Innovation Insight for ML-Powered Coding Assistants

Published 21 November 2022 - ID G00777224 - 12 min read

By Analyst(s): Arun Batchu, Van Baker, Manjunath Bhat

Initiatives: Software Engineering Technologies; Adopt Modern Architectures and Technologies

A new generation of coding assistants for professional developers are demonstrating not only longer and novel completions, but also being able to use comments to generate code. Software engineering leaders should exploit these innovative tools to increase the productivity of their teams.

Additional Perspectives

- Summary Translation: Innovation Insight for ML-Powered Coding Assistants (09 March 2023)

Overview

Key Findings

- Code generation products based on foundation models are able to generate complex and longer suggestions resulting in a significant increase in developer productivity.
- These products are able to understand the intent of the developer better by using surrounding comments and code as input. Developers and models thus work as a programming “pair” to solve problems.
- Provenance and licensing compliance of code used for training, the runtime performance of the models, and quality of generated code are critical concerns and potential weaknesses.

Recommendations

Software engineering leaders responsible for software engineering technologies should:

- Pilot machine learning (ML)-powered coding assistants, with an eye toward fast rollouts to maximize developer productivity.

- Handpick a few software engineers to master the new skill of "prompt engineering," using a combination of natural language and coding practices to figure out how to optimize code generation with minimal effort.
- Form and lead a cross-functional team to identify, monitor and solve for emerging challenges in copyright infringement, code quality and nonperformant code generation.

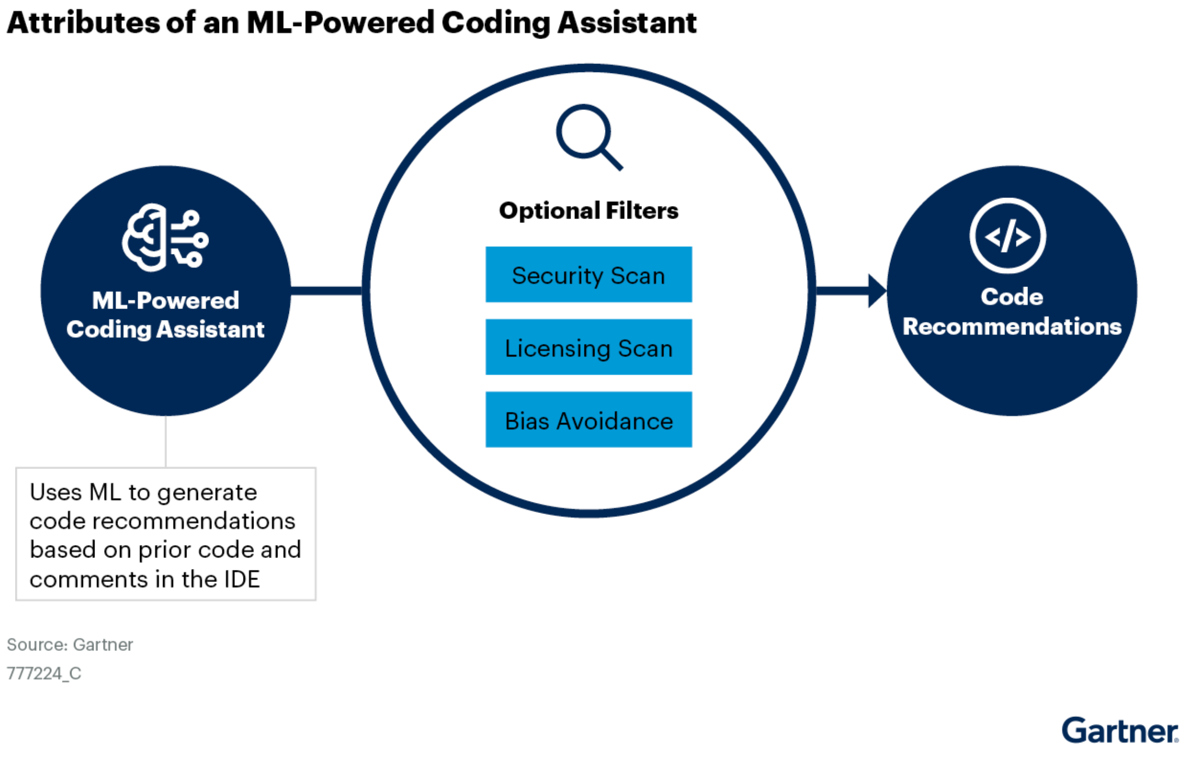
Strategic Planning Assumption

By 2027, 50% of developers will use ML-powered coding tools, up from less than 5% today

Introduction

Code completion tools in modern integrated development environments (IDEs), such as Virtual Studio Code (VS Code) and IntelliJ, have become essential for a developer to handle code complexity. However, developers are overwhelmed because rule-based engines are not able to keep pace with the rapid growth of enterprise code and the open-source code dependencies.

Figure 1: Attributes of an ML-Powered Coding Assistant


Attributes of an ML-Powered Coding Assistant
ML-Powered Coding Assistant
Optional Filters
Security Scan
Licensing Scan
Bias Avoidance
Code Recommendations
Uses ML to generate code recommendations based on prior code and comments in the IDE
Gartner

Source: Gartner
777224_C

As an answer to this acute problem, researchers and innovative products have recently demonstrated that artificial intelligence (AI) models trained on a large corpus of source code and natural language texts can generate longer, complex code suggestions based on developer input in the form of comments and surrounding code (see Note1: Evolution of ML-Powered Coding Assistants). According to claims by vendors, developers are accepting a good portion of these suggestions, decreasing coding creation times and reducing the number of context switches.

However, new challenges around productivity impact, optimal interaction design, quality of the generated code, intellectual property attribution and bias in generated snippets are emerging.

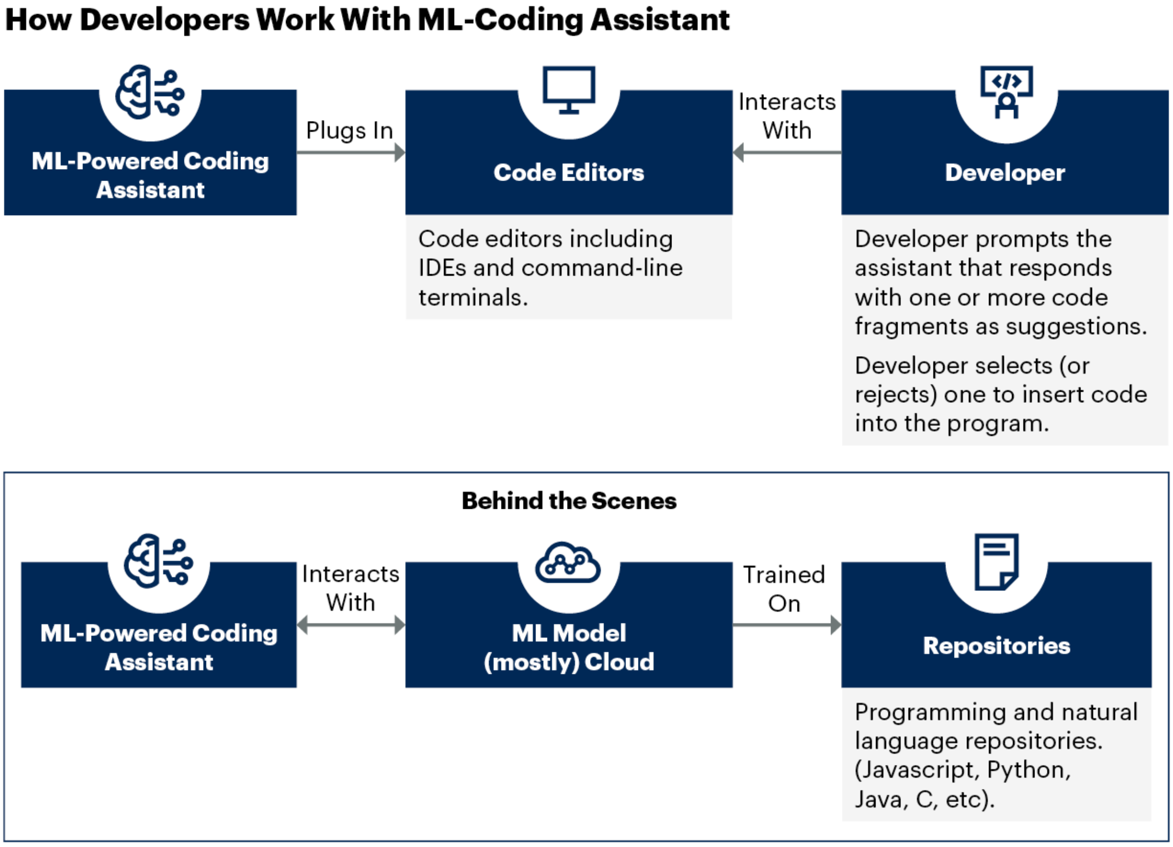
What should a software engineering leader do in the face of this transformative, immature innovation that has both promise and risk?

Software engineering leaders should define a strategy to pilot these powerful tools, and develop a plan to mitigate challenges as they arise.

Description

> **Definition**
>
> An ML-powered coding assistant takes the comments and code surrounding the cursor in a code editor as input prompt. Upon the command of a developer, it then shows its top predictions of new single or multiline code fragments it synthesized, for the developer to accept from.

Figure 2: How Developers Work With ML-Coding Assistant


How Developers Work With ML-Coding Assistant
ML-Powered Coding Assistant
Plugs In
Code Editors
Code editors including IDEs and command-line terminals.
Interacts With
Developer
Developer prompts the assistant that responds with one or more code fragments as suggestions.
Developer selects (or rejects) one to insert code into the program.
Behind the Scenes
ML-Powered Coding Assistant
Interacts With
ML Model (mostly) Cloud
Trained On
Repositories
Programming and natural language repositories. (Javascript, Python, Java, C, etc).

Source: Gartner
777224_C

Developer Experience:

The developer installs the coding assistant as a plug-in to a supported code editor (such as VS Code) or integrated development environment (such as JetBrains or Visual Studio). The coding assistant binds to hotkeys that the developer uses to invoke its assistance. Depending upon the assistant, the developer will use a combination of text and code hints to prompt the assistant to produce valuable code fragments that help them to advance their solution.

Model Training and Inference:

It is important to understand how the ML model is trained and used. This Innovation Insight focuses on deep-learning models that use Generative Pretrained Transformer (GPT) neural-net architecture on a very large corpus of code that contains natural language descriptions such as those embedded in code comments. The size of the training corpus is of the order of millions or billions of lines of code. The training process is expensive, using a large number of advanced computing resources. The resulting model is itself very large, hence most of the models are offered as a managed service, using graphics processing units (GPUs) and application-specific processing units (ASICs) for inference.

Snippet Generation and Acceptance Process:

Once a developer invokes the assistant, the client gathers the surrounding text and sends it to the server running the ML inference service. The service uses this prompt as input and generates a “next best code” recommendation list. Some of the recommendations generated by the inference might not be valid code. The assistant eliminates such invalid code by applying semantic rules, reducing the recommendation list to a few (usually one to five) that it presents to the developer. Some of the tools apply additional filters to eliminate recommended code that contains known biases, explicit content, security vulnerabilities or license infringements. The developer then cycles through the list choosing one or none of the suggestions. The plug-in should do this without adversely affecting developer flow.

Benefits and Uses

ML-powered coding assistants have two main use cases today, accelerated software development and program exploration. [1,2,3,4]

Accelerated Software Development

Developers rely heavily on modern integrated development environments to help them write code that solves problems. One of the key features of these environments is code completion that allows the developer to invoke the engine to show them a list of syntactically valid code fragments that follows the current code token. The conventional code completion engine is a rule-based semantic engine that looks at all possibilities in the current file as well as all other code that is reachable within the scope. These suggestions save an enormous amount of developer time that they would have to otherwise spend to discover and iterate.

Despite their capabilities, current semantic engines are not able to rank and predict which of the possible lines the developer is likely to use, or recommend to the developer based on what other developers have done. This is where ML comes in. ML models trained on millions (and in some cases, billions of lines of code) are able to predict, based on the patterns they have learned, what lines have most frequently been used given the current context; and recommend them to the developer.

This capability, if it works in the long run, has significant implications in improving the productivity of any programmer. Indeed, research shows that these AIs can generate 3% to 40% of the new code, when developers accept these ML completion suggestions. Vendors claim that up to 30% of suggestions are being accepted. The AI plug-in reduces context switches (which break “flow,” reducing developer productivity) because developers don’t have to switch to other applications to search for relevant code to insert. Because of ML helping developers to choose better completions, build-iterate (iteration) times are reduced, thus accelerating software development.

Program Exploration

Even more exciting and transformational than software development acceleration is the use case where a developer interacts with these models in an exploratory, conversational manner. Starting with a vague idea and iteratively shaping a program using a combination of natural language and code in a generative feedback loop. While vendors have demonstrated development of simple games, we should anticipate diverse and more sophisticated program development as the number and sophistication of available models increases along with our ability to design and engineer better prompts.

Code generation beyond the traditional IDE

While the IDE is the primary tool for most application developers, other development environments are also benefiting from these innovations. Jupyter Notebooks are used heavily by data scientists and ML engineers. However only a few ML-powered coding assistants integrate with Jupyter.

The command-line terminal is another place where a lot of glue code is written and executed. Unlike the IDE, very little, if any, innovation has gone into making a terminal smarter, thus limiting the infrastructure and system engineer’s ability to be more productive and innovative. Warp is a company that is bringing ML-enhanced command-line suggestions to terminal users, and making the ubiquitous terminal smart.

With the rise of nonprogrammers who are able to develop using visual paradigms, a new class of platforms under the moniker “low code” have taken root. Some vendors of these products (such as Microsoft Power Apps, Mendix, OutSystems or Quidgest Genio) are looking to generate code or artifacts using AI models.

Benefits

- **Developer Productivity —** Digital products and the employees involved in developing these products — broadly “developers” — become increasingly pivotal for an organization to survive and thrive, it is only natural that we will increasingly care about developers’ productivity. Software development is a difficult skill, and therefore the support capability of AI assistants is welcomed. While debate about what developer productivity actually is and how to measure it rages on, that these tools will increase developer productivity is not in doubt.

- **Developer Satisfaction —** ML-enhanced code generation models seem to increase a developer’s satisfaction in addition to making them highly productive. This is evidenced from our interviews, testimonials shared by vendors, and observing developers’ sentiment on social media. However, we do not know yet whether this greater satisfaction while writing code might be offset by other problems downstream (see Risks section).

Risks

ML-powered coding assistants based on foundational models may propagate inherent challenges: firstly they can fail unexpectedly, secondly, they can harbor bias and thirdly they are opaque and poorly understood. In addition, the code assistants have other specific programming-code-related challenges that need to be addressed:

- **Copyrights and intellectual property —** The power of these models comes from the fact that they have been trained on a very large corpus of source code, but where would you find so much code? Open-source repositories of course. However, open-source software (OSS) is not free of obligations. About 20% of OSS is governed by “copyleft” licensing which restricts reproduction of code, without releasing the containing code to OSS. What if the model synthesizes code fragments that look like code that is governed by such a license? [7]

- **Unsecure code —** These models have been trained using not only high-quality code, but also code with security vulnerabilities and bugs. Therefore, when prompted, they can generate code that has unsecure coding patterns or bugs. Indeed researchers have managed to coax GitHub Copilot to generate code that contained known vulnerabilities. [5]
- **Biased and offensive code and comment generation —** What if the code that the model was trained on was offensive, biased code? Would it then, unintentionally or by malicious manipulation, emit biased code and comments?
- **Privacy of client code —** ML models can be improved by retraining based on the effectiveness of their predictions. In order to do that these models need to send pieces of information from the developer's local environment to the service. However, that action should raise privacy and security concerns.
- **Blind trust —** Vendors have self-reported that developers are accepting approximately one-third of code suggestions, and in some cases 30% of new code is generated by the AI. Generated code should go through the same pipeline as nongenerated code goes through, including human (pull requests for example) and automated tools (such as static application security testing [SAST]) review. [5,6]
- **Maintainability —** Many developers struggle with writing appropriately documented code. A hybrid/AI-assisted cowriting partnership may improve productivity and even accuracy, but if not explained correctly may inadvertently create a maintenance issue because no one fully understands it, not even the original developer.

Adoption Rate

This innovation has penetrated less than 5% of the target population of software developers as of 2022. Given its transformational benefits we expect the adoption to increase rapidly, with 50% of developers adopting it in five years.

Recommendations

Software engineering leaders should responsible for software engineering technologies should:

- Pilot ML-powered coding assistants, with an eye toward fast rollouts to maximize developer productivity. These innovations are evolving rapidly with commercial offerings currently more mature than open source versions. Vendors differ in how developers interact with their products because their products use different models. As a result of these differences, different developers may prefer different products. Make it easy for willing developers to use approved products to maximize their productivity and satisfaction.

- Handpick a few software engineers to master the new skill of “prompt engineering” using a combination of natural language and coding practices to figure out how to optimize code generation with minimal effort. Similar to other generative AI applications such as art, discovering prompt patterns and cataloging them will be highly beneficial.

- Form and lead a cross-functional team to identify, monitor and solve for emerging challenges in copyright infringement, code-quality and nonperformant code generation.

Representative Providers

- Tabnine
- OpenAI (Codex)
- GitHub (Copilot)
- AWS (CodeWhisperer)
- IBM (CodeNet)
- Salesforce (T5, CodeGen)
- FauxPilot
- HuggingFace

Evidence

[1] Productivity Assessment of Neural Code Completion, arxiv.org.

[2] Grounded Copilot: How Programmers interact with Code-Generating Models, arxiv.org.

[3] ML-Enhanced Code Completion Improves Developer Productivity, Google Research.

[4] Evaluating Large Language Models Trained on Code, arxiv.org.

[5] Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions, arxiv.org.

[6] Is GitHub Copilot a substitute for Human Pair-programming? An Empirical Study, IEEE Xplore.

[7] Distribution of Permissive and Copyleft Open-Source Licenses Worldwide From 2012 to 2020, Statista.

Acronym Key and Glossary Terms

LLM	Large language model
GPT, GPT-2, GPT-3	Generative Pretrained Transformer, version 2, version 3

Note1: Evolution of ML-Powered Coding Assistants

In July 2019, a student of computer science at the University of Waterloo, Jacob Jackson released an ML-enhanced autocompleter based on OpenAI's GPT2, called "Deep Tabnine." He trained the model on about 2 million lines of code, picked from OSS. The model became instantly popular with developers, setting social media ablaze with rave reviews. GPT2 is a general purpose large language model and has 1.5 billion hyperparameters and it set the stage for bigger and better models.

In June 2020, OpenAI released an even larger neural-net model called GPT3, an LLM with 175 billion parameters, orders of magnitude larger than GPT2. OpenAI released its private API to selected developers to observe what applications they develop using GPT-3. Developers used GPT-3 to solve programming problems.

In August 2021 OpenAI released "Codex" a multimodal LLM model that they trained on over 10 million repositories of OSS code. Codex takes natural language as input and generates code. Based on Codex, Microsoft GitHub released a coding assistant product called "GitHub Copilot."

In June, 2022 Amazon announced their product called Amazon CodeWhisperer (preview). IBM is also working on developing a similar model based on their project called CodeNet. However, IBM is focused on developing a model that translates a program written in legacy languages to more current ones. Google has developed an internal model that they are testing with about 10,000 engineers. Salesforce AI Research labs is developing ML-enhanced code generation models called CodeGen and T5. FauxPilot is an open-source model developed by Brendan Dolan-Gavitt, assistant professor in the computer science and engineering department at NYU Tandon.

Recommended by the Authors

Some documents may not be available as part of your current Gartner subscription.

Quick Answer: Will Machine-Learning-Generated Code Replace Developers?