos, music, and more can be produced to varying degrees of success. Each field is at a different level of maturity. For instance, text, music, and images are generally more mature than video and voice generation. All this content is generated by learning from existing data or training.

For example, with images, think of generative AI as a super-intelligent artist that has studied and memorized the works of thousands of painters over time, and watched them paint while gathering tips about their technique and composition. It keeps track of how each of these painters did what they did. So, when you ask the tool to create a cyberpunk image of Mona Lisa riding a skateboard in the desert, it can do it easily (figure 1.3).

Figure 1.3 Mona Lisa riding a skateboard in the desert. Generated with Midjourney, a generative AI image-building tool.

Generative AI differs from traditional AI systems, which focus on analyzing and processing data. Instead, generative AI learns patterns and relationships within the data and uses this knowledge to create something new and unique.

1.7.2 Why coders care about generative AI

You picked up this book to boost your productivity as a Python developer with generative AI coding tools. Let's explore the key AI features we'll master together and how they'll help you write better code faster:

- *Code completion and suggestion*—Tools such as GitHub Copilot, Tabnine, and Blackbox AI use generative AI to analyze your code and suggest snippets. Here, you'll learn how to use these tools effectively. This not only saves time but also lets you focus on complex problems while AI manages routine tasks, speeding up your development.
- *Automated testing*—Generative AI can create a variety of test cases to cover edge cases. You'll learn to use AI for generating test suites quickly. This improves your code's reliability and catches bugs early, a crucial skill we'll practice. Better testing leads to much stronger software.
- *Documentation generation*—Generative AI analyzes your code and comments to automate documentation. This book will show you how to simplify this often-boring task, ensuring your projects are well-documented with less effort. This makes your software easier to maintain and user-friendly.
- *Natural language processing*—Chatbots and virtual assistants, such as ChatGPT, utilize generative AI to understand and produce human-like text. While our focus is on coding, knowing how these tools process language is important. The skills you gain here can help you build better prompts and have better experiences with generative AI tools in general.
- *Vibe coding*—At the far end of the AI-assisted development spectrum lies an emerging approach called "vibe coding," a conversational programming style where developers guide AI with natural language rather than detailed specifications. While this book focuses on structured, production-ready AI integration, we'll explore how tools such as Cursor enable rapid prototyping through simple conversation in chapter 10. This is a new and different way of interacting with AI tools, and it has become popular. Vibe coding is great for "just for fun" projects and rapid prototyping, where ideas can become working demos in minutes rather than hours.

1.8 Project workflow with AI assistance

Incorporating AI assistance into your software development workflow can significantly improve productivity, reduce repetitive tasks, and elevate the overall quality of your projects. So long as you verify the output, you can speed up the development process and build better products faster.

Let's outline a typical project workflow that leverages the generative AI tools mentioned above to enhance various aspects of the software development process (figure 1.4).

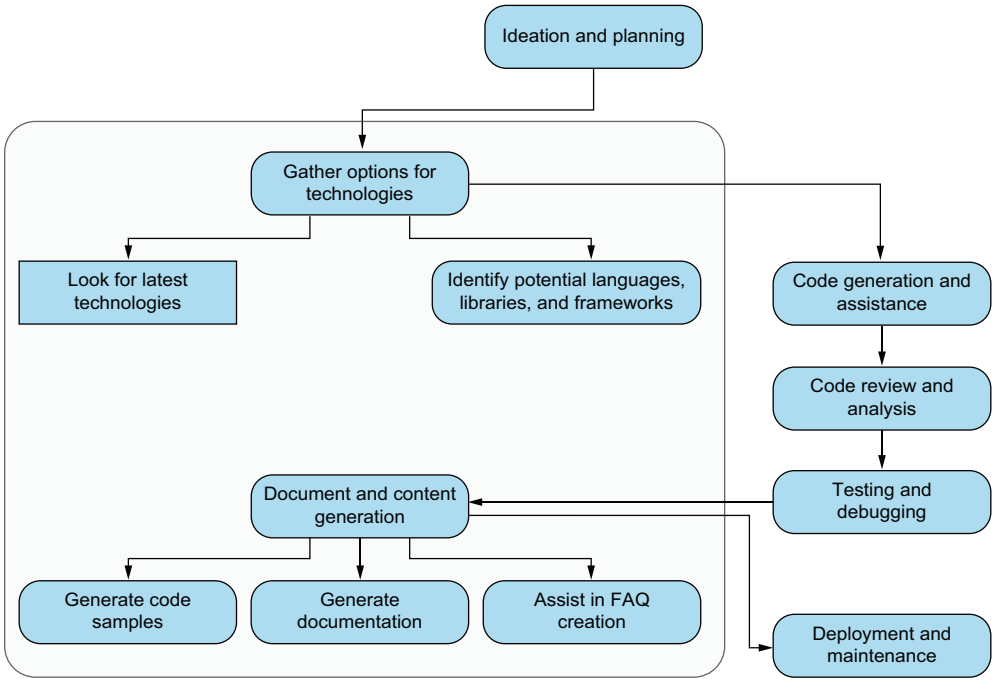

Figure 1.4 Generative AI tools play a big role in the planning process—from ideation and planning, to choosing a technology, and finally to building the software and deploying it. Each of these steps can be enhanced with AI tooling.

1.8.1 Ideation and planning

At this stage of the process, something chat-based such as Chat GPT, Gemini, or Bing is best to start with. Use plain English to describe what you want to do and send the tool prompts to get started, such as

JM Act as a software architect skilled in software design and development. I would like to create a TODO application where I can add a list of tasks to complete, and then mark them as completed. What tech stack should I use for a web-based version of this software?

Tools such as ChatGPT will give you a list of options for the frontend, backend, and database. Maybe you have a tech stack in mind. You could modify the prompt to say something like

JM Act as a software architect skilled in software design and development. I would like to create a TODO application where I can add a list of tasks to complete and then mark them as completed. I would like to build this with MariaDB, Flask, and Vue. How can I get started?

When sending this query to ChatGPT 4, I got this result:

Great! Here's a high-level overview of the steps you can take to build your TODO application using MariaDB, Flask, and Vue:

Database Design: First, you'll need to design the database schema for your TODO application. This will include creating a table to store the tasks, along with any relevant columns such as task name, description, due date, and completion status.

Set up MariaDB: Install MariaDB and set up a new database instance. Then, create the necessary tables and columns based on your database design.

Set up Flask: Install Flask, a popular Python-based web framework. Create a new Flask app and set up routes to handle CRUD (create, read, update, delete) operations on the task data.

Set up Vue: Install Vue.js, a popular JavaScript-based frontend framework. Create a new Vue.js app and set up components to display the task data and handle user interactions, such as adding a new task or marking a task as completed.

Connect Flask and Vue: Use Flask's REST API to expose the task data to your Vue.js app. You can use libraries such as Axios to make HTTP requests to the Flask API and retrieve the data.

Test and Deploy: Once you have the basic functionality working, test your application thoroughly to ensure it works as expected. Then, deploy your application to a web server or a cloud-based hosting service.

Some additional tips:

You can use SQLAlchemy, a popular Python-based ORM, to interact with the MariaDB database in Flask.

You may want to consider adding user authentication to your application to prevent unauthorized access to task data.

You can use Vue.js libraries such as Vuex and Vue Router to manage state and handle routing within your application.

Don't forget to add error handling and validation to your application to ensure it is robust and secure.

Overall, building a TODO application with MariaDB, Flask, and Vue.js should be a fun and rewarding project! Good luck!

You can use this conversational style to generate ideas and start an outline for your project to plan things out before getting into the code. This is also a good point to start documentation, such as a project proposal, feature descriptions, application requirements, and more, which will allow you to focus on high-level objectives.

1.8.2 Code generation and assistance

As you begin writing code, the integrated developer tools we mentioned earlier provide AI-powered code completions and suggestions. You can also use external tools to start "scaffolding" this code.

Scaffolding an application is generating a basic structure or framework for an application. This is similar to "create react app" with Node or "dotnet new" with C# applications. It involves automatically generating code, files, and directories that provide a starting point for development. This operation gives you a basic skeleton of an application you can complete and make whole.

I asked ChatGPT:

JM Please scaffold the routes of the backend API of this application in Python

As a result, it generated a full Python script with routes, a model, and methods automatically.

> **NOTE** While this prompt generates code that may be thorough and functional, do not blindly copy and paste it into your product, especially if you don't understand it. The code generated can be functional but not optimal and even dangerous if you do not fully understand it.

At this point, you want to write out your code and use the integrated tools to help you on your way.

1.8.3 Code review and analysis

External tools can be great for code review. You can use prompts such as

JM Evaluate the following code for performance, and identify any possible bottlenecks: (insert source code)

We will be doing this frequently later in the book.

External tools such as ChatGPT, Gemini, and Bing will evaluate your code and give you feedback. I have recently found that Gemini is very good at identifying bottlenecks and suggesting code improvements and rewrites. While ChatGPT is somewhat better for outlining and design, when I've dumped my own code into both, I've received some surprisingly good suggestions from Gemini, with little context. Figure 1.5 shows how you can drop in a function and request evaluation. Gemini doesn't know anything about how the code will be used but can still make suggestions.

You can perform similar actions for security as well. This is one of my favorite things to do with software. Use a prompt such as

JM Evaluate the following code for security issues, and identify any possible vulnerabilities:

```
(insert source code)
```

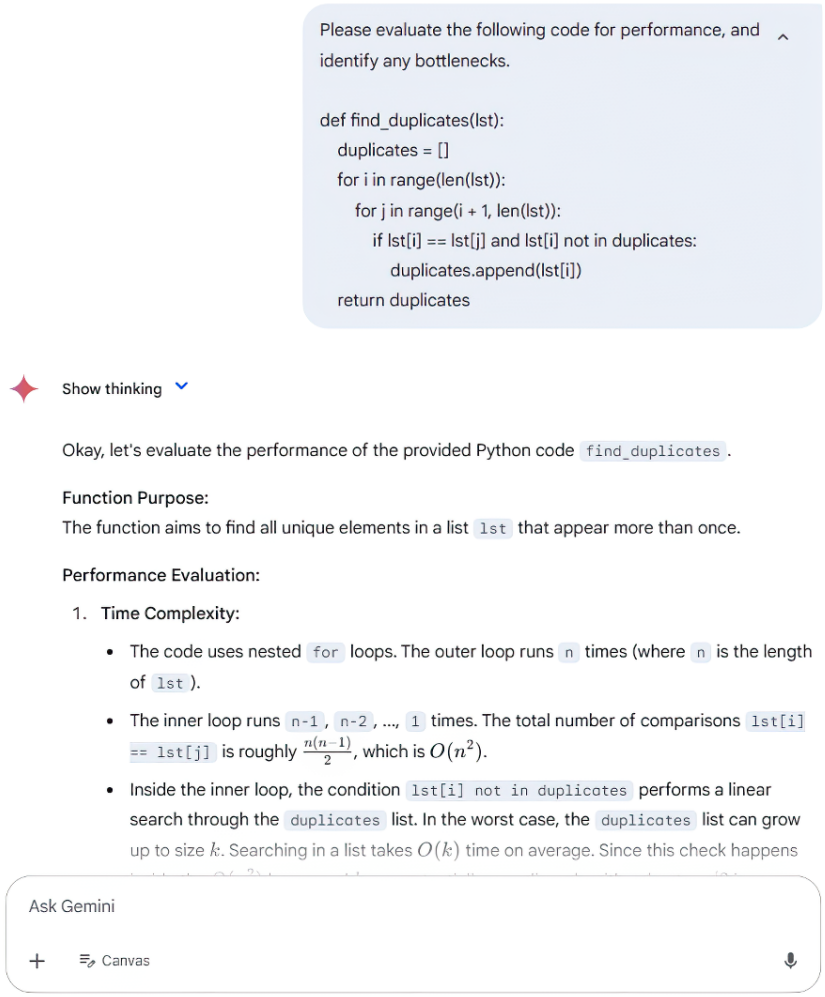

Figure 1.5 Even with a few lines of code, Gemini can analyze it and make suggestions to improve it.

In my experience, it has found glaring errors or provided suggestions to put you into a more security-focused mindset.

With Gemini, I've received replies that look like this:

> The code is well-written and easy to understand. It does not contain any obvious security vulnerabilities. However, there are a few things that could be improved to make it more secure.

ChatGPT gives similar results. These tips are crucial to changing the way you write software. It can give you tips you didn't know about or offer different ways of doing things you haven't thought about. We'll see that as we move through this book.

1.8.4 Testing and debugging

During the testing phase, you can use AI assistance to generate test cases and identify where software might fail. Integrated tools such as GitHub Copilot can generate tests for you. Tabnine has a beta version of a test generation component (as of this writing). Blackbox has a test generation component as well.

You can also use ChatGPT, Bing, or Gemini to generate unit tests. In my experience, ChatGPT has been great for debugging. If you can paste in code and an error message, it generates some good responses or places to look for problems. It will even tell you ways to debug or display information to help solve the problem. It's not perfect, but it's faster than a search engine, especially for tricky problems. We'll explore testing and debugging your code later.

1.8.5 Documentation and content generation

If you don't like writing documentation, AI tools are your friend. You can use the integrated IDE tools to some extent to generate documentation. Tabnine works very well at autocompleting code and API documentation. However, if you're creating a new document from scratch, I've found ChatGPT to be very good at this. You can dump in methods or even entire classes and spit out API documentation, code documentation, or human-readable instructions.

Be warned, however, it can be very dry. You want to use the output as *guidance* and put a human touch on it. There can be accuracy problems as well, so review it thoroughly.

By incorporating AI assistance throughout your software development workflow, you can employ generative AI tools to improve your process and help you develop faster. Using AI tools will save you time and effort and allow you to focus on more fun stuff.

1.9 Choosing the right generative AI tools

There are some considerations if you plan to use the generative AI tools discussed in this book. Whether you're an individual contributor, leader, or CEO, if you decide to utilize these tools, many of these factors apply to you.

1.9.1 Data quality and availability

As the old saying goes, "Garbage in, garbage out." Generative AI tools are only as good as the data they're trained on. They need high-quality, diverse, and extensive datasets to create great code as output. Unfortunately, you have no control over this input. You must trust the creators behind the product are using the best code possible for the *corpus*, or data used for training. Researching the tools lets you learn how each tool gathers data and decide based on that.

You must decide how much you value open source software for training versus proprietary software. Most tools use a combination of the two. Software trained on mostly open source software is a safe bet. A corpus derived from proprietary code can be more creative or meet a specific need. You must decide which one you value more.

The source code used to train these models provides unique challenges when it comes to licensing. If the tool is trained on licensed software and it generates something close enough to the original, there are some legal and copyright matters to think about. I'm no lawyer, so I can't help you much, and it's still a hotly debated topic in the field right now.

1.9.2 Integration with development workflows

Whether you're working solo or in an enterprise team with rigorous processes, as an experienced Python developer, you have likely established workflows. Integrating AI tools requires thoughtful consideration. If you are a solo developer,

- Evaluate how these tools interact with your existing IDE extensions and configurations
- Consider how to maintain consistency between AI-generated code and your personal coding style
- Develop strategies for validating AI suggestions against your domain-specific knowledge

For team environments,

- Establish team conventions for documenting which parts of the codebase used AI assistance (for future maintenance).
- Consider how AI tools interact with code review processes and standards.
- Address potential security concerns when AI tools process proprietary code.
- Ensure consistent access to these tools across your development team.

The learning curve varies across tools. While simple code completion might become intuitive within days, mastering more advanced capabilities such as architectural pattern generation could take weeks. Throughout this book, I highlight integration strategies that minimize disruption, while maximizing productivity gains.

1.9.3 Quality assurance

AI-generated code must function correctly and adhere to your organization's quality standards. Ideally you won't be using large amounts of AI generated code you don't understand, but it's still a consideration. These tools can speed up development and complicate code review and QA testing. You should always be transparent about your use of the tools and ensure they adhere to the standards expected of human-written code.

1.9.4 Keeping up with evolving tools

I don't have to tell you how fast these tools are evolving—you already know. You must stay abreast of the latest advancements in the tools and need to be familiar with advancements and new techniques. You must discern which of these tools and features are useful and not just a trend. And once again, it's a balancing act. You must determine if the long-term benefits are worth the churn of learning and re-learning the tool.

1.9.5 Shift in focus

In my career as a software engineer, I had to change my thinking from the day-to-day mechanics of writing code to more abstract skills. The shift from writing code to designing systems and architecture is a natural path for software developers. The AI tools accelerate that. and as you use the tools, you'll transition to thinking more like a designer than a coder much faster. Can you adapt to this rapid change? Can your team members?

1.10 Don't fear the rise of AI

A big concern among developers is the fear that AI will replace their jobs and render their skills obsolete. I agree that things will change, and some jobs will disappear. It's hard to deny that reality. Accenture (2023) estimates that language AI will support 40% of work hours as generative AI adoption grows (see https://mng.bz/pZo5). That same report says AI will spark creativity and innovation and "usher in an era of enterprise intelligence." Changes are coming.

While the concern is valid, it's sometimes overhyped. Software developer jobs won't disappear overnight. Things will change fast, and we're already seeing that. I don't have the psychic abilities to tell you what the future looks like, but nobody can deny that our industry will permanently change with generative AI. As a forward-looking developer learning about it, you're positioning yourself to stay ahead of the game.

Generative AI is a powerful ally rather than a threat. It's not a tool to replace you but a tool to make you more effective. By understanding and embracing generative AI now, you can harness its capabilities to augment your own skills. You *can* use it to streamline your workflow and create more innovative solutions. Consider generative AI as a smart assistant that can help you do what you do faster and more efficiently.

Generative AI is not a replacement for software developers. Here's why:

- *It handles the boring stuff.* Do you love writing boilerplate code? While it's not the worst part of the job, it's not fun. Writing repetitive code to do unexciting things is part of the job. It's more fun to create features, solve problems, and improve performance. Generative AI tools can generate repetitive code quickly, so you can focus on more interesting work. This interesting work is exactly what humans are good at, and AI (currently) isn't.
- *Humans still play the main role in problem solving.* AI driven tools will automate mundane tasks, write boring plumbing code, and even generate comments and documentation. AI can also generate tests very well. However, producing meaningful results still requires knowledge, intuition, and creativity. AI tools can generate code and make helpful suggestions. They don't fully understand all the context or your business. Critical decisions should still be made by the people involved.
- *Adaptability is required.* Nobody needs to tell you how fast this industry moves. It constantly evolves at an ever-increasing rate. Your challenges might be brand new, while your AI tool's knowledge may have stopped a year and a half ago.

Trends and best practices can easily change between the tool's deployment and when you're using it. Your ability to think critically, learn, and innovate is still better than that of AI. Having a good, experienced developer to gut-check the result from these tools is extremely valuable.

- *Ethical and moral considerations hold significance.* Generative AI is designed to be neutral. It has no opinions and no human sense of right or wrong. It's your job to ensure technology is used responsibly and ethically.
- *The human touch is still important.* Generative AI can produce impressive results. Folks in tech are using ChatGPT to build entire applications. However, remember your favorite AI tool isn't a person but clever automated assistant. It doesn't possess emotional intelligence or empathy. It doesn't understand context or experiences the same way you do. Your ability to connect with users and see things from their perspective is important. Your creativity and gut feelings are still needed in this space.

Some say the tools can encourage laziness and usher unchecked code into products. Or they can help people write software they don't understand. There is an element of truth to these possible side effects of code generation and the abstraction of code in general. However, like any tool, it depends on how you use it and how experienced developers influence the new folks. This is a larger problem we'll address later.

1.11 Go forth and code!

While we've covered the fundamentals, the true power of these tools emerges when you master advanced techniques explored in upcoming chapters:

- *Context management*—You'll learn how to structure your codebase and prompts to provide optimal context for AI tools, resulting in more accurate and relevant suggestions.
- *Prompt engineering*—We'll develop sophisticated prompting techniques that guide AI tools to generate precisely the code patterns your project requires.
- *Testing integration*—You'll discover how to generate comprehensive test suites that cover edge cases you might not have considered.
- *Architectural guidance*—Beyond just code completion, you'll learn how to use AI for higher-level design decisions and architecture validation.

Each of these techniques builds on the fundamentals covered in this chapter, transforming generative AI from a useful assistant into an indispensable development partner.

This book takes a hands-on approach to mastering generative AI tools. Each chapter builds on a core project—creating a full-featured web application—while introducing new AI tools and techniques. You'll start with basic code generation and progress to testing, documentation, and advanced features.

Rather than just reading about concepts, you'll learn by doing: writing prompts, evaluating AI responses, and integrating generated code into real applications.

By the end of this book, you'll be able to

- Use AI to speed up your coding workflow by 30%–50%.
- Generate and validate high-quality code and documentation.
- Choose the right generative AI tool for different coding tasks.
- Build complete applications using AI assistance.
- Debug and optimize AI-generated code.

You'll see practical examples, clear diagrams, and step-by-step instructions suitable for both beginners and experienced developers. Each chapter includes exercises to reinforce your learning and real-world scenarios you're likely to encounter in your development work. Later in the book we'll explore "vibe coding" where we use plain language to create software.

Join me in the next chapter as we roll our sleeves and build software with generative AI tools.

Summary

- Generative AI tools are changing software development. They boost productivity by generating code, finding bugs, and automating documentation.
- Modern AI tools come in two types: integrated tools such as GitHub Copilot and Tabnine, which work in your IDE, and standalone solutions such as ChatGPT and Gemini for bigger tasks.
- Large language models (LLMs) drive these tools. They learn patterns from massive code datasets, which help them create relevant code based on probabilities.
- It's important to know that AI tools make predictions and not certain outcomes. This understanding sets realistic expectations and shows why checking results is key.
- The AI-assisted development workflow includes ideation, planning, coding, testing, and documentation. Each phase gains unique benefits from AI support.
- To adopt AI tools effectively, consider training data quality, how they fit into workflows, quality assurance processes, and the need to adapt to new tools quickly.
- AI tools do not replace developers. Instead, they take care of routine tasks, letting humans focus on problem-solving, design, and creativity.
- As you read this book, you'll discover ways to use these tools, seeing them as valuable partners in your Python development process.