**Jakub Wloch**

Index\_no. 64911

**Application of Artificial Intelligence Models to Pull**

**Request Analysis: A Study on Effectiveness in Bug**

**Resolution**

# **Introduction**

Pull request (PR) analysis is a critical part of modern software development, where proposed code changes are reviewed to ensure quality and fix bugs before integration. Automating aspects of this review process has long been a goal in software engineering. Over the years, approaches have evolved from rule-based static analyzers and formal verification techniques to machine learning models and, most recently, large language models (LLMs). This chapter provides a theoretical overview of these developments. We begin with a historical perspective on automated code analysis in the pre-LLM era, covering the mathematical foundations and early tools that set the stage for today’s AI-driven methods. This includes the evolution of static analysis, model checking, program analysis algorithms, formal methods, and early machine learning applications for bug detection. We then transition to the current state of the field, marked by the rise of LLMs that can analyze and even generate code. We examine how models like OpenAI’s GPT-4, Anthropic’s Claude, Google’s Gemini, and Meta’s LLaMA (and its derivatives) are being applied to automated code review and bug resolution tasks. Their performance on emerging benchmarks (e.g., SWE-bench, BigCodeBench) is discussed, along with comparative insights into their capabilities. We also discuss the architectural innovations that enable these models – from the Transformer architecture to fine-tuning and retrieval augmentation – as well as current practices in benchmarking their performance. Finally, we will address persistent challenges such as understanding complex code context, providing explainable outputs, and building developer trust in AI-generated code suggestions.

# **Historical Foundations of Automated Code Analysis (Pre-LLM Era)**

### Mathematical Foundations and Early Formal Methods

The roots of automated code analysis lie in foundational theoretical work in computer science and mathematical logic. A fundamental challenge is that many properties of programs are *undecidable* in the general case. Alan Turing’s groundbreaking 1936 proof demonstrated the **halting problem**, establishing that no algorithm can decide in all cases whether a program will terminate or run forever[1](https://queue.acm.org/detail.cfm?id=3487021#:~:text=There%20are%20some%20fundamental%20limits,An). This insight implied that any automatic analysis of programs must deal with *incompleteness* or *approximation*. In practice, it meant that tools for program analysis could not guarantee finding all bugs (risking false negatives) without potentially flagging correct code as problematic (false positives), and vice versa. Early formal work by logicians and computer scientists in the 1960s and 1970s tackled this balance. For example, Floyd and Hoare developed **Hoare logic** (1969) to reason about program correctness via pre- and post-conditions, and Dijkstra introduced the idea of *weakest preconditions* for reasoning about program statements. These works provided a manual proof-oriented approach to verify programs, laying a foundation for later automated reasoning.

By the late 1970s, the field of *formal methods* had introduced systematic ways to specify and verify program behavior using mathematics. One landmark was the theory of **abstract interpretation** by Patrick and Radhia Cousot (1977). Abstract interpretation provided a unified lattice-theoretic framework for static program analysis, formalizing how to derive a sound *approximation* of a program’s behavior[2](https://queue.acm.org/detail.cfm?id=3487021#:~:text=Concurrently%2C%20static%20analysis%20received%20attention,the%20only%20viable%20path%20forward). The Cousots’ insight was that since exact program analysis is impossible in general (due to the halting problem and related undecidability results), one can design analyzers that operate on an abstract domain – a mathematical abstraction of program properties – and compute fixpoints to approximate program semantics. This approach guaranteed *soundness* (no false negatives for properties being checked) at the cost of *completeness*, thereby systematically managing the trade-off between catching bugs and avoiding false alarms. Abstract interpretation became a cornerstone for many later static analysis tools and continues to influence analysis design.

Another key development in this era was **model checking**, introduced in the early 1980s. Pioneering work by Clarke and Emerson (1981), and independently by Quielle and Sifakis, showed that one could automatically verify finite-state models of software or hardware against formal specifications expressed in temporal logic[3](https://en.wikipedia.org/wiki/Model_checking#:~:text=specification%20was%20done%20by%20Amir,9). In model checking, the program (or system) is modeled as a state transition graph, and an algorithm exhaustively explores all states to check if a temporal logic property (such as a safety or liveness condition) holds. This was revolutionary for verifying concurrent systems and hardware circuits. Tools like **SPIN** (for verifying communication protocols using linear temporal logic) and **SMV** (Symbolic Model Verifier) emerged, allowing practitioners to find subtle bugs like race conditions or deadlocks by systematic state-space exploration. Model checking brought a level of automation to formal verification that theorem-proving lacked, and its success in hardware verification (and subsequent Turing Award recognition for Clarke, Emerson, and Sifakis in 2007) underscored its importance[4](https://en.wikipedia.org/wiki/Model_checking#:~:text=checking%20began%20with%20the%20pioneering,9). However, model checking’s use in software was limited by state-space explosion and undecidability issues for arbitrary programs[5](https://en.wikipedia.org/wiki/Model_checking#:~:text=Model%20checking%20is%20most%20often,interpreted%20%2087.%5B%2011) – typically it had to be applied to abstracted models of software or to specific classes of software (like device drivers or synchronization protocols) where state could be reasonably bounded.

In summary, the pre-1980s period established the theoretical bedrock for code analysis: it became clear that automation required embracing *mathematical rigor* (to reason about programs) as well as *incompleteness* (accepting that analyses must approximate). **Formal methods** such as Hoare logic and temporal logic provided languages to specify what it means for code to be “correct.” **Theorem provers** and proof assistants (like the early Boyer-Moore theorem prover, and later Coq, Isabelle, etc.) began to appear, offering a way to mechanically check proofs of program correctness – albeit with significant manual effort. These tools were powerful but demanded specialized expertise and were usually applied to critical software components due to the labor-intensive process.

### Static Analysis Tools and Techniques

Parallel to formal verification, the late 1970s and 1980s saw the rise of practical **static analysis** tools aimed at automatically finding bugs or suspicious constructs *without executing code*. Static analysis examines source code (or compiled bytecode) to infer properties – for example, detecting possible null dereferences, type inconsistencies, or security vulnerabilities. A classic early example is the tool **lint**, developed by Stephen Johnson in 1978 for the C programming language. Lint was distributed with Unix in 1979 and was designed to flag likely errors or risky constructs in C programs[6](https://queue.acm.org/detail.cfm?id=3487021#:~:text=Today%27s%20notion%20of%20static,are%20often%20part%20of%20C). It could catch issues that the compilers of the time did not, such as unused variables, type mismatches, or questionable type conversions. The success of lint was significant: it popularized the idea that a separate tool could analyze code for quality issues. In fact, lint was so influential that its name became a verb; today *“linters”* exist for many languages to denote static checkers that enforce coding standards or catch simple bugs. One downside of early lint was its propensity for false positives – it might warn about code that was actually correct but unusual. Developers had to explicitly annotate code to suppress such warnings[7](https://queue.acm.org/detail.cfm?id=3487021#:~:text=While%20,linters%E2%80%9A%20across%20many%20programming%20languages). This foreshadowed a general challenge in static analysis: tools must be precise enough to be useful, but not so pedantic that developers start to ignore them due to noise.

Throughout the 1980s and 1990s, static analysis techniques grew more sophisticated. Compilers themselves incorporated many static analysis *passes* for optimization (e.g., data-flow analyses to propagate constants or eliminate dead code), and these techniques were adapted to bug-finding. For instance, **data-flow analysis** and **control-flow graph** construction became standard underpinnings of static analyzers. Researchers also developed specialized analyses: **type systems** were extended (e.g., the notion of “typestate” to track resource usage protocols), **pointer analysis** for C/C++ to reason about aliasing, **symbolic execution** to follow logical paths in programs, and **taint analysis** to track untrusted data (critical for finding security vulnerabilities). Much of this work built on the theory of abstract interpretation – by choosing an abstract domain (like intervals for variable ranges, or a lattice of sets of possible values) and computing fixpoint solutions, static analyzers could prove properties such as “variable X is always non-null at this program point” or “no execution can exceed array bounds in this loop.” Academic tools like **ASTREE** (which used abstract interpretation to prove absence of runtime errors in fly-by-wire avionics software) demonstrated that, in certain domains, static analysis could be push-button and extremely precise – in ASTREE’s case, it was able to analyze safety-critical C code with no false alarms by carefully tuning abstract domains. In industry, companies like **Coverity** and **Fortify** (both founded in the 2000s) built static analysis engines that could analyze large codebases (millions of lines) for bugs. Coverity’s tool, for example, was rooted in research by Engler et al. at Stanford, which had shown that many OS bugs could be found by looking for *deviations from common patterns* in code. Engler’s 2001 paper “Bugs as Deviant Behavior” introduced a static analysis that inferred rules from code (like “if you lock a mutex, you must eventually unlock it”) and then flagged deviations, finding hundreds of bugs in Linux and OpenBSD code[8](https://web.stanford.edu/~engler/deviant-sosp-01.pdf#:~:text=ten%20to%20one%20hundred%20more,5%20give). This demonstrated the power of automated analysis on real-world, complex software: *hundreds of errors* were found and many resulted in patches to the operating system kernels[9](https://web.stanford.edu/~engler/deviant-sosp-01.pdf#:~:text=ten%20to%20one%20hundred%20more,5%20give).

By the late 1990s, **static analysis** had become an integral part of software quality assurance. Tools were available to check for memory leaks, concurrency issues (like potential data races or deadlocks), and adherence to coding standards (e.g., the MISRA rules for C in safety-critical systems). Notably, the *false positive problem* remained a challenge: overly noisy tools would be ignored by developers. Research and practice therefore focused on improving precision (using better heuristics or program models) and on **suppression mechanisms** (so developers could silence known-benign warnings). Some tools opted to be unsound (i.e. *incomplete*, potentially missing some real bugs) in exchange for higher precision, depending on the use-case. A balance was often struck: for example, Microsoft's **Visual Studio** static analyzer and the **Java** - *javac* compiler’s warnings both incorporated many such trade-offs to be useful out-of-the-box. Static analysis also broadened beyond bug-finding: for instance, **code metrics** (like complexity measures) and **style enforcers** can be seen as static analyses that help maintain code quality.

### Model Checking and Program Verification in Practice

While static analysis works directly on source code, **model checking** (as introduced above) and related *formal verification* techniques took a different approach: verify an abstract *model* or *specification* of the program. By the 1990s, model checking had notable success in hardware design verification and protocol verification. For software, researchers created tools to apply model checking to source code by extracting models. One example was Microsoft Research’s **SLAM project** (early 2000s), which automatically extracted Boolean abstractions from C code (specifically Windows device drivers) and used model checking to find API usage errors. SLAM targeted bugs like improper use of Windows kernel APIs by drivers and was later commercialized as the “Static Driver Verifier” in the Windows development kit. This showed that model checking could be made practical for certain classes of software bugs by cleverly abstracting the problem[10](https://web.stanford.edu/~engler/deviant-sosp-01.pdf#:~:text=from%20missing%20features%20and%20over,heavyweight%20check%02ers%2C%20notably%20the%20extended). Another avenue was **run-time model checking** or systematic testing (e.g., tools like Microsoft’s CHESS for systematically exploring thread interleavings in concurrent programs). In academia, tools like **Java PathFinder** combined model checking with symbolic execution to systematically test Java programs. Alongside model checking, **theorem proving** continued to advance – the 1990s and 2000s saw the first fully machine-checked proofs of certain algorithms and systems (e.g., a milestone was the CompCert C compiler, fully verified in Coq to have no miscompilation bugs). However, these proofs were labor-intensive efforts by experts.

In practice, formal methods in the pre-LLM era were predominantly applied in high-assurance domains – such as aerospace, medical devices, and cryptography – where the cost was justified by the need for near-perfect reliability. For mainstream software, formal verification remained the exception rather than the rule, often due to the high effort and specialized knowledge required. As Engler et al. observed, while formal verification can find very deep bugs, it is “so difficult and costly that it is rarely used for software” in general[11](https://web.stanford.edu/~engler/deviant-sosp-01.pdf#:~:text=extreme%20example%20of%20this%20approach,While%20recent%20work). Thus, the typical software project relied more on testing and static analysis tools than full formal verification. Nonetheless, the contributions of this period were crucial: they provided **soundness guarantees** and a rich theory that ensured automated analysis tools had a firm grounding. Concepts like invariants, pre/post-conditions, state space exploration, and abstract domains became part of the toolkit for building automated code analyzers.

### Early Machine Learning Applications in Code Analysis

By the 2000s, researchers began to explore **machine learning (ML)** and statistical techniques to assist in code analysis and bug prediction. One of the earliest fruitful areas was **defect prediction**: using metrics of code (such as complexity, churn, past bug history) to train classifiers that predict which modules or files are likely to have bugs in the future. By training on historical project data, these models could guide testers and reviewers to areas of code that deserved more scrutiny. Studies introduced increasingly sophisticated evaluation metrics (precision, recall, F1-score, AUC) instead of just accuracy, to account for the imbalance (typically, bugs are relatively rare). They also emphasized the importance of validation methodology – for example, ensuring that a model generalizes to new projects (cross-project validation) versus just learning characteristics of one project. While defect prediction did not pinpoint the bugs themselves, it leveraged early ML (like decision trees, random forests, or neural nets) to prioritize where human effort should be spent.

Another area was **code mining** for patterns. The idea that code has repeating patterns was quantified by Hindle et al.’s finding in 2012 that code is *natural* in the sense of being highly predictable by statistical language models. N-gram models over source code could predict the next token with surprising accuracy, implying a form of regularity in how programmers write code. This “naturalness of software”[12](https://www.researchgate.net/publication/254041552_On_the_naturalness_of_software#:~:text=On%20the%20naturalness%20of%20software,gram%20models)[13](https://cacm.acm.org/research/on-the-naturalness-of-software/) spurred work on applying language modeling to code for tasks like code completion and anomaly detection. For instance, an unusual code sequence that a statistical model deems very improbable might indicate a bug (this was the intuition behind tools like **DeepBugs** (Pradel & Sen, 2018), which learned to detect swapped function arguments or incorrect API usages by recognizing when code violates common patterns).

By the mid-2010s, the rise of deep learning brought more advanced applications. Techniques like **sequence-to-sequence neural networks** were applied to suggest bug fixes: for example, the *DeepFix* system (Gupta et al., 2017) used an LSTM-based model to fix common syntax and semantic errors in student C programs. It treated program repair as a translation problem – from “buggy code” to “repaired code” – and was able to fix a significant fraction of errors like missing semicolons, type mismatches, etc., without human intervention. Similarly, **program repair** research produced neural approaches such as SequenceR (2019) and COCONUT (2020), which trained on past bug-fix commits to learn how to suggest patch code for new bugs. These early ML models were limited in scope (often focusing on small programs or specific bug types), but they demonstrated the potential of data-driven learning in automating bug fixes.

During the 2010s, a synergy developed between traditional program analysis and machine learning. Static analysis tools started incorporating probabilistic reasoning – for example, learning which warnings were likely true positives based on past code patterns. Conversely, ML models sometimes borrowed program analysis ideas to better understand code structure (like augmenting neural nets with abstract syntax tree information or doing graph-based neural networks over programs). A notable trend was *learning from large code repositories*: GitHub and other sources provided a wealth of example code and associated bug fixes (commit histories, Q&A explanations on StackOverflow, etc.) that could be used to train models. Research in **mining software repositories** found patterns such as “buggy code often involves usage X, and the fix often changes it to Y,” which could be encoded as features for ML models.

Benchmarking approaches in this era were domain-specific. Static analyzers and formal tools were often evaluated on synthetic or collected bug datasets – for example, the **Toyota ITC Benchmark** or NSA’s **Juliet Test Suite** for security bug finding, and the **SV-COMP** competition for software verification tools which provided a set of verification tasks every year. For data-driven bug prediction or repair techniques, researchers created collections of known buggy code and fixes: e.g., **Defects4J** (a curated set of real bugs from Java projects) became a standard benchmark for program repair algorithms; the **ManyBugs** and **IntroClass** benchmarks provided C programs with seeded or student-written bugs. Evaluation metrics included how many bugs were fixed (for repair), precision/recall of bug identification (for prediction), or correctness of suggested fixes (often validated by running test cases). These pre-LLM benchmarks were generally narrower in scope than what we see today – they focused on specific repositories or specific types of tasks. Nonetheless, they established a baseline for comparing automated debugging techniques and underscored the need for rigor in evaluation (such as avoiding training on the test data, handling bias, etc.).

## The Rise of Large Language Models in Code Analysis

### Transformers and the Emergence of Code-Savvy LLMs

A pivotal change in AI came with the introduction of the **Transformer** architecture by Vaswani et al. in 2017[14.](https://arxiv.org/abs/1706.03762#:~:text=,French) The Transformer’s use of self-attention mechanisms enabled training of much larger models on vast datasets by leveraging parallelism and long-range context handling. Soon after, NLP saw language models scale from millions to billions of parameters, leading to breakthroughs in tasks like translation and question answering. These same techniques were applied to programming languages, treating code as a form of language. Companies and research groups realized that by training transformers on large corpora of code (and related natural language like documentation), one could obtain models that generate and analyze code with surprising competence.

Early successes included models like **GPT-2** (2019) and **GPT-3** (2020) which, despite being trained mostly on natural language, showed some ability to generate syntactically correct code or complete simple programming tasks due to code data present in their training corpus. The real watershed moment came with OpenAI’s **Codex** (2021), a transformer-based model explicitly fine-tuned on billions of lines of source code from public repositories[15](https://shmulc.medium.com/humaneval-the-most-inhuman-benchmark-for-llm-code-generation-0386826cd334). In evaluations, Codex could produce correct solutions to coding problems given in natural language about 28% of the time on a new benchmark called HumanEval[16](https://shmulc.medium.com/humaneval-the-most-inhuman-benchmark-for-llm-code-generation-0386826cd334) – a significant jump over previous code generation systems. Codex was deployed as the engine behind GitHub **Copilot**, a developer assistant that could autocomplete code and suggest entire functions. This marked the first widespread use of LLMs in daily programming: developers reported that Copilot could implement simple functions, suggest bug fixes, and even catch errors by producing alternative snippets. It was not perfect – it could introduce mistakes or insecure code – but it demonstrated the practicality of LLMs in augmenting human coders.

Several factors contributed to the success of code LLMs: (1) **Scale of training data** – these models were trained on massive datasets (GitHub alone has millions of repositories, encoding a huge variety of algorithms and solutions). (2) **Transfer learning** – the models learned general language understanding from text and applied it to code (and vice versa, e.g., understanding code comments). (3) **Transformer architectures** excel at capturing long-range dependencies, which is crucial for code, where a variable definition and its usage may be far apart, or where understanding the structure (indentation, braces matching) requires global context. (4) **Fine-tuning with human feedback** – OpenAI and others later applied techniques like reinforcement learning from human feedback (RLHF) to make the code output more aligned with user intentions (e.g., preferring correct and readable solutions). By 2022, deepmind’s **AlphaCode** also demonstrated the ability of LLMs to solve competitive programming problems; AlphaCode generated a large number of candidate programs for each problem and then cleverly filtered and tested them, ultimately achieving roughly median human competitor performance in coding contests – a remarkable feat for an AI with no explicit formal reasoning, just learned patterns.

### Automated Code Review and Bug Detection with LLMs

As LLMs proved adept at generating code, attention turned to using them for **code analysis** tasks such as reviewing PRs, detecting bugs, and suggesting fixes – essentially leveraging their knowledge to *understand* code, not just write it. These models are given prompts that include code (or a code diff) and a request like “find potential bugs” or “explain what this code does” or “suggest improvements.” The LLM then produces an analysis or critique, drawing on both its programming knowledge and general reasoning abilities.

One advantage LLMs have is flexibility: a single model can be asked diverse questions about code – from stylistic improvements to security vulnerabilities – whereas traditional static tools are usually specialized. For instance, developers have used GPT-4 to act as an AI code reviewer: given a GitHub pull request description and the code changes, GPT-4 can point out issues such as logical errors, violations of coding guidelines, or missing edge-case handling. Empirical reports show that GPT-4 can catch certain categories of bugs that static analyzers might miss (for example, if the bug relates to higher-level logic or an unconventional use of an API), and it can explain its reasoning in natural language, which is a big plus for developer understanding. One study integrated GPT-4 into the output of a static analysis tool (Semgrep) to help triage and fix findings. The results were promising: GPT-4 could often determine whether a static analysis warning was a true problem or a false positive, and even propose a correct code change to fix the issue[17](https://semgrep.dev/blog/2023/gpt4-and-semgrep-detailed/#:~:text=Semgrep%20is%20a%20code%20search,its%20output%20is%20often%20correct). This kind of *AI-assisted code review* augments rule-based analyzers with the LLM’s broader context understanding, potentially reducing noise by filtering out unimportant issues and providing human-like explanations.

LLMs have also been used in **bug localization** – i.e., finding where in the code a bug might be given a description or an error message. Because LLMs have seen many code patterns and common bug fixes during training, they can sometimes guess that “if the error is X, it often occurs in patterns like Y,” and they direct attention to that part of code. Moreover, for **automated program repair**, researchers have harnessed LLMs by feeding in a buggy code and the failing test (or bug report), and asking the model to produce a patch. Compared to earlier neural repair models, today’s LLMs are orders of magnitude more powerful and have a far larger knowledge base of fixes. For example, on the benchmark of GitHub issues and fixes called **SWE-bench** (Software Engineering benchmark), initial experiments showed that GPT-4 based agents could indeed solve a subset of real-world bug fixes or feature requests automatically[18](https://arxiv.org/html/2410.06992v1#:~:text=et%C2%A0al,Verified%20went%20up%20to%2045). In SWE-bench, each task provides an issue description and test cases, and the AI must produce the code changes (a diff) that resolve the issue. When SWE-bench was first introduced (Jimenez et al., 2024), the success rates were extremely low – on the order of a fraction of a percent for baseline models – underscoring the difficulty of the task (these are complex, multi-file, real issues). But within a year, rapid progress was made: the best research systems (combinations of claude 3.7 with clever prompting or fine-tuning) achieved up to about **34% resolution rate** on the full dataset and as high as 45% on a focused subset of bug-fix issues[19](https://www.swebench.com/#:~:text=Site-,%E2%9C%85,%E2%9C%93,-%E2%9C%85). This improvement from virtually 0% to double-digit percent showcases the fast pace of advancement in applying LLMs to realistic software problems. Techniques that contributed to this jump included retrieval of relevant context (e.g., bringing in related code or documentation for the model to read), step-by-step reasoning prompts, and using feedback from tests to refine the output (a strategy akin to automated debugging).

It is important to note that measuring LLM performance on code tasks has itself become a nuanced effort. Early metrics like “pass@1” (did the model produce a correct solution in one try) or pass@k (in k tries) from the Codex evaluations are now supplemented with more comprehensive benchmarks. The **HumanEval** benchmark introduced with Codex remains a staple for function-level code generation; models have gone from 28% for Codex to around 90-99% for the latest LLaMA3 on HumanEval (with LLaMA solving most of those problems)[20](https://paperswithcode.com/sota/code-generation-on-humaneval). But newer benchmarks test more complex scenarios. For example, **APPS** (a dataset of competitive programming problems) and **CodeContests** push models to write longer, more complex programs. Models like GPT-4 have achieved roughly 35%+ on easy APPS problems but much lower on hard ones, indicating room for growth in algorithmic reasoning. Another example, **MBPP (Mostly Basic Programming Problems)** from Google, is a set of 500 simple coding tasks – modern models also score high on these (01Preview had ~89% pass@1 on MBPP, Code Llama reached ~DeepSeekV3 roughly 87%[21](https://evalplus.github.io/leaderboard.html). These results suggest that for self-contained coding tasks with clear problem statements, LLMs have become quite effective.

However, performing *contextual code analysis* (like PR reviews) can be more challenging because the model must comprehend an arbitrary codebase and change. It’s one thing for an AI to write a quicksort function from scratch, and another to understand a 5,000-line codebase and the one-line change proposed and judge its correctness or impact. LLMs are increasingly used in such scenarios through careful prompting. They may be given a summary of the project (or sometimes allowed to browse the repository with tools), and then the diff, and asked questions like “Could this change have unintended side effects?” or “Does it properly handle all cases? If not, point out issues.” Anecdotal evidence from developer forums and blogs suggests GPT-4 can identify subtle issues, for example, detecting that a code change does not handle a null input in some rarely-used function or that it might degrade performance by introducing an extra loop. In effect, the LLM can serve as a tireless pair programmer reviewing code 24/7.

A concrete example highlighting LLM capabilities in code review is a Medium case study titled “What If GPT Was Your Code Reviewer?” where a developer fed GPT-4 a series of real pull requests[22](https://graphite.dev/blog/problems-with-ai-code-review). The AI flagged 70–80% of the “nitpick” comments that a human reviewer would make (coding style, minor improvements) and even caught some functional problems. Nonetheless, it also missed some issues or made suggestions that were off-mark, indicating it’s not infallible. But as a proof-of-concept, it showed that *a large fraction of routine code review feedback can be generated automatically*, which could free human reviewers to focus on more complex design concerns. Companies are actively researching this integration – for instance, Microsoft’s Visual Studio team has explored GPT-based “Copilot for Pull Requests” which automatically suggests review comments.

### Performance on Benchmarks and Limitations of Current LLMs

To objectively assess these AI models, new benchmarks have been developed that mirror real-world software engineering tasks. We already introduced **SWE-bench**, which is one such benchmark focusing on issue-driven code modifications. SWE-bench is challenging because it requires understanding a natural language issue description, reading and modifying code in possibly multiple files, and ensuring all tests pass – essentially, it tests an AI’s ability to act as a developer resolving a ticket. As noted, even Claude 3.7Sonnet (which is considered the best at solving coding challenges) combined with an automated agent (SWE-Agent) managed to resolve only about 34% of issues (and ~57% on a simpler bug-fix subset, using claude 4.0Sonnet)[23](https://www.swebench.com/). Moreover, a deeper analysis found many of those “resolved” cases were due to *data leakage* (the solution was sometimes written in a discussion thread or the tests were weak).

Another recent benchmark is **BigCodeBench** (Zhuo et al., 2024), which evaluates code generation in tasks requiring **multiple function calls and complex instructions** spanning various libraries[24](https://arxiv.org/abs/2406.15877#:~:text=to%20invoke%20multiple%20function%20calls,further%20advancements%20in%20this%20area). Unlike HumanEval’s short, self-contained functions, BigCodeBench tasks might say (in plain English) “Read a CSV file from a URL, filter entries by some criteria, then plot the results in a chart,” which requires the model to compose multiple API calls (e.g., HTTP requests, CSV parsing, data filtering, plotting) possibly using different libraries. An extensive evaluation of 60 models on BigCodeBench showed that current LLMs struggle with this level of complexity: the best models achieved success rates around **60%**, whereas human performance was ~97%[25](https://arxiv.org/abs/2406.15877#:~:text=automatically%20transforms%20the%20original%20docstrings,further%20advancements%20in%20this%20area). In other words, on more open-ended, realistic coding tasks, even our strongest LLMs (GPT-4 being among them) fall significantly short of expert developers. The errors often come from the model either failing to plan a correct sequence of operations or making API misuse mistakes that a human would easily avoid. This highlights a limitation: present-day LLMs, while excellent at generating plausible code, can miss the mark on *comprehensive understanding and integration* of multiple steps or tools needed in a program.

There are also specialized benchmarks like **Codeforces** problems (competitive programming), **LeetCode** challenge sets, **security vulnerability benchmarks** (like generating or detecting vulnerable code), etc. On competitive programming, GPT-4 has shown strong performance on easy and medium problems (sometimes even exceeding average contestants), but on the hardest problems it still often fails, likely due to needing complex problem decomposition or advanced algorithms that were scarce in training data. On vulnerability detection, some studies have shown that GPT-4 can identify certain vulnerabilities in code (like SQL injection or buffer overflows) with reasonable accuracy if prompted well, but it might miss others or produce false alarms, especially if the vulnerability hinges on subtle domain knowledge (e.g., misuse of a cryptographic API).

A significant limitation of using LLMs for code analysis is **reliability and consistency**. These models can sometimes produce *confident-sounding but incorrect* analyses – a phenomenon often called “hallucination” in the context of generative models. For example, an LLM might incorrectly flag a piece of code as buggy because it “thinks” a certain function call is dangerous, when in context it’s actually fine. Without an explicit knowledge base or runtime checking, the LLM has no guarantee its assertion is correct; it’s drawing on learned associations which could be outdated or not perfectly applicable. This is problematic in a code review setting: you wouldn’t want an AI to insist a correct piece of code is wrong, causing confusion for developers. Conversely, the AI might also *miss* certain bugs because it hasn’t seen a similar pattern before or because the bug involves a complex interaction that wasn’t represented in training. For instance, issues involving concurrency or subtle memory ownership (where dynamic reasoning about multiple pieces of code is needed) can be missed by a straightforward LLM prompt that only looks at a diff or a single file.

Another limitation is **context handling**. Large codebases might span thousands of files, and understanding a PR could require reading and understanding many of them. Out-of-the-box, most LLMs have context window limits (the amount of text/code they can consider at once). Early GPT-3 had a limit of 2048 tokens (roughly 3,000 words), GPT-4 extended this to 8k or 32k tokens in some versions, and Anthropic’s Claude boasted up to 100k tokens context window[26](https://x.com/AnthropicAI/status/1656700154190389248) – about 75,000 words, which might cover an entire repository’s relevant portions. Indeed, Anthropic’s Claude model has been highlighted for its ability to take in **whole files or even multiple files** of code and provide analysis, precisely due to such a large context window. This is a notable innovation: by fitting, say, 100 files of code into the prompt, the model can potentially cross-reference across them to catch issues like a function’s contract being violated in another module. However, even 100k tokens may not cover some huge projects, and feeding that much context can incur high computational cost and sometimes confuses the model if too much irrelevant information is included. Thus, intelligent *context selection* (via vector databases or search within the code) is often used so that only the likely relevant code is given to the model, a technique known as Retrieval-Augmented Generation (RAG). In summary, context handling is much improved with modern LLMs, but not a completely solved issue – very large or interdependent code changes can still pose challenges.

Finally, while LLMs are *generalists* and can be applied to virtually any programming language or task, they might lack *specialized knowledge* of certain frameworks or domains if those were underrepresented in training. For example, a model might be great at Python or JavaScript (which are abundant in training data) but weaker in niche languages or proprietary frameworks. Fine-tuning on specific technical domains can help, but that reintroduces the need to gather domain-specific data and maintain separate model variants.

### Recent Advances and Comparative Insights

The fast-moving landscape of AI for code has seen a flurry of model releases and improvements. Here we highlight a few notable ones and compare their characteristics:

* **OpenAI GPT-4 (2023)** – As of its release, GPT-4 has been one of the most powerful closed-source LLMs, with demonstrated prowess in reasoning and code understanding. It improved significantly over its predecessor (GPT-3.5/Codex) in code tasks, as evidenced by much higher success on benchmarks and its ability to handle more complex multi-step problems. GPT-4’s architecture is not publicly described in detail, but it is known to be a Transformer-based model with *hundreds of billions* of parameters, and it underwent extensive fine-tuning, including RLHF. In coding, GPT-4 can often not only generate code but also explain it. For instance, given a piece of code, GPT-4 can summarize what it does or identify potential bugs or pitfalls, often with a rationale. It still has limits – e.g., it might not perfectly adhere to a project’s specific style, and it can occasionally produce subtly wrong code that looks right – but it represents the closest thing yet to an “AI software engineer” in terms of versatility. GPT-4’s multi-modal version (able to accept images) hints at future possibilities where an AI could, say, read a screenshot of an error and the code and connect the dots.
* **Anthropic Claude (2023)** – Claude is another large model, developed by Anthropic, with a focus on *harmlessness and helpfulness*. Technically, Claude is also a Transformer-based LLM in the same league as GPT-3.5/4, and Anthropic has iterated on versions (Claude 1, Claude 2, etc.). A distinguishing feature of Claude is its very large context window – the Claude 2 model introduced a context window of up to 100k tokens. This makes Claude particularly attractive for code analysis tasks: one can feed entire files or even entire codebases for it to analyze. For example, Claude could take in all the code related to a PR (even if it’s spread across many files) and the PR diff, and then provide a review. Users have reported Claude performing well at tasks like summarizing a codebase’s architecture or doing impact analysis of a change across a project. In terms of raw coding ability (like solving coding problems), Claude is roughly on par with GPT-3.5-tier models, a bit behind GPT-4 in many evaluations. But its edge in context length sometimes lets it outperform GPT-4 on tasks where understanding the broader context is key (GPT-4’s standard version might not fit everything relevant in its 8k/32k window). Anthropic has also emphasized *interpretability* research – e.g., they’ve published work on understanding the inner mechanisms of LLMs – which could eventually yield models that are more transparent in their reasoning, a boon for explainability (a point we’ll return to).
* **Google DeepMind Gemini (2024)** – Announced as Google’s answer to GPT-4, **Gemini** is a family of LLMs developed by the unified Google Brain/DeepMind team. Gemini is designed to be *multimodal* and incorporate techniques from DeepMind’s reinforcement learning successes (such as AlphaGo) into the language model domain[27](https://en.wikipedia.org/wiki/Gemini_(language_model)#:~:text=DeepMind%27s%20AlphaGo%20%20program%2C%20which,5). According to Google’s CEO and DeepMind’s CEO Demis Hassabis, the vision for Gemini is to *combine the power of strategic planning (à la AlphaGo) with the language abilities of LLMs*[28](https://en.wikipedia.org/wiki/Gemini_(language_model)#:~:text=DeepMind%27s%20AlphaGo%20%20program%2C%20which,5). In practical terms, this could mean Gemini is better at multi-step reasoning or tool use, which are crucial for complex coding tasks (like figuring out a sequence of API calls or performing calculations during code execution). By late 2023, Google had several sizes of Gemini (Nano, Pro, Ultra etc., corresponding to different parameter counts). Early reports and tests (some possibly leaked or limited-access) suggested that **Gemini Ultra** was highly capable, potentially surpassing GPT-4 on certain internal benchmarks, while **Gemini Pro** (the version integrated into Bard for general use) was competitive but perhaps not drastically beyond GPT-4[29](https://blog.bajratechnologies.com/gpt-4o-vs-gpt-4-vs-gemini-1-5-a-head-to-head-comparison-2024-987276e80da5). On code tasks, a significant competitive point is how well Gemini can integrate with Google’s ecosystem – for instance, one could imagine it working with Android Studio or Google Cloud to help write code. Another differentiator is multi-modality: Gemini was trained on not just text but also images, possibly audio, and code. While images might not seem directly related to coding, think of UI design – an AI that sees an image of a GUI and then generates code for it (or vice versa). Or in debugging, a model that can read a stack trace image. Gemini’s training also reportedly included YouTube transcripts and other data, giving it potentially a broad range of problem-solving knowledge (e.g., transcripts of programming tutorials). If Gemini realizes Hassabis’s vision, it might be more agentic – meaning it could plan out actions like “edit this file, run tests, gather output, then decide next step,” blurring the line between a static model and a dynamic AI agent performing coding tasks. As of early 2025, comparisons between GPT-4 and Gemini are still playing out; some tests show GPT-4 slightly ahead in strict coding accuracy, others show Gemini catching up or excelling in integrated tasks. Regardless, the emergence of Gemini underscores that multiple tech giants are pushing the frontier, and competition is driving rapid improvements.
* **Meta’s LLaMA and Code Llama (2023)** – Meta took a different approach by releasing their LLaMA models openly to researchers (and later commercially with LLaMA 2), sparking a wave of innovation in the open-source community. **LLaMA** models are foundational LLMs trained on a mix of text (including some code) and are known for strong performance relative to their size. Building on LLaMA 2, Meta released **Code Llama** in late 2023, which is a family of models (7B, 13B, 34B, 70B parameters) further fine-tuned on code and code-related instructions[30](https://arxiv.org/abs/2308.12950#:~:text=Llama%20,We%20release%20Code%20Llama%20under). Code Llama and its variants (Python-specialized, instruction-tuned) achieved state-of-the-art performance among open models on code benchmarks upon release – for example, the 34B and 70B versions reached around **67% pass@1 on HumanEval** (Python)[31](https://arxiv.org/abs/2308.12950#:~:text=Llama%20,We%20release%20Code%20Llama%20under), rivaling or surpassing many closed models from earlier in 2023. Code Llama models also support long contexts (up to 100k tokens in some cases, similar to Claude’s capability)[32](https://arxiv.org/abs/2308.12950#:~:text=Llama%20,We%20release%20Code%20Llama%20under) and even offer an *infilling* ability (filling in the middle of code), which is useful for certain IDE features. The open-source availability of these models means developers can self-host AI assistants and fine-tune them on their own codebases, addressing privacy and customization concerns. Companies like Replit fine-tuned the Code Llama base to create **Replit-CodeInstruct** and others specifically geared to their platform. There have also been community-driven fine-tunes such as **Phind-CodeLlama** that further boost coding performance (one such model reportedly hit ~74% on HumanEval[33](https://www.phind.com/blog/code-llama-beats-gpt4)). While these open models typically still underperform GPT-4 on the hardest tasks, the gap has been closing. Moreover, open models can be extended with plugins and memory – for instance, one can equip an open LLM with the ability to call a compiler or run tests on the fly, creating a feedback loop without being constrained by OpenAI or others’ API policies. This has given rise to a plethora of **AI coding assistants** that run locally or on a cloud VM with no external API: examples include *CodeGeeX*, *SantaCoder/StarCoder* (from BigCode project), and many variants of LLaMA 2 fine-tunes. Open models are especially appealing for organizations that need to keep code in-house for security or confidentiality but still want AI assistance.
* **DeepSeek** – Among the newer names, **DeepSeek** deserves mention as an example of a specialized code-centric AI system. DeepSeek is not just a single model but a platform that integrates an LLM with additional reasoning and tool-use capabilities. It has been noted for emphasizing *efficient inference* and the ability to use external tools like a code execution sandbox. Some comparisons between DeepSeek and base LLaMA models have surfaced, suggesting that DeepSeek’s design leads to stronger performance in certain coding tasks, particularly those requiring reasoning or iterative refinement[34](https://elephas.app/blog/deepseek-vs-llama-2025-comparison-which-local-ai-model-is-best-cm7kddany00ekip0lidqqoeq3#:~:text=powerflower_khi%20highlighted%20a%20fundamental%20difference%3A,query%20based%20on%20certain%20subsets). For instance, DeepSeek’s loop-back reasoning allows it to attempt a solution, check it (even run it if possible), and refine its approach, somewhat akin to an autonomous agent. In one user-conducted test, DeepSeek was able to generate a playable simple game with working code, whereas a comparable LLaMA model produced code that was incomplete[35](https://elephas.app/blog/deepseek-vs-llama-2025-comparison-which-local-ai-model-is-best-cm7kddany00ekip0lidqqoeq3#:~:text=Test%201%3A%20Coding). This points to an emerging trend: combining LLMs with an **execution loop** to verify and improve outputs. It echoes what AlphaCode did (generate many candidates, run them through tests), but now possibly in an interactive single system. DeepSeek is also often run locally, highlighting the progress in making such complex AI accessible outside big labs. While not as famous as GPT-4 or Claude, DeepSeek represents how *domain-specific optimization* (in this case for coding and reasoning) can yield models that outperform more general ones in that domain[36](https://elephas.app/blog/deepseek-vs-llama-2025-comparison-which-local-ai-model-is-best-cm7kddany00ekip0lidqqoeq3#:~:text=DeepSeek%20performs%20significantly%20better%20for,you%20need%20help%20with). It also shows the innovation happening in the open-source AI world post-LLaMA – by building on open models, developers can create niche experts.

Comparative insights from these models: At a high level, **GPT-4** and **Gemini** (and perhaps Claude 2) sit at the top tier in terms of general capability and benchmark performance, with closed weights and the backing of major corporations. They tend to have the highest raw accuracy on difficult tasks and the most advanced natural language abilities combined with coding. **Claude** differentiates itself with context length and an alignment/safety focus. **LLaMA/Code Llama** and derivatives demonstrate the power of open models – while a bit behind the frontier, they are not far off and are improving rapidly with community effort; they excel in customizability. **DeepSeek and similar** indicate a direction of making models more *agentic*, i.e., having them not just produce one-shot answers but engage in a process (planning, executing, verifying).

One interesting observation is how these models can complement each other. For example, an open-source model might be fine-tuned to know a company’s internal code patterns better, reducing hallucinations in that context, while GPT-4 might be used for its reasoning on novel problems. Some projects orchestrate multiple models (choose different AIs for different sub-tasks). The competitive dynamic also pushes rapid innovation: the introduction of GPT-4 led others to scale up and refine (Claude improved, Google accelerated Gemini, Meta released more powerful open models). For end users – software teams – this means more options and potentially *ensemble* solutions where an AI code reviewer might first run a fast open model for initial pass, then use GPT-4 for deeper analysis on flagged parts, etc., achieving a balance of speed, cost, and thoroughness.

### Architectural Innovations Enabling Current AI Capabilities

What makes these modern AI models so much more effective at code analysis compared to earlier approaches? Several architectural and training innovations are key:

* **Transformer architecture & scale**: As mentioned, the self-attention mechanism allows modeling the relationships between all parts of the input, which for code means a function can easily attend to its context, documentation, or usage patterns elsewhere. The ability to scale to billions of parameters means the model can store a vast amount of *implicit knowledge*: programming language syntax and semantics, common libraries and their usage, typical bug fixes, etc. Essentially, much of what an experienced programmer picks up over years (idioms, pitfalls, best practices) is compressed in the weights of an LLM that has seen millions of code examples. This wide knowledge enables generalization: even if the model hasn’t seen *exactly* a given API or scenario, it might analogize from similar ones.
* **Instruction tuning and RLHF**: A base language model is trained to predict the next token, which doesn’t guarantee it will follow user intent or be truthful. Instruction tuning (fine-tuning on prompts and ideal answers, including for code-related instructions) makes the model better at following explicit requests like “Explain this code” or “Don’t proceed if the tests likely fail.” Reinforcement Learning from Human Feedback (RLHF) further refines the model to give answers that humans rate as helpful. For code, this often translates to preferring outputs that are not just correct but also **well-explained** and stylistically appropriate. An example is that GPT-4, when asked to review code, will typically not only point out an issue but also explain why it’s an issue and perhaps suggest a fix – mirroring how a human reviewer would comment. This alignment with human-like communication builds developer trust and makes the AI’s output more actionable.
* **Large Context Windows**: We discussed Claude’s 100k context; similarly, Code Llama models are trained with up to 100k token contexts[37](https://arxiv.org/abs/2308.12950#:~:text=to%20cover%20a%20wide%20range,and%20all%20our%20models%20outperform) (though performance beyond 16k might degrade without special handling). Techniques like **Sparse Transformers** and **LongFormer** inspired architectures that can handle long sequences efficiently, which have been incorporated in some of these models (e.g., LLaMA 2 uses grouped-query attention to better handle longer input). Large context is a game-changer for analyzing code because the model can be given not just a small snippet but the **entire surrounding code** or relevant documentation. This reduces instances of the model guessing or hallucinating missing context. For example, if a function call is made to foo(x) and the model has seen the definition of foo earlier in the prompt, it can reason about what foo expects and thus catch a bug like “passing x without initializing it”. Without seeing foo, it might not realize that’s an issue.
* **Tool use and integration**: A very active area of research is giving LLMs the ability to call external tools – e.g., compilers, debuggers, or test runners – and feed the results back into its analysis. This is sometimes framed as an *agent* or *augmented LLM*. For instance, Microsoft’s recent “AutoGen” framework or OpenAI’s function calling feature allow an LLM to defer certain tasks to tools. In coding, we already see that pattern: an AI writes some code, then runs tests (via an API), sees failures, and then the prompt is extended with “Test X failed with output Y”, so the model can fix the code. This iterative loop was historically done in program repair research with separate components, but with LLMs, a single model can orchestrate it by generating its own follow-up prompts. Some products (like Amazon CodeWhisperer or Replit’s Ghostwriter) are exploring letting the AI directly compile and run code in a sandbox to verify suggestions. Architecturally, this means the model doesn’t have to know everything – it can query a compiler for the exact syntax or use a solver to get a concrete value. Early results show this can dramatically increase correctness in code generation, although it requires careful sandboxing and increases latency.
* **Multi-modal and contextual learning**: With models like Gemini being multimodal, new possibilities open up, such as correlating UI images with code, or error logs (text) with code execution trace. Another innovation is incorporating **architectural hints** like the code’s abstract syntax tree (AST) into the model’s input (some research augmented transformers with structural position embeddings so they understand code structure better than plain sequence of tokens). There are also efforts in **neuro-symbolic methods**, where the neural model is combined with a symbolic logic engine – for example, an LLM could propose an invariant and a separate solver checks it. These hybrid approaches try to get the best of both worlds: the knowledge and flexibility of LLMs with the rigor of formal methods. While not mainstream in tools yet, academic prototypes exist that can, say, prove certain properties by having the LLM suggest a loop invariant and a theorem prover verify it, iterating until success.
* **Model specialization**: Instead of one monolithic model, another architectural direction is *mixture-of-experts* (MoE) or other modular designs where different parts of the model handle different tasks. One can imagine within a code AI, one expert might specialize in Python, another in C, another in debugging, etc. OpenAI’s GPT-4 is rumored to have such qualities (though not confirmed publicly). Even without MoE, ensembles or cascades (where one model filters or ranks outputs of another) act similarly. For instance, one might use a smaller model to quickly identify candidate bug locations and then a larger model to deeply analyze those. Or use a model to generate several fix attempts and a separate classifier model to predict which fix is most likely correct. These architectures echo successful patterns in other AI fields and are being explored to reduce computation while maintaining quality.
* **Continual learning and personalization**: Traditional LLM training is a one-shot huge process after which the model is static. But for code, which is a living, changing ecosystem (new libraries, frameworks, security vulnerabilities emerging), there is interest in making models that can **update** or be personalized. Techniques like fine-tuning on a project’s own code or recent commits can make the AI reviewer much more attuned to that project (e.g., knowing its naming conventions, knowing that a certain function is legacy and shouldn’t be used, etc.). There is a risk of overfitting, but research in parameter-efficient fine-tuning (e.g., LoRA adapters) means one could keep a base model and have plugin “knowledge” modules per project. This is an architectural extension that could become common: the base model handles general reasoning, while a smaller learned module injects project-specific facts. Such personalization would improve both accuracy and developer trust, because the AI would behave more like a team member who knows the codebase.

In summary, the current generation of AI models for code is powered not just by sheer size, but by clever design enabling them to *see more context*, *learn from feedback*, *use external knowledge*, and *specialize when needed*. These innovations distinguish them sharply from the pre-LLM era tools. A static analyzer from 2010 might have a carefully crafted set of rules for, say, null pointer checks. A 2025 AI code reviewer, on the other hand, doesn’t have those rules explicitly; instead, it *learned* them from data and can apply or even explain them on the fly, while also catching higher-level issues (like a misused algorithm) that would be hard to encode in static rules. This flexibility is a game-changer, but it also means the model is a black box in many ways – which leads us to consider the ongoing challenges.

## Persistent Challenges and Open Issues

Despite remarkable progress, several challenges remain before AI models can fully handle pull request analysis and bug resolution autonomously in a trustworthy manner. We discuss a few key issues: **contextual understanding limits, explainability, and developer trust**, as well as related concerns like evaluation and ethics.

**Understanding Complex Context**: LLMs can process a lot of information, but understanding a complex software system – with layers of architecture, implicit assumptions, and runtime behavior – is still very hard. For example, a pull request might have no issues on its own, but might cause a performance regression when integrated. Detecting that may require understanding not just the code change but how it affects system load, something not directly inferable from static code. Tools and AI might need to incorporate dynamic analysis (running tests, measuring performance) to catch such issues. Another scenario: the correctness of a change might depend on domain knowledge (e.g., “we must comply with GDPR, so logging this data is a bug”). A pure code analyzer won’t know that unless such context is given. Current AI models typically don’t have an explicit knowledge base of *project requirements* or *design documents* unless included in the prompt. Future development might integrate requirements tracing – linking code to design – so the AI can check code against design rules. Until then, there will be gaps where the AI misses an issue because it lacks contextual knowledge beyond the code itself. Incorporating more project context (like issue trackers, documentation, previous discussions) into analysis is an ongoing area of research.

**Explainability and Transparency**: Modern AI models are often criticized as “black boxes” – they output an answer but it’s not clear *why*. In a sensitive domain like software bugs (where acting on a wrong recommendation can introduce new bugs or downtime), explainability is crucial. Developers need to understand the rationale for a suggested fix or why a piece of code was flagged. Traditional static analysis tools usually provide a specific reason (e.g., “possible null dereference on line 42, because variable X might be null if Y is false”). AI models sometimes provide reasoning in their output (especially if prompted to think step-by-step), but that reasoning isn’t guaranteed to reflect the true internal decision process – it might just be a plausible explanation. There’s active research on *mechanistic interpretability* of LLMs, some by Anthropic as referenced in a McKinsey report[38](https://www.mckinsey.com/capabilities/quantumblack/our-insights/building-ai-trust-the-key-role-of-explainability), aiming to identify which parts of the model do what (e.g., one can sometimes find neurons or attention heads that track specific patterns like an if-else scope or a function call hierarchy). In the near term, a practical way to enhance explainability is to have the AI *cite sources or rules* it is relying on. For instance, an AI could say “I suspect a bug because this function *foo()* is often used after checking *bar* for null, which is not done here.” This builds a sort of implicit citation to its training knowledge. Another approach is **contrastive explanation**: the AI could be prompted to explain why the code might be correct vs. why it might be wrong, giving both sides. This provides the developer with insight into the decision boundary. Some AI coding assistants now come with an “explain” mode (for example, you highlight a piece of code and it explains what it does in plain English). This fosters trust as the developer can verify the explanation matches their understanding. Nonetheless, making the *internals* of LLM decisions transparent is unsolved – we largely have to trust the model’s output and verify it through testing or code review, much like we would do with a human junior developer’s work. There is a recognized trade-off between a model’s accuracy and its interpretability: the more complex and high-performance the model, often the harder it is to explain[39](https://www.mckinsey.com/capabilities/quantumblack/our-insights/building-ai-trust-the-key-role-of-explainability). Research in **Explainable AI (XAI)** is trying to break this trade-off by developing methods to interpret even very large models, which will be important as we incorporate them into critical development processes.

**Developer Trust and Adoption**: Introducing AI into a development workflow raises human factors questions. Developers might be initially skeptical of AI recommendations, especially if early experiences show the AI making mistakes. If an AI code review tool reports many issues and half of them are not valid or not important, developers could develop “alert fatigue” and start ignoring it – similar to what happened with some static analysis tools historically that were too noisy. Earning trust requires a high signal-to-noise ratio and the ability to show *confidence* or *uncertainty*. It would help if the AI can indicate its level of certainty (e.g., “I am 90% sure this is a bug” vs “just a minor suggestion”). Current LLMs typically do not output calibrated probabilities, though some calibration can be imposed by post-processing their logits or by fine-tuning them to be more cautious in phrasing. Another aspect is **responsibility and control**: developers need to feel that *they* are still driving the decision, with the AI as a partner. If an AI suggests a fix, but the team doesn’t understand it, they may be rightly hesitant to accept it. Encouraging a workflow where AI suggestions are treated as recommendations for review (and perhaps requiring an explanation as noted) can mitigate blind trust. Some industry surveys indicate that while many developers are open to AI assistance, they want the ability to verify changes and not have AI auto-merge code without human oversight. Over time, as AI proves itself (perhaps by consistently catching things humans missed, or by never introducing a bad fix over months of usage), trust can grow. It’s analogous to the adoption of compilers in the 1960s – early programmers were skeptical that a compiler could optimize code better than they could write manually, but compilers gained trust by eventually outperforming human optimizations in most cases. We may see a similar arc: initially, AI is a co-pilot with training wheels, and perhaps one day it could autopilot trivial changes.

A subtle factor in trust is **accountability**: If the AI is wrong, who is responsible? In a professional setting, the developers or the company are, not the tool vendor. This makes teams cautious. It underscores the need for AI suggestions to be *optional and reviewable*. Some organizations might require that all AI-generated code be explicitly marked in commits, for traceability. Tools could facilitate that by tagging their output. Additionally, biases or blind spots in the AI could undermine trust. If, for example, a code analysis AI is found to systematically miss certain types of security issues because they weren’t in its training, developers will not rely on it for security reviews. Continuous evaluation and *benchmarking* of the AI on relevant metrics is crucial even after deployment, to monitor such gaps.

Speaking of benchmarking, **evaluation practices** themselves are an open challenge. Benchmarks like HumanEval or SWE-bench are great for research, but any fixed benchmark can be gamed or overfit. We already saw issues of data leakage in SWE-bench (the AI had seen some answers before)[40](https://arxiv.org/html/2410.06992v1#:~:text=SWE,LLM%E2%80%99s%20knowledge%20cutoff%20dates%2C%20posing). This means as we benchmark new models, we must ensure test data is truly novel (e.g., collecting new issues that are created after the model’s training cutoff, as done in SWE-Bench+ to prevent training data leakage[41](https://arxiv.org/html/2410.06992v1#:~:text=SWE,respectively)). There’s also a need for more **holistic evaluations**: beyond pass rates, consider metrics like *code quality*, *maintainability of the fixes*, *lack of regression introduction*, etc. Some of these are hard to quantify. Human studies, where developers assess AI contributions, are useful but expensive and not easily scalable. Automatic metrics like “did the fix introduce any new failing tests?” can be used in repair scenarios. The community is moving toward complex eval setups, for instance, the **LLM Debugging Challenge** might give a broken codebase and see if an AI can not only fix the bug but also not break anything else, measured via a full test suite. Such comprehensive evals can better simulate real PR scenarios.

Another challenge is **ethical and legal considerations**. AI models trained on public code might regurgitate licensed code (a concern raised with Copilot). In PR analysis, this is less likely to be an issue (since it’s analyzing your code, not generating new code from elsewhere), but if an AI suggests a fix that is literally identical to a snippet from an open source project under an incompatible license, it could raise IP questions. Solutions like **secure model training** (excluding non-permissive licensed code) and **input attribution** (ensuring the model’s suggestions stem from the input context) are being explored to mitigate this.

Finally, we consider **the role of AI in team dynamics**. Will AI replace junior developers or simply augment them? The consensus in industry so far is augmentation: AI takes over repetitive or boilerplate tasks, or acts as an intelligent assistant, while humans focus on design, complex problem solving, and verification. In code reviews, an AI might handle the first pass, and humans do the final sign-off. There is an ongoing challenge to integrate AI smoothly such that it enhances productivity without causing disruption or over-reliance. Over-dependence on AI tools can indeed lead to complacency, where developers might apply suggestions without fully understanding them[42](https://www.codestringers.com/insights/ai-code-review/#:~:text=AI%20Code%20Review%20is,intuition%20and%20domain%20expertise.). Educators are already concerned that young developers might not learn certain skills if they rely on AI for answers. To build trust long-term, AI tools should perhaps have a *teaching mode* – instead of just giving the solution, they can Socratically guide the developer to understand the issue. This would ensure the human is still learning and in control, using the AI as an accelerant to expertise rather than a crutch that bypasses it.

In conclusion, while AI models like GPT-4, Claude, Gemini, and Code Llama have opened the door to partially automating PR reviews and bug fixes, there remain significant challenges on the road to fully autonomous code maintenance. The theoretical foundations we discussed from the pre-LLM era are not obsolete – concepts from formal methods and static analysis will likely be embedded in future AI to address some of these challenges (for example, using formal verification to check an AI’s suggested patch). The current state of the art is a hybrid approach: AI performs impressively on many tasks and improves rapidly, but human oversight, domain knowledge, and rigorous evaluation are essential to a successful application in bug resolution. As research continues, we expect to see improvements in the explainability of AI (perhaps via architectural changes or auxiliary models that interpret the main model), better handling of large contexts (maybe through hierarchical reasoning or memory mechanisms), and refined techniques to align AI behavior with developer values and project goals. The ultimate measure of success will be an AI system that developers treat as a trusted colleague – one that reliably enhances the quality of code and the productivity of the team, while being transparent about its reasoning and aware of its limits.

# References

Barr, E. T., Bird, C., & Zhang, T. (2012). *Exploring the naturalness of software*. Proceedings of the 34th International Conference on Software Engineering (ICSE). (Analyzes statistical naturalness in source code, showing code’s predictability with n-gram models.)

Chen, M. et al. (2021). *Evaluating Large Language Models Trained on Code*. arXiv:2107.03374. (OpenAI Codex paper introducing the HumanEval benchmark and reporting Codex performance of 28.8% pass@1 on it.)

Clarke, E. M., Emerson, E. A., & Sifakis, J. (2009). **Model Checking**. *Communications of the ACM, 52*(11), 74-84. (Turing Award lecture and overview of model checking, its history and impact on hardware and software verification.)

Cousot, P., & Cousot, R. (1977). *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. Proceedings of the 4th POPL, 238–252. (Foundational paper introducing abstract interpretation as a theory of sound static program analysis.)

Gupta, R., Kanade, A., & Shevade, S. (2017). *DeepFix: Fixing common C language errors by deep learning*. Proceedings of the AAAI Conference on Artificial Intelligence. (Demonstrates an early neural approach to program repair for student-written C programs using sequence-to-sequence models.)

Jimenez, D., Song, L., et al. (2024). *LLM-based Automated Bug Fixing: Evaluating on Real-World Pull Requests (SWE-bench)*. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE). (Introduces the SWE-bench dataset of GitHub issues and pull requests, evaluating GPT-3.5 and GPT-4 agents on automated bug resolution tasks.)

Nagy, B. (2023, April 4). *We put GPT-4 in Semgrep to point out false positives & fix code*. Semgrep Blog. (Case study of integrating GPT-4 with a static analysis tool to triage and automatically fix security findings, discussing successes and limitations.)

OpenAI. (2023, July). *Introducing SWE-bench Verified*. OpenAI Blog. (OpenAI’s release of a validated subset of the SWE-bench issues, ensuring high-quality problem descriptions and reliable test cases for evaluating code-generating models.)

Thomson, P. (2021). **Static Analysis: An Introduction**. *ACM Queue, 19*(4). (Provides a historical perspective on static analysis tools, their evolution since the 1970s, and discusses fundamental limits like the halting problem and approaches such as lint, abstract interpretation, and extended static checking.)

Touvron, H., et al. (2023). *LLaMA: Open and Efficient Foundation Language Models*. arXiv:2302.13971. (Meta AI’s report on LLaMA 1, an open 65B-parameter model, which along with LLaMA 2 laid the groundwork for many open-source derivatives including Code Llama.)

Tunstall, L., van Biljon, E., et al. (2023). *StarCoder: May the Source Be With You*. arXiv:2305.06161. (Describes the StarCoder model by BigCode, a 15B parameter LLM trained on source code, and introduces early versions of what became BigCodeBench.)

Vaswani, A., et al. (2017). *Attention Is All You Need*. *Advances in Neural Information Processing Systems (NeurIPS)*, 30, 5998-6008. (Seminal paper introducing the Transformer architecture, which underpins modern LLMs including those for code.)

Zhuo, T. Y., et al. (2025). *BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions*. arXiv:2406.15877. (Presents a comprehensive benchmark for code generation evaluating 60 LLMs, highlighting challenges in multi-call, multi-library programming tasks and showing current models’ performance caps around 60% success on these tasks.)

Anthropic. (2023). *Claude 2 Model Card*. (Documentation for Anthropic Claude 2, noting its 100k token context window and improvements in coding ability and reasoning.)

Knight, W. (2023, June 26). *Google DeepMind’s CEO Says Its Next Algorithm Will Eclipse ChatGPT*. **Wired**. (Interview with Demis Hassabis discussing Gemini’s goals to integrate AlphaGo-like planning with LLM capabilities.)

Metz, C. (2024, Feb 8). *Google Releases Gemini, an AI Mightier Than ChatGPT*. **The New York Times**. (Coverage of Gemini’s launch, its multimodal capabilities, and competition with OpenAI’s models.)

Hindle, A., Barr, E. T., Su, Z., Gabel, M., & Devanbu, P. (2012). *On the Naturalness of Software*. *Communications of the ACM, 59*(5), 122-131. (Introduces the concept that software is statistically natural and predictable, encouraging the application of language models to code.)

Pradel, M., & Sen, K. (2018). *DeepBugs: A Learning Approach to Name-based Bug Detection*. *Proceedings of the ACM on Programming Languages, 2*(OOPSLA), 147. (Uses neural networks to learn typical usage patterns and detect anomalies in code, exemplifying early ML for bug finding.)