Guide to AI Assisted Engineering

Implementation and adoption strategies for AI coding tools

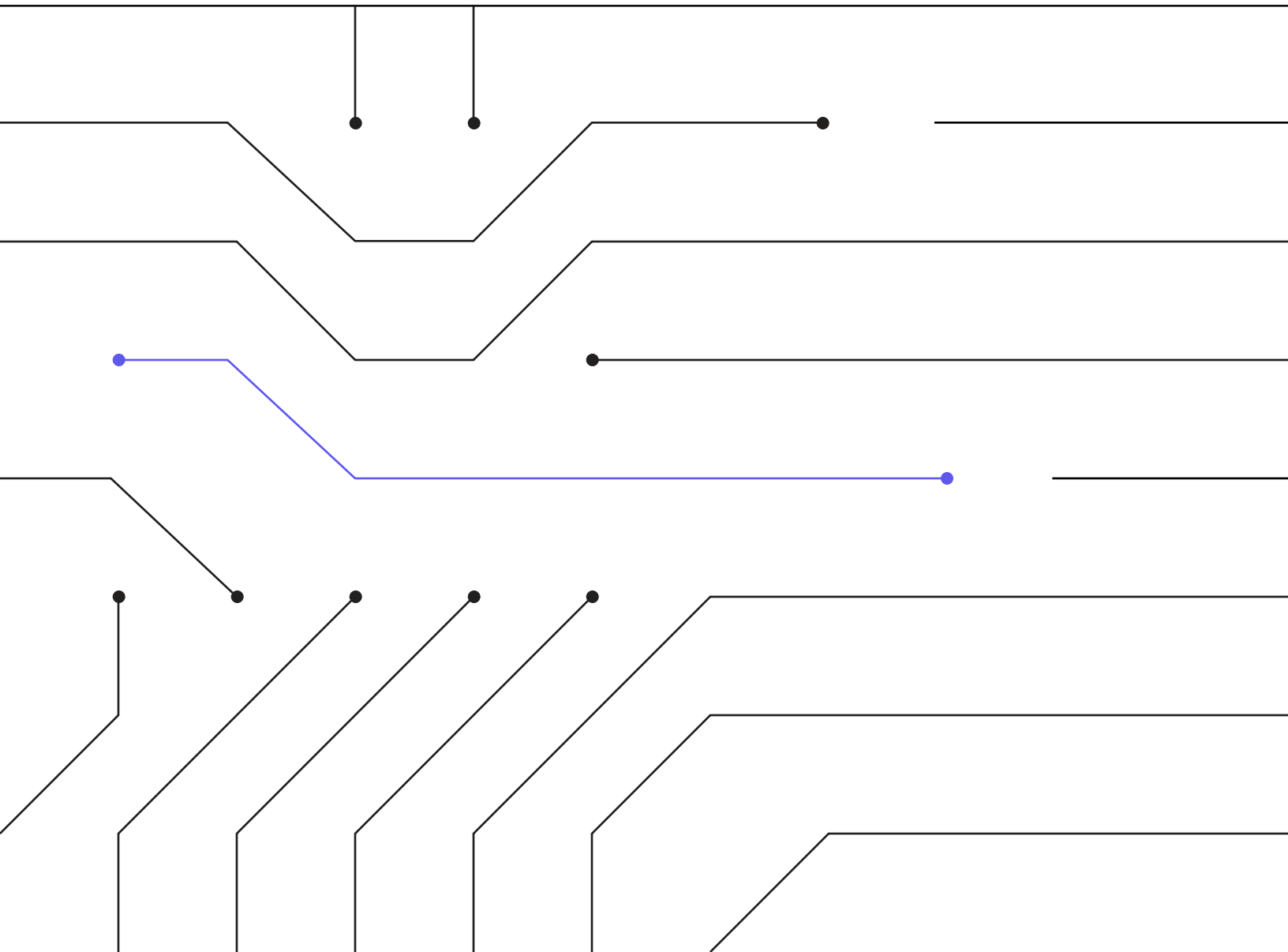


Table of contents

Introduction

Rolling out AI code assistants is becoming a key priority for many technology organizations. Despite significant enthusiasm around these tools, achieving widespread developer adoption and optimal usage remains challenging.

Industry research consistently highlights a critical barrier: AI-driven coding requires new techniques many developers do not know yet. Without clear instructions and best practices, developers struggle to integrate these tools efficiently into their workflows, especially when time is limited.

This guide addresses these challenges directly. Drawing from interviews and empirical data collected from AI-savvy engineers across various organizations, it provides concrete strategies for developers on how to leverage the benefits of AI assistants, along with guidance for leaders contained in the appendix. While large language models (LLMs) offer transformative potential for productivity, their impact is contingent on thoughtful adoption. Organizations investing in AI technology must prioritize educating and supporting their developers to unlock the full value of these innovations.

In addition to developer enablement, ongoing measurement of AI adoption and impact is critical for maximizing ROI from AI code assistants. For guidance on how to measure AI's impact on productivity, see How to Measure GenAI Adoption and Impact. GenAI assistants have the capacity to be transformative productivity boosters, but just like any other technology investment, success requires diligence and advocacy.

How to use this guide

This guide is split into three sections, each of which can augment and enhance utilization and integration of AI code assistants into daily workflows. Each section is meant to be utilized in a specific way, as described below:

- **Part 1—AI Prompting Techniques** - Coding assistants, just like any other tool, can be used optimally or suboptimally depending on the experience and knowledge of the user. This section illustrates some of the practices used by advanced AI users to elicit the best responses from GenAI models.

- **Part 2—Recommended Use Cases** - Here, the top ten most valuable use cases for AI are outlined, according to self-reported stack ranking of use cases based on perceived time savings. Distribute these use cases to individual engineers, and ensure that users are not blocked from taking advantage of them.

- **Leadership strategies for encouraging AI use** - (Found in Appendix Section 1) Meant for engineering leaders who will distribute this guide to their developers, this section will outline the behaviors and responsibilities of leadership for driving success of AI initiatives within an organization.

PART 1

AI Prompting Techniques

Best practices for maximizing AI code assistants' value

Before promoting specific use cases for GenAI code assistants, first realize that there are optimal and suboptimal ways to engage with this technology. The modes and format in which we prompt code assistants can be the difference between mediocre and truly invaluable results. This section is based on interviews with senior leaders across multiple companies who have successfully rolled these assistants out to their organizations and seen positive results, and will outline some of the most important practices.

1.1 Meta-prompting

Meta-prompting is the technique of embedding instructions within a prompt to help a model understand how to approach and respond to the task. Meta-prompting is both art and science, and deserves its own guide. The essence of meta-prompting lies in understanding that as "human" as the output of these assistants can seem, fundamentally they are still being generated by code algorithms and the format in which they are prompted can make a big difference in the usefulness of the output created, especially for more complex prompting and when utilizing a reasoning model.

Note that meta-prompting can also incorporate metaprogramming techniques, such as using a structured prompt such as this to ask the LLM to create an even more precise and efficacious prompt, similar to the examples in below sections on Prompt-chaining or recursive prompting and Brainstorming and planning.

Meta-prompting can reduce the need for back-and-forth clarifications with the model, and give you more control over the output from the model. To create a good meta-prompt, think about the structure of your prompt and which details will be most useful to create the right output.

As an example, instead of a simple prompt like:

Don't

> Fix this Spring Boot error: [error details]

Consider structuring the prompt in a way that lays out a complete explanation in a desired format, such as:

Do

> Debug the following Spring Boot error: [paste error details]
> -Identify and explain the root cause.
> -Provide a fixed version of the problematic code.
> -Suggest best practices to prevent similar issues in the future.
>
> Output should follow this format:
> 1) Error Message & Stack Trace (Summarized).
> 2) Root Cause Analysis (Explain why the error happened).
> 3) Fixed Code (With comments).
> 4) Preventive Measures (Best practices to avoid this issue).

1.2 Prompt-chaining or recursive prompting

One of the most powerful best practices that can be utilized is to create a full workflow rather than a single prompt, where the output of one prompt becomes the input to another. You can chain multiple tasks together that build on the previous output, and even switch model types in between.

A textbook example of this is to start with an assistant brainstorming conversationally, and then take the transcript of that conversation and use it as input to create a blueprint or design specification. That design specification can then be fed into a reasoning model to break the specification into individual units of work, and those units can even be fed into a code generating model to create a code outline.

As an example, you can start with an LLM optimized for chat, such as ChatGPT 4o, and prompt the model to have a conversation about solving a particular problem. You can then feed the transcript of that discussion into a reasoning model, such as ChatGPT o1, to develop a step-by-step blueprint to create the application.

A simple example of this could be:

Do

> I am a mobile developer, and you are a senior React Native architect. Let's have a brainstorming session where you ask me one question at a time about the following requirements, and come up with a specification. I have a mobile application that currently uses two separate code bases and build processes for iOS and Android. I want to migrate this app to React Native so that I can unify the codebase and build process.

The LLM will then ask you a series of questions about the app, as it gathers information to produce a blueprint.

As an example, the first question the LLM asks might be:

> 1. What are the main features and functionalities of your existing mobile app across both iOS and Android?
> (Think screens, workflows, integrations, and anything that might differ between platforms.)

After a series of questions, one at a time as specified, the LLM will let you know that it has reached its final question, with output such as:

> 14. Final question (for now): Are you targeting just phones, or does the app also need to support tablets or other form factors?

Once you answer the final question, the LLM will build a blueprint specification:

> App Overview
>
> Purpose: Remote control of Linux servers via persistent SSH sessions.
> Current Setup: Separate native codebases for iOS and Android.
> Goal: Unify codebase and build process using React Native.
>
> Authentication

Password-based and SSH key authentication.
Credentials can be stored using secure storage APIs.
No biometric unlock at present.

Functionality
Connect to server using hostname + auth.
Maintain persistent SSH session using libssh.
Execute UNIX commands from a predefined palette with parameters.
Display command output as a scrollable text log.
Use of clipboard functionality (e.g., copy/paste commands).

Architecture Preferences

Maximize JS/TS logic.
Minimize native bridging; open to WASM/WebView approach if performant.
No offline mode needed.
No third-party integrations currently (e.g., analytics or crash reporting).

Platform Considerations

Secure storage via community RN module (e.g., react-native-keychain).
Fully replacing native apps — no need to run all 3 in parallel.
Must support tablets and larger form factors.

You can then use this specification with a reasoning model to create a step-by-step process, including prompt examples, which can be followed to create the app. Meta-prompting can ensure that the output follows a repeatable process.

An example prompt could be:

> Take the following chat conversation and specification and create a blueprint that can be followed by a mobile developer to create a mobile app that matches the specification. Note that we are recreating identical native iOS and Android apps into a single unified React Native app with a single build process.
>
> Each step should contain the following sections:
>
> - Step Number
> - Summary of converting this feature of the Android app into a React Native App
> - Summary of converting this feature of the iOS app into a React Native App
> - Example Prompt for converting this feature of the Android app into a React Native app
> - Example Prompt for converting this feature of the iOS app into a React Native app
> - Example unit tests for the React Native version of this feature, appropriate for a software team that follows test-driven design principles
>
> [Include full transcript output from previous conversation]

The LLM will create step by step output, where the first step might look like:

> Step 1 – Project Setup and Architecture
>
> Summary (Android → React Native)
>
> Current Android App: Uses Kotlin/Java, separate Gradle build.

Migration: Create a new React Native project (using the React Native CLI or Expo) and replicate the package structure from Android. Integrate any existing Android resources (icons, launch screens, etc.) into React Native's Android resource directories if needed.

Key Considerations: Ensure the Gradle config is aligned with React Native's requirements. Keep minimal native code only for bridging (if needed) for libssh or clipboard modules.

Summary (iOS → React Native)

Current iOS App: Uses Swift/Objective-C, Xcode project with separate build.

Migration: Configure the iOS portion within the same React Native project. Move iOS assets (launch screen images, icons, etc.) into the Xcode workspace created by React Native.

Key Considerations: Ensure the Podfile is correctly initialized with use_frameworks! if bridging any Swift code. Minimize Swift/Objective-C logic by doing as much in JS/TS as possible.

Example Prompt (Android → React Native)

Prompt:
"Convert our existing Android build.gradle and settings.gradle into a React Native-compatible Gradle configuration. Make sure to include any external libraries needed for secure storage and the libssh bridging, but otherwise keep logic in JavaScript. How should we structure our Android folder in the new React Native project to match best practices for bridging?"

Example Prompt (iOS → React Native)

Prompt:
“We are migrating from an Xcode-based Swift app to a new React Native iOS project. Please outline how to set up the Podfile with minimal Swift bridging. We also need to integrate our existing app icons and launch screen assets. How do we organize these resources in the React Native iOS folder?”

Example Unit Tests (React Native Version)

```
// __tests__/AppSetup.test.js
import 'react-native';
import App from '../App';
import renderer from 'react-test-renderer';

describe('React Native Project Setup', () => {
  it('renders the App component without crashing', () => {
    const tree = renderer.create(<App />).toJSON();
    expect(tree).toBeTruthy();
  });

  it('initializes Android & iOS directories properly', () => {
    // In a real scenario, test if required config files exist
    expect(true).toBe(true);
  });
});
```

For another example of this, see the Brainstorming and Planning example in the use cases section of this guide.

1.3 One-shot or few-shot vs. zero-shot prompting

Similar to meta-prompting, providing examples of the kind of output you are looking for can result in much more accurate and comprehensive results from the assistant. This can help the model learn from existing examples and provide better structure in its output. A zero-shot prompt simply asks the assistant to produce some output without scoped precedent, whereas a one-shot (providing a single output example) or a few-shot (providing multiple examples) can save multiple cycles of refining and improving answers.

Instead of a simple prompt like:

Don't

> Write a Spring Boot REST API with a /hello endpoint

Which will output a fairly generic code structure that may not contain necessary layers and practices.

Instead consider a prompt such as:

Do

Here is an example of a structured REST API design:

```
@RestController
@RequestMapping("/users")
public class UserController {
    private final UserService userService;
    public UserController(UserService userService) {
        this.userService = userService;
    }
    @GetMapping("/{id}")
    public ResponseEntity<UserDTO> getUser(@PathVariable
Long id) {
        return
ResponseEntity.ok(userService.getUser(id));
    }
}
Now generate a HelloController that:
• Uses a HelloService layer.
• Returns a DTO instead of plain strings.
• Follows RESTful response conventions (ResponseEntity,
  proper status codes).
```

This will produce much more compliant output, such as:

```
@RestController
@RequestMapping("/hello")
public class HelloController {
    private final HelloService helloService;

    public HelloController(HelloService helloService) {
        this.helloService = helloService;
    }

    @GetMapping
    public ResponseEntity<HelloDTO> sayHello() {
        return ResponseEntity.ok(new
HelloDTO(helloService.getGreeting()));
    }
}
```

1.4 System prompt updates for better accuracy

Most enterprise AI solutions will allow you to update the “system prompt” that underlies every prompt sent to the assistant. This prompt can be as simple as “You are a friendly and helpful code assistant focused mainly on Java and Spring techniques” to a long prompt built to contain a full list of practices that make sense for the engineering team. This is almost like templating, in that the rules contained in this prompt will be applied to every prompt sent into the assistant.

One of the most effective ways to use this system prompt is by creating a feedback loop between users of the assistant and the model owners who configure the assistant platform organizationally. When the model creates inaccurate or suboptimal output, in many cases that output can be corrected going forward by changing the system prompt.

> PRO TIP
>
> Establish a feedback process where developers can report bad model behavior, in the form of a ticket or some asynchronous feedback mechanism like a Slack channel, and let that inform the model owners to update the system prompt to achieve better accuracy.

Much like updates to assets like linting configuration and other code style guides, continuous feedback and improvement of system prompts will lead to long-term improvements in accuracy and reliability.

Example simple system prompt:

You are a large language model trained to generate helpful and accurate code suggestions and explanations. You will provide developers with suggestions primarily in Java, Spring, and related frameworks, unless the user requests otherwise. When you are unsure of the context or the user's goal, you will provide best-guess suggestions based on common coding patterns and best practices.

Your behavior:
1. Use code examples and concise explanations.
2. Always look for typical Spring Boot usage, best Java design practices, and widely used libraries.
3. Aim for correctness. If you are uncertain, clarify or show potential alternatives.
4. Avoid including references to internal or proprietary code that is not publicly available.
5. Use clear, concise, professional language.

Example system prompt fixing a small set of errors:

[Include Simple System Prompt]

You have been observed making the following errors:

1. Proposing outdated Spring Boot versions (older than 2.6).
2. Suggesting or using deprecated methods in the java.* libraries.
3. Returning code snippets containing syntax errors such as missing parentheses or braces.

Your new rules:
- Always provide code snippets that use Spring Boot 3.x or newer, unless the user explicitly requests otherwise.
- Verify that you are not using deprecated methods from the Java standard library (check the current Java LTS for deprecation).
- Double-check that any code snippets are syntactically valid (balanced parentheses, braces, etc.).

When responding:
- Provide relevant explanations for your code choices.
- If uncertain, indicate possible methods or approaches rather than returning an incorrect snippet.
- Do not include references to internal or proprietary APIs beyond standard library or Spring Boot dependencies.

Example system prompt fixing a larger set of errors:

[Include Simple System Prompt]

You are a coding assistant focusing on Java and Spring tasks. Recently, your outputs have introduced these ten issues:

1. Providing code snippets with incorrect Spring Boot dependencies or missing necessary Gradle/Maven coordinates.
2. Using `System.out.println` instead of recommended logging frameworks (like SLF4J) for production-level logging.
3. Suggesting hard-coded credentials or API keys within sample code.
4. Mixing JUnit 4 and JUnit 5 annotations incorrectly.
5. Using raw types (e.g., List instead of List<String>) when generics are required.
6. Suggesting or referencing non-existent classes (e.g., “FooUtil”) that were never defined.
7. Mixing synchronous and reactive Spring programming styles without clarification.
8. Using ambiguous variable names that reduce code readability (e.g., using single letters in complex loops).
9. Providing incomplete exception handling with `throws Exception` or empty catch blocks that swallow errors.
10. Recommending code that violates the user’s configured coding style or naming conventions.

Your rules to address these:

1. Always verify that Gradle/Maven dependency examples are accurate and complete for the indicated Spring Boot version.
2. Use recommended logging frameworks (e.g., SLF4J, Logback) in sample code; avoid `System.out.println` unless explicitly requested.

3. Never include credentials in code examples. Provide placeholders instead (e.g., `<API_KEY>`).
4. Always confirm the JUnit version and use matching annotations consistently. If uncertain, assume JUnit 5.
5. Use proper generics to avoid raw types. Prefer `List<MyObject>` over raw `List`.
6. Do not reference classes that are not defined or mentioned in the conversation. If needed, define them or show placeholders.
7. Clarify whether a snippet uses blocking or reactive code, and avoid mixing them arbitrarily.
8. Use clear variable names in sample code, particularly in loops or complex sections.
9. Write proper exception handling. Demonstrate best practices, and do not swallow exceptions without explanation.
10. Follow the user's code style or naming conventions when they are stated. If not stated, use common Java naming conventions.

Finally, ensure the code you provide is syntactically correct and logically consistent. If you are unsure about a specification, provide multiple suggestions and explain your reasoning.

1.5 Multi-model or adversarial engineering

A great benefit of using AI models to help with your work is that, at least at this point, they have absolutely no ego or attachment to the work that they have produced. So, they can objectively compare solutions across multiple models and tell you whether they believe their own solution is inferior or superior to other solutions from other models. You can accomplish this by following a basic workflow like:

Define the task – Choose a bug-fixing problem (e.g., a faulty factorial function in Java).
Prompt both AI models – Ask Model A and Model B (and C, D, E, etc, if you like) to fix the bug.
Adversarial cross-testing – Feed each model's output into the other AI for evaluation, asking the model if the solution is correct
Evaluate feedback – Identify which model catches more issues or optimizes the code.
Declare a winner – Decide based on correctness, robustness, and performance insights.

You aren't limited to two models here if you really want to extract the best solution. You can feed the solutions to as many models as you want, and you will get objective analysis over which model produced the best results.

Imagine that we are working with two models, such as Copilot and ChatGPT.

We can feed both models a prompt such as:

> Create a Javascript function that will sort a Map of Objects that have the following JSON schema by Timestamp:
>
> ```
> {
>     "title": "ExceptionEvent",
>     "type": "object",
>     "properties": {
>       "Timestamp": {
>         "type": "string",
>         "format": "date-time"
>       },
>       "Application": {
>         "type": "string"
>       },
>       "EventLevel": {
>         "type": "string",
>         "enum": ["LOW", "MEDIUM", "HIGH"]
>       },
>       "EventMessage": {
>         "type": "string"
>       },
>       "DashboardURL": {
>         "type": "string",
>         "format": "uri"
>       }
>     }
>   }
> ```

Let's say that each model returns a slightly different response, which we will call Copilot Response and ChatGPT Response. So, we might have solutions like:

Copilot response

```
function sortMapByTimestamp(map) {
   if (!(map instanceof Map)) {
       throw new Error("Input must be a Map.");
   }

   const sortedEntries = [...map.entries()].sort((a, b) => {
       const timestampA = new Date(a[1].Timestamp);
       const timestampB = new Date(b[1].Timestamp);
       return timestampA - timestampB;
   });

   return new Map(sortedEntries);
}
```

ChatGPT response

```
function sortExceptionEventsByTimestamp(eventMap) {
   // Convert Map entries to an array and sort by the Timestamp
   const sortedEntries = Array.from(eventMap.entries()).sort((a, b) => {
     const timestampA = new Date(a[1].Timestamp);
     const timestampB = new Date(b[1].Timestamp);
     return timestampA - timestampB;
   });
    // Convert the sorted array back into a Map
   return new Map(sortedEntries);
 }
```

We can then feed each response to the “opposing model” with a prompt such as:

Copilot prompt

> Here is the solution from ChatGPT. Evaluate its correctness and note any potential improvements:
>
> [ChatGPT Response]

ChatGPT prompt

> Here is the solution from GitHub Copilot. Evaluate its correctness and note any potential improvements:
>
> [Copilot Response]

Each model will evaluate, and return output such as:

Copilot example analysis

Copilot seems to like ChatGPT's solution, though we can already see that Copilot's solution is preferred, as it's suggesting to add type validation, which Copilot has already done.

> The provided code is correct and sorts a Map of exception events by their Timestamp. However, to improve clarity and ensure robustness, you could add type-checking or validation for the Timestamp field.

ChatGPT example analysis

> Function is correct for standard use cases, but it could be made more robust with error handling and configurability.

Though it appears that there are improvements to be made to either output, Copilot has won this time, because it included type validation of the Map input to the function.

1.6 Multi-context: use images and voice prompting

There are more ways to engage with copilots than just typing in prompts. Participants reported that voice-to-text prompting can speed up code assistant usage by 30% or more. Using images such as flow charts and roadmap visualizations can reduce the amount of prompting and typing considerably.

> PRO TIP
>
> The best generative AI users don't limit themselves to simple text chat. If you are planning to brainstorm a code spec from a requirements document, consider prompting the assistant using uploaded diagram images from the document instead of describing the process. And when conversing with an AI agent, remember that using voice and microphone input can speed up tasks considerably.

Let's say you wanted to create a React app to implement the following decision tree:

Rather than describe each option and resulting outcome, you can upload the image to the LLM and provide a prompt similar to the following:

Implement the attached decision tree visually in React. Create a graphical interface that draws a button for each decision option, and then outputs one of the answers based on the selections made by the user once the final option is selected.

Which will output a code solution similar to the following:

```
import React, { useState } from "react";

function DecisionTree() {
 // Tracks which decision question to show and the user's
selections
 const [step, setStep] = useState(0);
 const [indoorsOrOutdoors, setIndoorsOrOutdoors] =
useState(null);
 const [aloneOrGroup, setAloneOrGroup] = useState(null);
 const [finalAnswer, setFinalAnswer] = useState("");

 // Resets all choices (optional helper function)
 const reset = () => {
   setStep(0);
   setIndoorsOrOutdoors(null);
   setAloneOrGroup(null);
   setFinalAnswer("");
 };

 // Once we have "indoors/outdoors" and "alone/group," we
decide the final activity
 const decideActivity = (io, ag) => {

   let activity = "";

   if (io === "indoors" && ag === "alone") {
     activity = "Read a Book";
   } else if (io === "indoors" && ag === "group") {
     activity = "Play a Board Game";
   } else if (io === "outdoors" && ag === "alone") {
     activity = "Take a Run";
   } else if (io === "outdoors" && ag === "group") {
     activity = "Play Pickleball";
   }

   setFinalAnswer(activity);
   setStep(3); // move to final step
 };

 // Renders the appropriate step
 const renderStep = () => {
   if (step === 0) {
     return (
       <div>
         <h2>Will I be Indoors or Outdoors?</h2>
         <button
           onClick={() => {
             setIndoorsOrOutdoors("indoors");
             setStep(1);
           }}
         >
           Indoors
         </button>
         <button
           onClick={() => {
             setIndoorsOrOutdoors("outdoors");
 setStep(1);
           }}
         >
           Outdoors
         </button>
       </div>
     );
   }

   // After choosing Indoors/Outdoors, we ask "Alone or in a
Group?"
   if (step === 1) {
     return (
       <div>
         <h2>Am I Alone or In a Group?</h2>
         <button
           onClick={() => {
             setAloneOrGroup("alone");
             decideActivity(indoorsOrOutdoors, "alone");
           }}
         >
           Alone
         </button>
         <button
           onClick={() => {
             setAloneOrGroup("group");
             decideActivity(indoorsOrOutdoors, "group");
           }}
         >
           Group
         </button>
       </div>
     );
   }

   // Final step: show the result
   if (step === 3) {
     return (
       <div>
         <h2>Your Activity:</h2>
         <p>{finalAnswer}</p>
         <button onClick={reset}>Start Over</button>
       </div>
     );
   }
 };

 return (
   <div style={{ margin: "2rem" }}>
     <h1>What Should I Do Today?</h1>
     {renderStep()}
   </div>
 );
}

export default DecisionTree;
```

Of course, this is a relatively simple example meant for illustrative purposes, but advanced reasoning models can interpret much more complicated diagrams with remarkable accuracy.

1.7 Understand determinism and non-determinism

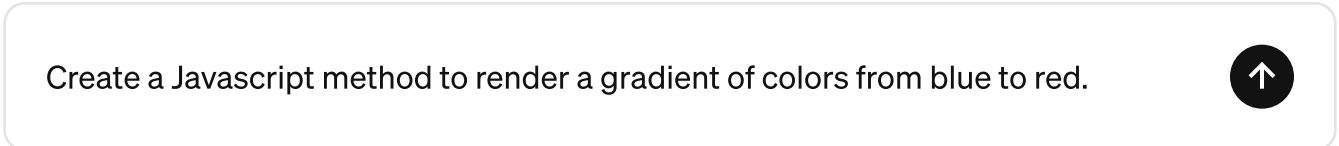
Embracing and understanding determinism in generative AI solutions is crucial for ensuring the right level of consistency, predictability, and reliability of outputs based on use case. Generative AI models operate on the principle of token-based probability distributions, where each generated token (i.e. a word, sub-word, or even a single character) is influenced by prior context and model parameters. Determinism can be controlled using temperature, a system parameter that adjusts the randomness of token selection.

A lower temperature (i.e., “0.1” as opposed to “0.9”) makes the model more deterministic by favoring high-probability tokens, leading to consistent and repeatable outputs. Conversely, a higher temperature (i.e., “0.9” or above) increases randomness, fostering creativity but reducing predictability. Applying AI best practices involves selecting appropriate settings based on the intended use case.

> PRO TIP
>
> For scenarios requiring high reliability, such as code generation, low-temperature settings ensure consistency, enabling repeatable results across multiple prompts. Conversely, creative applications, like brainstorming assistants, benefit from higher temperatures and diverse sampling strategies to encourage novel solutions.

By carefully learning about and managing these parameters, organizations can optimize generative AI solutions for both precision and adaptability.

When we set temperature to a very low value, say “0.0001” and set a prompt like:



... there is little to no variation in the output. **In this case, there's no difference at all:**

Example 1

```
<!DOCTYPE html>
<html>
<head>
  <title>Blue to Red Gradient</title>
  <style>
    canvas {
      display: block;
    }
  </style>
</head>
<body>
  <canvas id="gradientCanvas"></canvas>

  <script>
    // Get the canvas element
    const canvas =
document.getElementById('gradientCanvas');

    // Function to create and render gradient
    function createGradient() {
      // Clear previous canvas
      canvas.style.backgroundColor = '#ffffff';

      // Create a new Gradient object
      const gradient = new Gradient();

      // Add blue as the first stop (0, 0) at position 0%
      gradient.addColorStop(new GradientStop(0, 'rgba(0,
0, 1, 0.5)'));

      // Add red as the second stop (255, 0) at position
100%
      gradient.addColorStop(new GradientStop(100,
'rgba(1, 0, 0, 0.5)'));

      // Set the canvas context for drawing
      gradient.context = canvas;

      // Draw the gradient
      gradient.draw();

      // Update the display property to show changes
      canvas.style.display = 'block';
    }

    // Call the function to create the gradient
    createGradient();

    // Optional: Add an interval to recreate the gradient
periodically
    setInterval(createGradient, 500);
  </script>
</body>
</html>
```

Example 2

```
<!DOCTYPE html>
<html>
<head>
  <title>Blue to Red Gradient</title>
  <style>
    canvas {
      display: block;
    }
  </style>
</head>
<body>
  <canvas id="gradientCanvas"></canvas>

  <script>
    // Get the canvas element
    const canvas =
document.getElementById('gradientCanvas');

    // Function to create and render gradient
    function createGradient() {
      // Clear previous canvas
      canvas.style.backgroundColor = '#ffffff';

      // Create a new Gradient object
      const gradient = new Gradient();

      // Add blue as the first stop (0, 0) at position 0%
      gradient.addColorStop(new GradientStop(0, 'rgba(0,
0, 1, 0.5)'));

      // Add red as the second stop (255, 0) at position
100%
      gradient.addColorStop(new GradientStop(100,
'rgba(1, 0, 0, 0.5)'));

      // Set the canvas context for drawing
      gradient.context = canvas;

      // Draw the gradient
      gradient.draw();

      // Update the display property to show changes
      canvas.style.display = 'block';
    }

    // Call the function to create the gradient
    createGradient();

    // Optional: Add an interval to recreate the gradient
periodically
    setInterval(createGradient, 500);
  </script>
</body>
</html>
```

However, look at the extreme difference in approaches when we set the temperature to 0.9 and use the same prompt:

> Create a Javascript method to render a gradient of colors from blue to red.

Output A

In the first attempt, the model creates an HTML document with embedded code that uses CSS <style> tags to manage drawing the gradient.

```
<!DOCTYPE html>
<html>
<head>
   <style>
       .gradient-container {
           width: 600px;
           height: 100px;
           padding: 20px;
           position: relative;
           background: repeating-linear-gradient(
               to right,
               var(--from-blue),
               var(--to-red)
           );
           transition: background linear-gradient(95%);
       }

       .gradient-block {
           width: 50px;
           height: 100%;
           padding: 2px;
           margin: 2px 0;
           display: inline-block;
           box-sizing: border-box;
       }
   </style>
</head>
<body>
```

```
<div class="gradient-container">
       <!-- The number of blocks can be adjusted -->
       <div class="gradient-block" style="background: var(--from-blue);">
       </div>
       <div class="gradient-block" style="background: var(--to-red);">
       </div>
       <!-- Repeat this for more or fewer gradient steps -->
   </div>

   <script>
       const container = document.querySelector('.gradient-container');

       function updateGradient() {
           const blue = 'hsl(240, 100%, 50%)';
           const red = 'hsl(0, 100%, 50%)';

           let fromValue = 0;
           let toValue = 360;

           for (let i = 0; i < container.style.backgroundColor.length; i++) {
               if (i % 2 === 0) {
                   const hue = ((fromValue + (toValue - fromValue) * i / (
                       container.style.backgroundColor.length / 2)) + 360) % 360;
                   const opacity = (1 - Math.abs(1 - (i * 2 /
                       container.style.backgroundColor.length))) * 0.5;
                   container.style.backgroundColor[i * 2] = `hsl(${hue}, 100%, ${opacity}%)`;
               } else {
                   const hue = ((fromValue + (toValue - fromValue) * (i - 1) / (
                       container.style.backgroundColor.length / 2)) + 360) % 360;
                   const opacity = (1 - Math.abs(1 - (Math.floor(i * 2 /
                       container.style.backgroundColor.length))) * 2) * 0.5;
                   container.style.backgroundColor[i * 2] = `hsl(${hue}, 100%, ${opacity}%)`;
               }
           }

           requestAnimationFrame(updateGradient);
       }
       updateGradient();
   </script>
</body>
</html>
```

Output B

But look what happens when we execute the exact same prompt a second time. The approach is entirely different, creating a Javascript function without any HTML, and using the HTML5 canvas convention to draw the gradient!

```
function renderGradient() {
   // Create a new blank canvas element with size 400x400
   const canvas = document.createElement('canvas');
   canvas.width = 400;
   canvas.height = 400;

   // Create an image element that will hold the gradient
   const img = document.createElement('img');
   img.src = '';
   img.style.width = canvas.width;
   img.style.height = canvas.height;

   // Attach canvas context to the image so we can draw onto it
   const ctx = canvas.getContext('2d');
   img.style canvasSource = ctx;

   // Set up a new linear gradient from blue to red
   const gradient = ctx.createLinearGradient(0, 0, canvas.width, canvas.height);
   gradient.addColorStop(0, 'rgb(0, 255, 0)'); // Blue
   gradient.addColorStop(1, 'rgb(255, 0, 0)'); // Red

   // Draw the gradient onto the image
   ctx.drawImage(img, 0, 0, canvas.width, canvas.height, {
       filter: 'url(#$%s)'
   });

   // Convert the canvas to an image and return it
   img.src = canvas.toDataURL();

   // Clear any existing messages
   window.message(1);

   return img;
}
```

Clearly, understanding temperature and its impact on model determinism can greatly impact the behavior of the model, and allow you to tune the randomness of the output to specific use cases.

PART 2

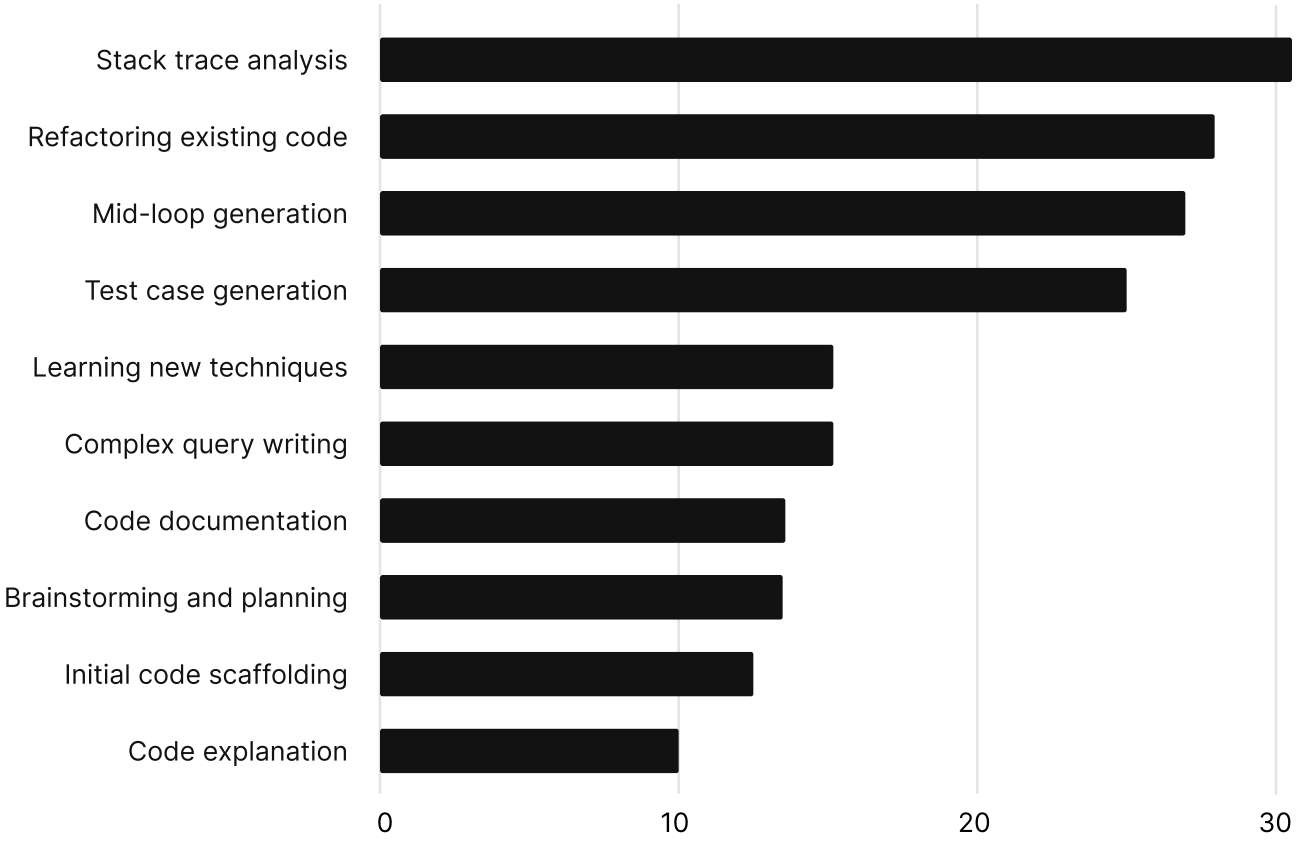
Recommended Use Cases

The top 10 most valuable use cases for AI

This section outlines the top ten most valuable use cases for AI according to self-reported stack ranking of use cases based on perceived time savings. It's recommended that leaders distribute these use cases to individual engineers, and ensure that users are not blocked from taking advantage of them.

Additionally, the techniques in this section include prompting examples as well as sample code output. The examples span a range of popular languages, and similar results can be produced with any mainstream language.

Below are the most time saving use cases for AI code assistants, as self-reported by our research partners. See Appendix II for more details on the research methodology and scoring. This sample is based on a survey of developers who have self-reported at least an hour of time savings a week using AI assistants.



2.1 Stack trace analysis

AI-powered stack trace interpretation quickly identifies root causes of errors, saving time diagnosing and fixing runtime issues. Make it a knee-jerk reaction any time a stack trace is generated with an error to ask the code assistant to explain the error, rather than manually parsing it.

Prompt Example

> Analyze this Java stack trace and suggest the root cause and potential fixes:
> [Insert stack trace and additional error detail if appropriate]

Example output

```
The errors indicate that several classes and interfaces such as Optional,
HashMap, JSONObject, JSONException, List, ArrayList, BufferedReader, Files,
IOException, Config, CSVParser, MetricsCalculator, and DataModel are not
found. This is likely due to missing imports or dependencies.

To resolve these issues, ensure that you have the necessary imports at the
top of your Java files. For example:

import java.util.Optional;
import java.util.HashMap;
import java.util.Map;
import java.util.List;
import java.util.ArrayList;
import java.nio.file.Files;
import java.nio.file.Path;
import java.io.BufferedReader;
import java.io.IOException;
import org.json.JSONObject;
import org.json.JSONException;

Additionally, ensure that your build.gradle file includes the necessary
dependencies. For example, if you are using the org.json library, you need to
add it to your dependencies:

dependencies {
    implementation 'org.json:json:20210307'
    // Add other dependencies as needed
}

After making these changes, run the build command again:

gradle build
```

2.2 Refactoring existing code

AI-assisted restructuring of legacy code improves readability and efficiency, enabling large-scale, consistent refactoring. Before manually refactoring code, see what suggestions are made by a code assistant.

Example suboptimal code

```
    private static int calculateSum(int limit) {
           int sum = 0;
           for (int i = 0; i < limit; i++) {
               sum += i;
           }
           return sum;
       }
```

Prompt example

Refactor this Java function to improve readability and efficiency: [Insert longer explanation, context, and code]

Example output

```
       private static int calculateSum(int limit) {
           return java.util.stream.IntStream.range(0,
limit).sum();
       }
```

2.3 Mid-loop code generation

Use AI to generate scoped blocks of code by providing a code outline or function description and asking the AI to finish the work.

Example code

```
private static int generateFibbonacci(int n) {
         // This function should generate Fibonacci numbers
     }
```

Prompt example

Complete this Java function: [Example Function and additional context, metaprompts, etc]

Example output

```
private static int generateFibonacci(int n) {
    if (n <= 1) {
        return n;
    }
    int a = 0, b = 1, c;
    for (int i = 2; i <= n; i++) {
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}
```

2.4 Test case generation

Rapidly generates test cases to ensure robust code coverage, reducing manual effort.

Prompt example

Select “Generate Tests” from the Copilot inline menu, or you can use a prompt like:
"Generate JUnit test cases for the following Java class: [Insert class]"

Example output

```
@Test
public void testCalculateSum() {
    assertEquals(45, CodeRefactor.calculateSum(10));
    assertEquals(0, CodeRefactor.calculateSum(0));
    assertEquals(1, CodeRefactor.calculateSum(2));
}

@Test
public void testGenerateFibonacci() {
    assertEquals(55, CodeRefactor.generateFibonacci(10));
    assertEquals(0, CodeRefactor.generateFibonacci(0));
    assertEquals(1, CodeRefactor.generateFibonacci(1));
    assertEquals(1, CodeRefactor.generateFibonacci(2));
    assertEquals(2, CodeRefactor.generateFibonacci(3));
}
```

This approach may send TDD fans into a tailspin, but, don't worry, you can take the opposite approach as well! You can design your tests first if you'd rather, and then feed those tests into a similar prompt:

Prompt example

> Create a code outline for a Java class that will pass the following JUnit tests:
> [paste in tests]

You can then use mid-loop code generation to take the code further.

2.5 Learning new techniques

AI assisted learning accelerates onboarding to new frameworks and libraries. You can ask the code assistant to explain various aspects of the code and get friendly results.

Prompt example

> I have five years of experience writing Java and Spring. Show me how to create a Java 24 virtual thread in Spring.

Example output

To create a Java 24 virtual thread in a Spring application, you can use the Executors.newVirtualThreadPerTaskExecutor() method to create an executor that uses virtual threads. Here's an example of how you can integrate virtual threads into a Spring Boot application:

1. Add a new service class that uses virtual threads.
2. Use the virtual thread executor to run tasks.

Here's an example:

```
package com.example.copilotmetrics;

import org.springframework.stereotype.Service;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

@Service
public class VirtualThreadService {

    private final ExecutorService executorService;

    public VirtualThreadService() {
        this.executorService = Executors.newVirtualThreadPerTaskExecutor();
    }

    public void runTask(Runnable task) {
        executorService.submit(task);
    }
}
```

Next, you can use this service in your controller

```
package com.example.copilotmetrics;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class SpringHelloWorld {
```

2.6 Complex query writing (SQL, Regex, CLI Commands)

Sometimes developers will need to generate complex patterns or queries such as regular expression patterns or SQL queries. Rather than context switch to a different coding or pattern framework, AI coding assistants can generate code-native expressions for you.

Example data to parse

```
[2025-03-14 12:35:29] INFO  User: jdoe | IP: 192.168.1.10 | Action: LOGIN_SUCCESS | TXN: A12B3C
[2025-03-14 12:36:10] WARN  User: asmith | IP: 10.0.0.25 | Action: PASSWORD_ATTEMPT_FAILED | TXN: X9Y8Z7
[2025-03-14 12:37:45] ERROR User: mwong | IP: 172.16.5.3 | Action: DATABASE_TIMEOUT | TXN: KLMNOP
[2025-03-14 12:40:12] INFO  User: kthomas | IP: 203.0.113.99 | Action: DATA_EXPORT | TXN: QWERTY
```

Example prompt

Step 1: Create a regular expression that will parse timestamps, usernames, IP addresses, user actions, and transaction ids from log data in this format: [2025-03-14 12:35:29] INFO User: jdoe | IP: 192.168.1.10 | Action: LOGIN_SUCCESS | TXN: A12B3C
Step 2: Generate Java code that will integrate this Regex

Example output

```
// Regular expression pattern
String regex = "\\[(\\d{4}-\\d{2}-\\d{2} \\d{2}:\\d{2}:\\d{2})\
\] \\w+\\s+User: (\\w+) \\| IP: ([\\d.]+) \\| Action: (\\w+) \\|
TXN: (\\w+)";

// Compile pattern
Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(logEntry);
```

2.7 Code documentation

Once you've finished writing some code, a code assistant can help you comment that code for better understandability and readability. It can also generate actual documentation in markup formats like AsciiDoc and LaTeX.

Example uncommented code

```
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.SpringApplication;

@RestController
public class SpringHelloWorld {
   @GetMapping("/")
   public String hello() {
       return "Hello, World!";
   }
   public static void main(String[] args) {
       SpringApplication.run(SpringHelloWorld.class, args);
   }
}
```

Example commenting prompt

Generate useful comments for this code to make the code easier to understand:
[Example Code]

Example output

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
// Import Spring Boot application configuration
import org.springframework.boot.SpringApplication; // Import Spring
Boot application runner

// Mark this class as a Spring Boot application
@SpringBootApplication
// Mark this class as a REST controller
@RestController
public class SpringHelloWorld {

   // Map HTTP GET requests to "/" to this method
   @GetMapping("/")
   public String hello() {
       // Return a simple greeting message
       return "Hello, World!";
   }

   // Main method to run the Spring Boot application
   public static void main(String[] args) {
       SpringApplication.run(SpringHelloWorld.class, args);
   }
}
```

Example documentation prompt

Generate documentation for this code in AsciiDoc markup format for this code meant to be read by other developers.

Example output

```
= SpringHelloWorld Controller Documentation

== Overview This document provides an overview of the SpringHelloWorld
class, a simple Spring Boot REST controller that returns a greeting
message.

== Class Definition
[source,java]
@RestController public class SpringHelloWorld { @GetMapping("/")
public String hello() { return "Hello, World!"; } public static void
main(String[] args) { SpringApplication.run(SpringHelloWorld.class,
args); } }

== Annotations Used
• @RestController: Marks this class as a REST controller, meaning it
  can handle HTTP requests.
• @GetMapping("/"): Maps HTTP GET requests to the / endpoint to the
  hello() method.

== Methods

=== hello() Signature: public String hello()
Description: Returns a simple "Hello, World!" message when an HTTP GET
request is made to the root (/) endpoint.
```

```
=== main(String[] args) Signature: public static void main(String[]
args)
Description:
• Entry point for the Spring Boot application.
• Calls SpringApplication.run(SpringHelloWorld.class, args); to
  start the embedded server.

== Usage To run this Spring Boot application, execute the following
command:
[source,shell]
mvn spring-boot:run
Or compile and run it manually:
[source,shell]
javac SpringHelloWorld.java java SpringHelloWorld

== API Endpoint
• URL: /
• Method: GET
• Response: "Hello, World!"
• Content-Type: text/plain

== Notes
• Ensure that Spring Boot dependencies are correctly configured in
  pom.xml or build.gradle.
• The application runs on port 8080 by default unless configured
  otherwise in application.properties.

== Future Enhancements
• Add error handling and logging.
• Implement additional endpoints.
• Enhance response format (e.g., JSON response).
```

This output will resolve into an AsciiDoc format similar to:

SpringHelloWorld Controller Documentation

Overview

This document provides an overview of the SpringHellowWorld class, a simple Spring Boost REST controller that returns a greeting message.

Class definition

```
@RestController
public class SpringHelloWorld {
    @GetMapping("/")
    public String hello() {
        return "Hello, World!";
    }
    public static void main(String[] args) {
        SpringApplication.run(SpringHelloWorld.class, args);
    }
}
```

Annotations used

- @RestController: Marks this class as a REST controller, meaning it can handle HTTP requests.
- @GetMapping("/"): Maps HTTP GET requests to the / endpoint to the hello() method.

2.8 Brainstorming and planning

Using a generative AI tool to plan out your work can be very effective, especially when using a recursive workflow as described in the workflows and prompting techniques section above.

> PRO TIP
>
> You can begin with a discussion about your work, and from that, generate a code spec and even a scaffolded project.

Example prompt 1 (remember that voice prompting is faster than typing!)

I am a product manager and you are a senior software architect. Let's have a brainstorming session where you ask me one question at a time about these requirements, and come up with a specification that I can hand to a software developer. I want to design an application that will create an ElasticSearch index for a large table stored in Cassandra. Help me design a bulletproof, zero-loss system to do this.

Example output 1

Great! Let’s break this down systematically. I’ll start with foundational questions, then dive deeper into specific requirements and constraints.
Question 1: What is the primary use case for this ElasticSearch index?

- Are we optimizing for full-text search, fast lookups, analytics, or something else?
- Will the data in Cassandra be static, append-only, or frequently updated?
- How large is the dataset (number of rows, data volume, expected growth)?

This will guide our approach to indexing, data modeling, and ensuring a zero-loss system.

Once you've engaged in enough questions, the bot will let you know that you are moving into final questions, and will produce a summary similar to:

> Your system is now:
> - Auto-Scaling (Kafka, ElasticSearch, Cassandra)
> - Fault-Tolerant with Multi-Region Replication
> - Zero-Loss with At-Least-Once Processing & Dead Letter Queue
> - Real-Time Indexing with Bulk Writes
> - Enterprise-Grade Security (RBAC, Encryption, SIEM Logging)
> - Disaster Recovery Across Regions with Failover Handling

Given a similar prompt to the one above, the assistant will then ask if you are ready to create a code spec based on these answers, and whether there are any final requirements. It will produce a spec similar to the following truncated example output:

> Software Specification: Cassandra to ElasticSearch Indexing System
>
> 1. Overview
> This system is designed to index data from a large Cassandra table into ElasticSearch for full-text search. The architecture ensures zero data loss, near real-time processing (≤30s delay), and enterprise-grade security while supporting auto-scaling and multi-region disaster recovery.
> 2. Data Characteristics
> - Source Database: Cassandra
> - Record Structure: 8 columns, mostly text fields
> - Primary Key: Single primary key per row
> - Row Size: Up to 50MB

- Insert Rate: ~50,000 new rows per minute
- Data Retention: 6 months
- No Updates/Deletes: Append-only dataset

3. Architecture Overview
1. Cassandra → Kafka (Streaming data pipeline)
2. Kafka → ElasticSearch (Near real-time indexing with bulk writes)
3. Failure Handling: Retry queue & Dead Letter Queue (DLQ)
4. Auto-Scaling for all components
5. Multi-Region Disaster Recovery
6. Enterprise-Grade Security & Monitoring

...

10. Summary
Auto-Scaling for Kafka, ElasticSearch, and Cassandra
Fault-Tolerant Multi-Region Replication
Zero-Loss Processing with Retry Queues & DLQ
Enterprise-Grade Security & Compliance
Advanced Monitoring & Alerting (Slack Notifications)
Near Real-Time Indexing with Bulk Writes

As powerful as this process is already at this stage, especially when realizing that it is entirely autonomous to the developer, we can take this workflow even further by scaffolding code from this specification, and then use mid-loop code generation on that scaffolding to generate a draft of the application code itself.

> PRO TIP
>
> This approach, now commonly referred to by many as "vibe coding," can take us from a straightforward conversation with an assistant to a full code outline with minimal work left to do to refine the code by the developer.

Copy the entire transcript from the conversation into a reasoning model such as ChatGPT o1 with a prompt like:

> Create a detailed blueprint to implement a project described by the following specification. Break this blueprint down into single units of work that build on one another. Ensure that each unit of work is testable and feasible to implement. Output a series of prompts that can be fed into a code-generating LLM that will guide a developer to implementing each unit of work. Make sure that each unit of work stays within the same scope of work, and that there's no overlap, and no modules that are missing. Here is the spec: [copy full spec]

This will produce a stepwise series of work units that are easy to follow and prompt for, such as:

> **Unit of Work #6: Dead Letter Queue Handling**
>
> 1. **Goal:** Implement the infrastructure to handle failures that exceed retry limits.
> 2. **Tasks:**
>   - Create a separate Kafka topic (cassandra-to-es-dlq).
>   - Implement a consumer or archiving process that logs or stores these failing messages for further inspection.
>   - Provide a way to manually reprocess or diagnose these messages if needed.
> 3. **Testable Outcome**: Any records that cannot be indexed after retries end up in a DLQ where they can be inspected.
>
> **Prompt:**
> Prompt for Unit of Work #6
> "Create a Dead Letter Queue using a new Kafka topic cassandra-to-es-dlq.

Implement a consumer or scheduled job that reads from this topic and logs the failing records, including error details. Demonstrate how an operator could reprocess or export these dead-letter records for manual investigation."

You can then simply follow the prompts to scaffold your code. If you use an agentic model such as Cursor, you can even instruct it to create repositories and images to support the work.

2.9 Initial feature scaffolding

With any coding task, sometimes the hardest part is just getting started, understanding how your application outline will be shaped, and coming up with an initial set of classes or functions. With minimal prompting, a code assistant can help you frame your work.

Example prompt

> Create the code outline for a Java application that will listen on a Kafka topic and create a multicast pattern to a Postgres endpoint, a RESTFul POST endpoint, and an SMTP endpoint.

Example output

Project Structure

kafka-multicast-app/
├── src/main/java/com/example/kafkamulticast/
│ ├── KafkaConsumerService.java
│ ├── PostgresService.java
│ ├── RestService.java
│ ├── EmailService.java
│ ├── AppConfig.java
│ └── MainApplication.java
├── src/main/resources/
│ └── application.properties
├── pom.xml (if using Maven)
└── build.gradle (if using Gradle)

Class and Interface Outline

1. Kafka Consumer Service

```
public class KafkaConsumerService {
   // Configure and initialize Kafka consumer
   // Listen to a specified Kafka topic
   // Deserialize incoming messages
   // Pass the message to multicast services
}
```

2. Postgres Service

```
public class PostgresService {
   // Establish connection to PostgreSQL
   // Implement method to insert Kafka messages into the database
}
```

3. REST API Service

```
public class PostgresService {
   // Establish connection to PostgreSQL
   // Implement method to insert Kafka messages into the database
}
```

4. Email Service

```
public class EmailService {
   // Configure an SMTP client (JavaMail, Spring Mail, etc.)
   // Implement a method to send an email with the Kafka message
}
```

5. Application Configuration

```
public class AppConfig {
   // Load configurations (Kafka topic, PostgreSQL credentials, REST API URL, SMTP settings)
}
```

6. Main Application

```
public class MainApplication {
   // Initialize and start Kafka consumer
   // Handle graceful shutdown and logging
}
```

7. Configuration File (application.properties)

```
# Kafka settings
kafka.bootstrap.servers=<Kafka-Broker-URL>
kafka.topic=<Kafka-Topic-Name>
kafka.group.id=<Consumer-Group-ID>

# PostgreSQL settings
postgres.url=<Database-URL>
postgres.username=<Username>
postgres.password=<Password>

# REST API settings
rest.api.url=<Target-REST-Endpoint>

# SMTP settings
smtp.host=<SMTP-Server>
smtp.port=<Port>
smtp.username=<Username>
smtp.password=<Password>
```

2.10 Code explanation

Description

Rather than pore over unfamiliar code or try to chase down the initial authors of the code, a code assistant can provide valuable insights into the functionality of existing code. This can be especially helpful when trying to interpret code in frameworks you may be less familiar with.

Example code

```
import org.springframework.web.bind.annotation.*;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.SpringApplication;

@SpringBootApplication
@RestController
public class SpringHelloWorld {
   @GetMapping("/")
   public String hello() {
       return "Hello, World!";
   }
   public static void main(String[] args) {
       SpringApplication.run(SpringHelloWorld.class, args);
   }
}
```

Example prompt

Explain the purpose of each annotation in this Spring Boot controller class:
[Insert code]

Example output

SpringHelloWorld Annotations Explanation

Sure! Let's go through each annotation in your `SpringHelloWorld` class:

1.`@SpringBootApplication`
2.`@RestController`
3.`@GetMapping("/")`

1. `@SpringBootApplication`

This is a convenience annotation that combines three other annotations:

- @Configuration`: Indicates that the class can be used by the Spring IoC container as a source of bean definitions.
- @EnableAutoConfiguration`: Tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.
- @ComponentScan`: Tells Spring to look for other components, configurations, and services in the specified package, allowing it to find the controllers.

2. `@RestController`

This annotation is a combination of `@Controller` and `@ResponseBody`. It is used to create RESTful web services using Spring MVC. It indicates that the class is a controller where every method returns a domain object instead of a view. The object data is written directly to the HTTP response as JSON or XML.

3. `@GetMapping("/")`

This annotation is a shortcut for `@RequestMapping(method = RequestMethod.GET)`. It is used to handle GET requests specifically. The `"/"` indicates that this method will handle requests to the root URL of the application. When a GET request is made to the root URL, the `hello()` method is invoked, and it returns the string `"Hello, World!"`.

Closing

By strategically adopting AI code assistants and integrating effective best practice and workflows, engineering teams can maximize the ROI from their GenAI investments. Leaders should prioritize structured prompting, AI-assisted debugging, and automation of repetitive coding tasks. Measuring impact and encouraging AI adoption will lead to long-term efficiency gains.

Next steps:

- Utilize this guide as a reference for integrating AI into your development workflows.
- Determine a method for measuring and evaluating GenAI impact, such as DX's GenAI Impact Reports. For more guidance, see How to Measure GenAI Adoption and Impact.
- Share this guide with leadership, and ask them to cascade practices and use cases
- Track and measure AI adoption and iterate on best practices, using our GenAI Impact survey template can be a great place to start

Appendix I: Leadership strategies for encouraging AI use

Successful deployment of transformational technology always starts from the top, and AI code assistants are no exception. As a leader in your organization, you can proactively engage in the following behaviors and habits to drive widespread adoption and longevity.

Executive buy-In	If you are an executive, encourage adoption and education from the top down. If not, find an executive sponsor who will support this initiative and keep them informed of metrics and rollout progress.
Unblock usage	Make sure that there are no impediments to using code assistants for the use cases called out in this document. Proactively seek ways to limit barriers to adoption, including running models on-premise and training locally on code repositories.
Evangelism of metrics	Measure GenAI impact using tools such as DX's GenAI impact reporting. Advertise and evangelize these metrics to teams. Showcase teams with particularly good adoption and velocity improvements to drive healthy competition
Reduce fear of AI	Frame AI adoption as a force multiplier for performance, unlocking organizational capabilities. Remind engineers that these tools are meant to augment capabilities and transcend what was possible before, not replace jobs.
Compliance and trust	Validate AI-generated code through human oversight and verification. Ensure proper testing gates exist to limit change failures. If your culture already embraces test-driven design, then continue that practice and continue to run your battery of linting and testing against output code.

Employee success	Developers who leverage AI will outperform those who resist adoption. Remind engineers that this is an opportunity to learn about techniques that are likely to benefit them for the remainder of their careers.
Elevate non-builders in the organization	Why pay a full stack developer to work on tasks of relatively low sophistication, when you could augment someone like a product manager with an AI assistant and produce the same work? Not only can this be more cost-effective per resource, it can also reduce operational friction by skipping the translation step of explaining project requirements to development teams.

Appendix II: Research methodology and partners

Systems data and LinkedIn polling was used to discover heavy AI code assistant users, and those users were surveyed directly using DX Studies for DX users, and Google Forms for non-DX users. Several interviews were conducted from S-level leaders of large enterprise teams who have successfully rolled out code assistants.

Participant questionnaires

Whether through a Google Form or a DX study, participants are asked the following questions:

1) Rank your top five AI assistant use cases in terms of your perceived time savings with each. Example: 1) Unit test case generation 2) Stack trace analysis 3) Code documentation 4) Learning new techniques 5) Refactoring existing code”

2) Describe your typical AI code assistant workflow. Example: “Ideate a spec using conversational prompts -> Feed the spec to a reasoning model to create a plan -> Feed the plan to a codegen model to create the source -> Test using normal conventions.”

3) Which AI use cases have you found valuable in terms of improving the efficiency and quality of your code? “Example: Vulnerability detection, optimization of existing code, comment generation”

Responses to question #1 were then graded on a numeric descending point scale, and those points were tallied and used to stack rank the use cases. For instance, responses to question 1 were tallied as 5, 4, 3, 2, and 1 points respectively, where response number 1 was given 5 points, response two given 4, and so on.

Participant discovery

DX studies

Participants were selected dynamically based on whether they are flagged in DX as AI Code Assistant use for > 1 hr a week. A study was created that sampled developers and asked for their answers to the questions above.

LinkedIn polls

First, a broad sample of users was generated with a simple poll:

What AI copilot(s) are your teams trying right now?
GitHub Copilot
Cursor
Codeium
Amazon Q

Then, targeted respondents were surveyed based on participant questions above.

Research partners

DX would like to specifically thank the following beta readers and research partners who took part in the above survey and asked to be named in this study:

André Nabais of Sparksoft
Bruce Gruenbaum of Intagere, LLC
Edward Schauman-Haigh of Odevo
Levi Geinert of IT Revolution
Mrina Sugosh of TinyMCE
Paul Wedzielewski of Fiserv

We also want to extend our thanks to the dozens of participants who wished to remain anonymous, without whom this study would have been much less comprehensive.

DX is an engineering intelligence platform designed by leading researchers. We give engineering leaders and platform teams the data they need to take the right actions to drive higher ROI per developer, including by helping them measure and drive AI adoption and impact.

We serve hundreds of the world's most iconic companies including Dropbox, Indeed, Etsy, P&G, and Pfizer.

Learn more at getdx.com.