Expanded edition, including GPT-5!

Sonar

The Coding Personalities of Leading LLMs

A State of Code Report

Table of Contents

Introduction 3

Our approach 4

Foundation of shared strengths and flaws 5

Shared strengths 5

Shared flaws 7

Coding personalities 12

Personality traits 12

Coding archetypes 15

The baseline performer 16

The senior architect 16

The balanced predecessor 17

The efficient generalist 17

The unfulfilled promise 18

The rapid prototyper 18

Why “more capable” can be riskier 19

Conclusion 20

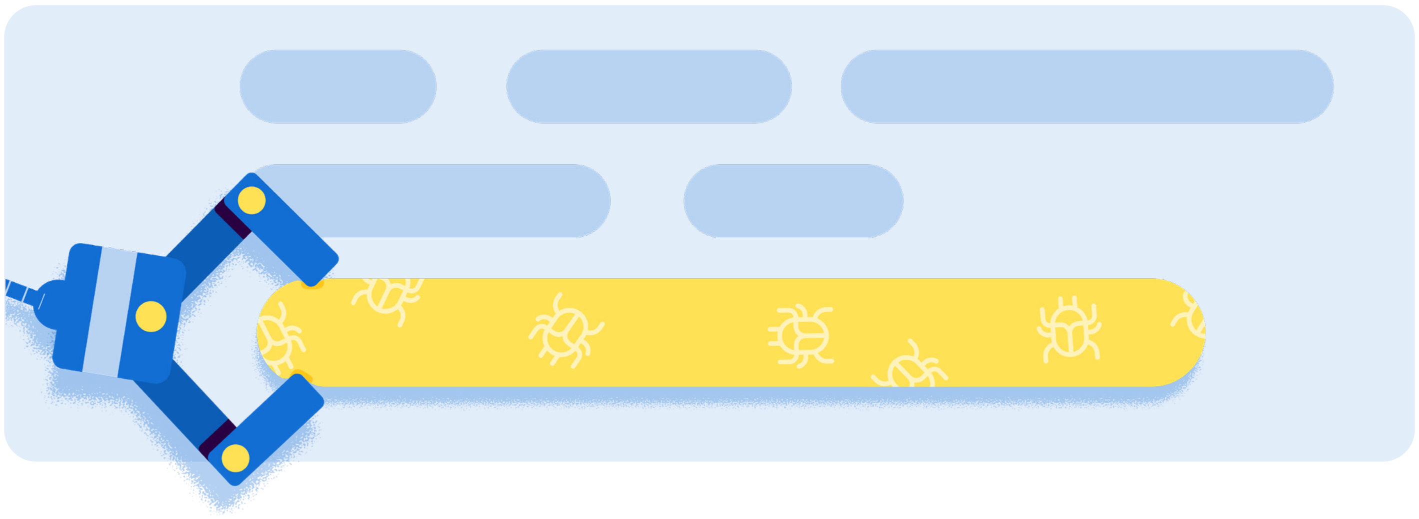
Introduction: Beyond the performance benchmark

AI has embedded itself in the software development lifecycle (SDLC) at an extraordinary speed. Tools such as Claude Code, Cursor, and GitHub Copilot are increasingly standard and necessary tools for software developers. Underlying all of these tools are Large Language Models (LLMs), some general purpose from companies like OpenAI, Anthropic, Meta, and Google, and some specially built for coding use cases.

Understanding the true capabilities of these models is of critical importance as the industry develops. However, the typical methods for evaluating these capabilities do not give a complete, high-resolution picture. A primary evaluation approach focuses on assessing LLM performance against benchmarks that test their ability to solve difficult coding challenges—what we consider to be an important but narrow test.

This relentless focus on performance benchmarks leads to what experts describe as “super spiky capability distributions.” As we will show in this report, this focus on performance benchmarks leads to LLMs that can solve difficult coding challenges, but do not necessarily write good code—that is, code that is reliable, secure, and maintainable.

It is critical that we move beyond relying only on performance benchmarks, and start to understand the full mosaic of capabilities of coding models, their personalities and habits, good and bad. By doing so, we can ensure that we have a more nuanced understanding that helps us more consistently select the best model for the job to be done.



Our approach: A deep analysis of LLM-generated code

To move beyond the standard performance benchmarks, Sonar developed a proprietary analysis framework for assessing LLM-generated code. This approach leverages the core strengths of the SonarQube Enterprise static analysis engine, which is built on over 16 years of experience in detecting complex bugs, vulnerabilities, and code smells in enterprise-grade software.

We combined this deep code analysis with best practices from coding model evaluations. Our analysis covers two classes of models: five leading "non-reasoning" LLMs (Anthropic's Claude Sonnet 4 and 3.7, OpenAI's GPT-4o, Meta's Llama 3.2 90B, and the open-source OpenCoder-8B) and one new "reasoning" model, GPT-5. To create a direct, apples-to-apples comparison, we evaluated GPT-5 in its **minimal reasoning mode**, which is analogous to the non-reasoning approach of its peers. For a full analysis of GPT-5's tunable reasoning capabilities, please see our supplemental report, "How Reasoning Impacts LLM Coding Models." Each model was tasked with completing over 4,442 distinct Java programming assignments from recognized sources, including MultiPL-E-mbpp-java, MultiPL-E-humaneval-java, and ComplexCodeEval.

Our goal was to provide a clear, objective analysis that creates opportunities for improvement and informed decision-making. For **model developers**, our findings offer a roadmap that goes beyond traditional performance benchmark scores, highlighting concrete areas to improve their coding models. For **software developers and their organizations**, our work provides critical insights needed to choose the right models for the right tasks, and ensure they are used safely and effectively.

A foundation of shared strengths and shared flaws

Before we discuss the unique personalities of each LLM, it is important to highlight the common foundation of strength and weaknesses that all models share. This section will detail these shared characteristics, starting with the powerful capabilities that have driven their widespread adoption.

As outlined in our approach, this report evaluates two distinct classes of models: traditional, non-reasoning LLMs and a new reasoning model, GPT-5. This distinction is critical to understanding the data that follows. In general, reasoning models represent a trade-off: their ability to reason about a problem often leads to higher functional correctness and better avoidance of common security flaws. However, this comes at the cost of generating significantly more verbose and complex code, which can introduce a new class of subtle, harder-to-detect bugs and a greater long-term maintainability burden.

Shared strengths

The code generation capabilities of large language models are fundamental to their growing application in software development. Our benchmark analysis provides quantitative data confirming a consistent set of shared competencies.

Table 1: LLM performance on MultiPL-E Java benchmarks

MultiPL-E benchmarks	GPT-5-minimal	Claude Sonnet 4	Claude 3.7 Sonnet	GPT-4o	Llama 3.2 90B	OpenCoder-8B
HumanEval (158 tasks)	91.77%	95.57%	84.28%	73.42%	61.64%	64.36%
MBPP (385 tasks)	68.13%	69.43%	67.62%	68.13%	61.40%	58.81%
Weighted test Pass@1 avg	75.37%	77.04%	72.46%	69.67%	61.47%	60.43%

Syntactic and boilerplate generation

The ability to generate syntactically valid code is a fundamental requirement for a coding assistant. The design of our benchmarks provides a direct measure of this skill, as syntactically flawless code is a prerequisite for passing any functional test. The high pass rates recorded in the table are therefore a clear indicator of this reliability. For example, Claude Sonnet 4's success rate of 95.57% & GPT-5-minimal's success rate of ~92% on HumanEval demonstrates a very high capability to produce valid, executable code.

Technical competence

Beyond correct syntax, the models demonstrate robust capabilities in algorithmic problem-solving. The "weighted test Pass@1 average" provides a balanced measure of this capability, and the scores achieved by models like GPT-5-minimal (75.37%) and Claude Sonnet 4 (77.04%) confirm a high degree of reliability.

Conceptual translation

Our analysis points to the models' notable capability for conceptual translation across different programming languages, suggesting their core capability is understanding abstract logic and translating it across linguistic environments.

Shared flaws

While the shared strengths drive AI's utility, we also found a consistent pattern of shared weaknesses that diminishes the overall effectiveness of the coding models.

A lack of security consciousness

All models demonstrate security weaknesses, but the introduction of reasoning models like GPT-5-minimal has fundamentally shifted the risk profile. Non-reasoning models consistently produce a high percentage of **'BLOCKER'** severity vulnerabilities. In contrast, GPT-5-minimal produces code with a vulnerability density 3-6 times lower than its peers. However, this comes at a cost: it trades common, well-understood flaws for more subtle, implementation-specific vulnerabilities. For example, it still produces "Path-traversal & Injection" flaws at a significant rate (20%), but it introduces a dramatically higher percentage of vulnerabilities related to "Inadequate I/O error-handling" than any other model.

Table 2: Subcategories of security vulnerabilities and their origins (% of total vulnerabilities for model)

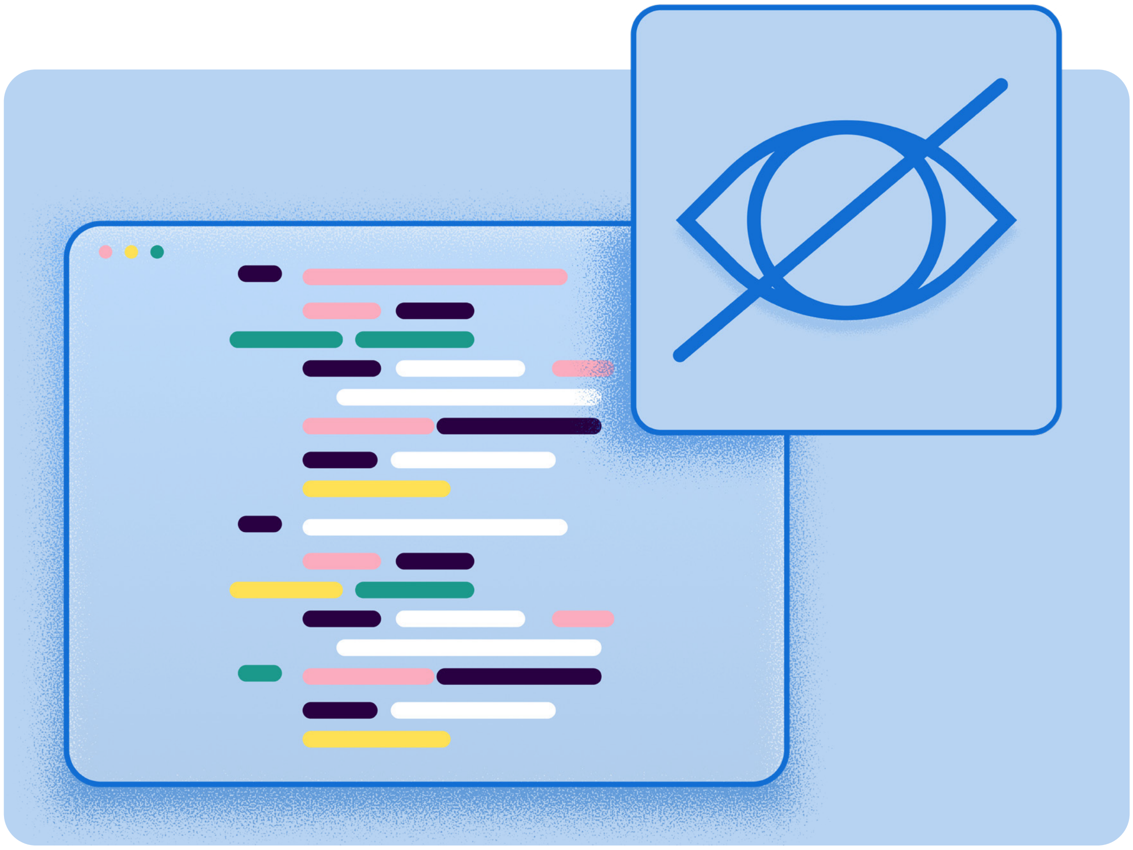
Category	GPT-5-minimal (%)	Claude-Sonnet 4 (%)	Claude-3.7 Sonnet (%)	GPT-4o (%)	Llama 3.2 90B (%)	OpenCoder -8B (%)
Path-traversal & Injection	20	34.04	31.03	33.93	26.83	28.36
Hard-coded credentials	5	14.18	10.34	17.86	23.58	29.85
Cryptography misconfiguration	23.33	24.82	23.28	19.64	22.76	22.39
XML external entity (XXE)	10	10.64	15.52	13.39	19.51	5.97
Inadequate I/O error-handling	30	4.96	7.76	7.14	4.88	7.46
Certificate-validation omissions	8.33	2.84	4.31	2.68	0	2.99
Other	5	7.8	7.76	4.46	1.63	0

These issues are further underscored by the severity of the vulnerabilities introduced. The following table breaks down this severity, revealing a fundamental difference between the reasoning and non-reasoning models. Our analysis found that a majority of vulnerabilities for every non-reasoning model are of 'BLOCKER' severity, the highest possible rating.

Table 3: Vulnerability Severity Distribution (% of total vulnerabilities)

LLM Model	BLOCKER %	CRITICAL %	MAJOR %	MINOR %
GPT-5-minimal	35.00	31.67	3.33	30.00
Claude Sonnet 4	59.57	28.37	5.67	6.38
Claude 3.7 Sonnet	56.03	28.45	5.17	10.34
GPT-4o	62.50	23.21	5.36	8.93
Llama 3.2 90B	70.73	22.76	1.63	4.88
OpenCoder-8B	64.18	26.87	1.49	7.46

This is not a matter of occasional hallucination but rather a structural failure rooted in the LLMs' foundational design and training. LLMs struggle to prevent injection flaws because doing so requires taint-tracking from an untrusted source to a sensitive sink, a non-local data flow analysis that is beyond the scope of their typical context window. They generate hard-coded secrets (like passwords) because these flaws exist in their training data.



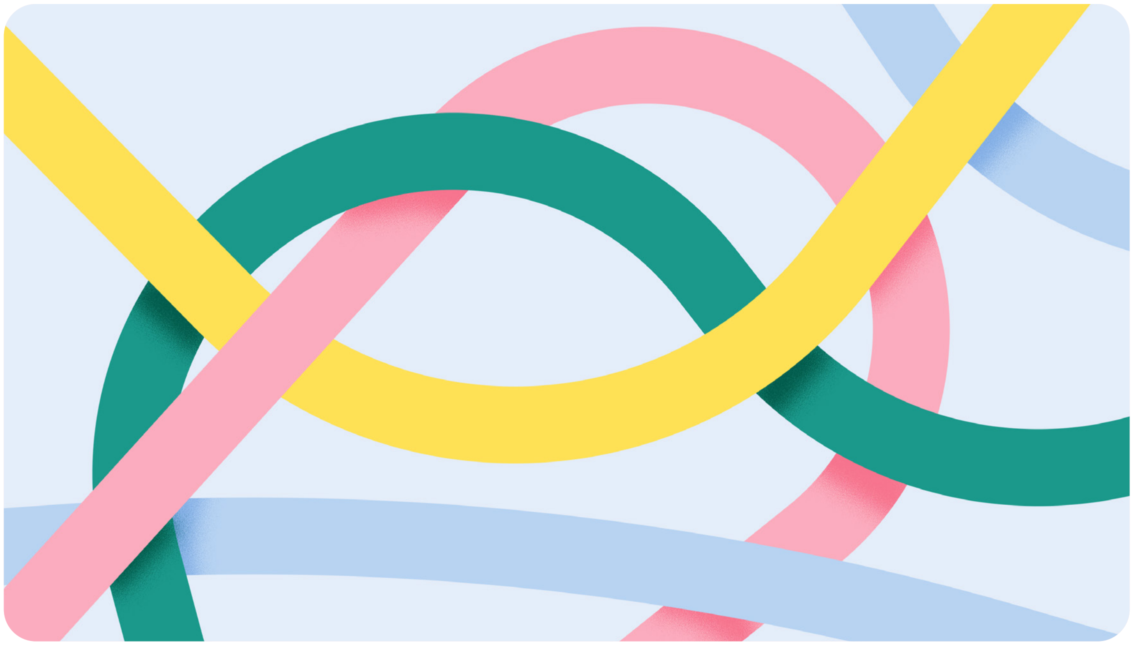
A struggle with engineering discipline

All LLMs evaluated demonstrate a consistent struggle with the core tenets of software engineering, particularly in areas that require a global, context-aware understanding of the application. Severe bugs like resource leaks (e.g., failing to close file streams) and API contract violations (e.g., ignoring critical error return values) appear consistently across all models. Reasoning presents a trade-off: higher reasoning in GPT-5 reduces fundamental logical errors like "Control-flow mistakes" but drastically increases advanced flaws like "Concurrency / threading" bugs, as the model attempts more complex solutions.

The table below details the most common bug categories.

Table 4: Subcategories of bugs and their origins (% of total bugs for model)

Category	GPT-5-minimal (%)	Claude Sonnet 4 (%)	Claude 3.7 Sonnet (%)	GPT-4o (%)	Llama 3.2 90B (%)	OpenCoder-8B (%)
Control-flow mistake	24.26	14.83	23.62	48.15	31.06	21.37
API contract violation	9.18	10.29	14.12	8.64	14.9	19.35
Exception handling	9.18	16.75	16.71	11.6	14.39	14.52
Resource management / leak	11.48	15.07	8.36	7.41	12.88	9.68
Type-safety / casts	5.25	11.24	12.97	7.9	6.82	7.66
Concurrency / threading	**20**	9.81	1.44	1.73	1.26	2.82
Null / data-value issues	3.77	7.89	7.49	8.89	5.81	6.85
Performance / structure	3.77	4.31	6.34	3.95	2.78	5.24
Pattern / regex	0.82	2.63	1.15	0.74	0.25	2.42
Data-structure bug	0	1.44	1.15	0	1.01	1.61
Serialization / serializable	0	0	0.58	0	0.76	1.61
Other	8.2	5.74	6.05	0.99	8.08	6.85



An inherent bias towards messy code

Perhaps the most fundamental shared flaw is a deep, inherent tendency towards producing “messy” code. For all non-reasoning models, code smells are the vast majority of issues. This is exacerbated by reasoning models like GPT-5; its focus on correctness and security comes at the cost of generating complex code, which in turn introduces a massive number of maintainability issues and a high proportion of ‘CRITICAL’ code smells.

Table 5: Distribution of issue types by LLM (% of total issues)

LLM	% Bugs	% Vulnerabilities	% Code smells
GPT-5-minimal	4.67%	0.46%	94.87%
Claude-Sonnet-4	5.85%	1.95%	92.19%
Claude-3.7-Sonnet	5.35%	1.76%	92.88%
GPT-4o	7.41%	2.05%	90.54%
Llama 3.2 90B	7.71%	2.38%	89.90%
OpenCoder-8B	6.33%	1.72%	91.95%

This massive volume of code smells is not just a matter of quantity, but also of severity. The following table provides the most direct evidence of the maintainability trade-off. The data shows that the entire GPT-5 family is a significant outlier, producing a much higher proportion of ‘CRITICAL’ code smells than any other model. This is the “cost” of its high functional performance: a direct and immediate increase in severe technical debt.

Table 6: Subcategories of code smells and their origins (% of total code smells for model)

Category	GPT-5minimal (%)	Claude Sonnet 4 (%)	Claude 3.7 Sonnet (%)	GPT-4o (%)	Llama 3.2 90B (%)	OpenCoder-8B (%)
Dead / unused / redundant code	14.7	14.83	17.43	26.3	34.82	42.74
Design / framework best practices	12.44	22.26	18.58	20.81	18.84	12.45
Assignment / field / scope visibility	9.94	11.96	15.35	13.21	11.32	11.95
Collection / generics / param / type	14.82	13.94	11.23	9.92	9.03	7.89
Regex / pattern / string / format	6.76	13.7	11.8	7.36	6.81	5.29
Cognitive / computational complexity	10.67	4.25	8.43	3.73	2.67	2.79
Control / conditional-logic smell	2.92	4.67	3.91	4.03	3.02	2.2
Deprecation / obsolete	0.82	2.01	2.34	2.08	2.89	4.01
Naming / style / documentation	2.1	2.69	2.5	2.84	2.16	1.89
Exception-handling smell	0.03	0.05	0.08	0.06	0.02	0.06
Other	24.75	9.64	8.33	9.64	8.41	8.72

These findings paint a clear picture of the shared baseline for the current generation of LLMs. On one hand, they share a powerful set of strengths, from generating syntactically correct code to solving complex algorithmic problems, which makes their emergence is so compelling. On the other hand, they are all built with the same blind spots: a consistent inability to write secure code, a struggle with engineering discipline, and an inherent bias towards generating technical debt.

To effectively leverage AI in coding, developers need to be prepared to recognize and compensate for the weaknesses in the models. Understanding the shared strengths and flaws is a crucial first step. However, just as every developer has their own personality and coding style, LLMs also exhibit their own individual styles. Security, quality, and reliability issues come to life in different ways in different models, and it is crucial to understand the nuances to get the best results.

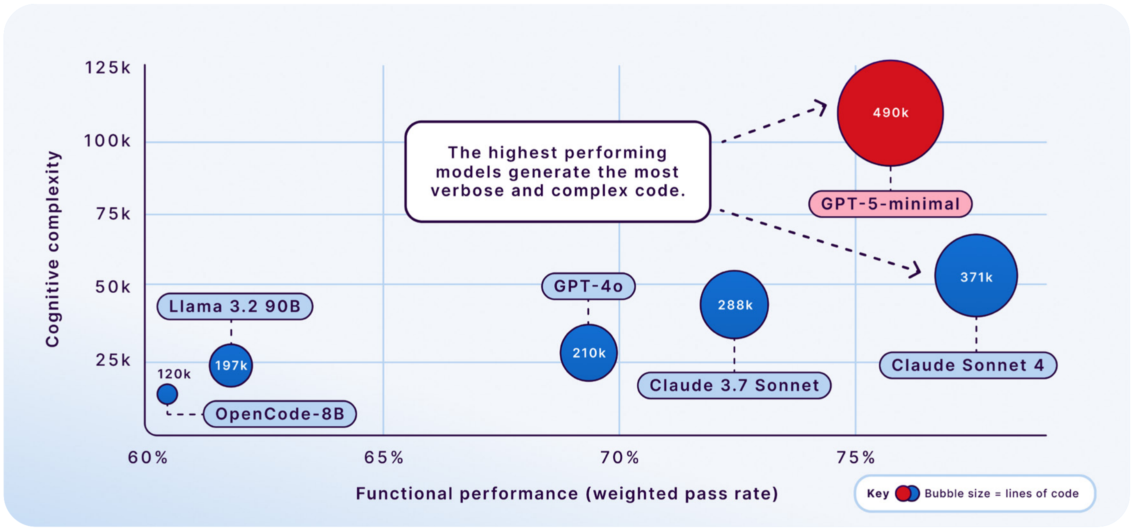
The coding personalities of leading LLMs

If the LLMs have many shared strengths and flaws, why does each LLM's code feel so different in production? This section confronts that apparent contradiction. Our analysis shows that each LLM has a unique and inherent style, a measurable "coding personality."

Coding personality traits

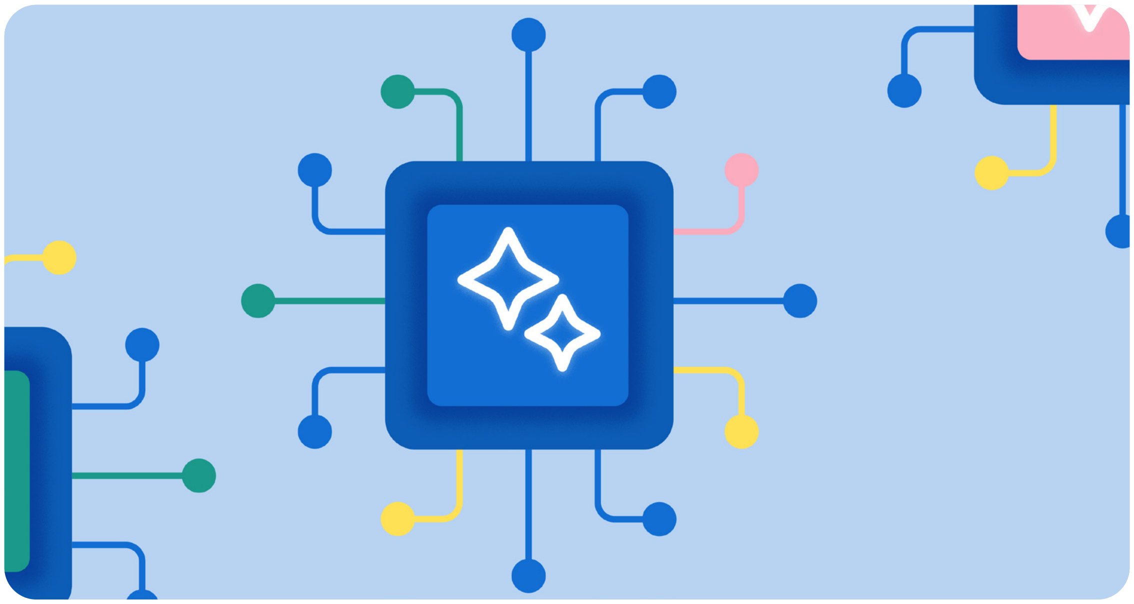
The evidence for these distinct coding personalities is not anecdotal—it is quantifiable in the most basic structural metrics of the generated code. While most models exhibit a single, consistent style, the introduction of reasoning models like GPT-5 adds a new dimension. For these models, the personality is not static; it's a spectrum. Key traits can change based on the chosen reasoning mode, creating a range of behaviors from a single underlying model.

Before we break down the individual traits, the following chart provides a visual summary of the LLM landscape where we tested six of them. It clearly illustrates a fundamental trade-off in the current generation of models: the highest-performing models consistently achieve their results by generating a significantly larger and more complex volume of code. As the chart shows, models like GPT-5-minimal and Claude Sonnet 4 occupy a distinct space, pushing the boundaries of functional performance at the cost of a massive increase in complexity and verbosity.



Our analysis groups these metrics into three primary traits that define each model's coding style:

- **Verbosity:** The sheer volume of code a model generates to solve a given set of tasks.
- **Complexity:** The structural and logical intricacy of the generated code, measured by metrics like cyclomatic and cognitive complexity.
- **Communication and documentation:** The density of comments in the code, which reveals the model's tendency to explain its work.



Verbosity

The most immediate personality trait that emerges is a model's verbosity. An analysis of the total lines of code (LOC) generated to solve the same set of 4,442 tasks reveals a huge stylistic difference. For instance, GPT-5-minimal demonstrated a highly-verbose personality, generating 490,010 LOC, second only to Claude Sonnet 4. In stark contrast, the OpenCoder-8B model was far more concise, producing only 120,288 LOC to address the exact same problems.

This is not a simple matter of length—it reflects a fundamental difference in approach. One model is expansive and comprehensive, attempting to build a complete, self-contained solution. The other is direct and economical, aiming for the quickest route to a working solution. It's not about one being better than the other, but this seemingly small stylistic decision has a big impact. A verbose model may produce code that is harder to review and navigate, while a concise model might omit important context or safeguards, demanding more effort from the human developer to make it production ready.

Complexity

Beyond sheer volume, the inherent complexity of the generated code quantifies the thinking style of the AI. Using metrics like cyclomatic and cognitive complexity, which measure the structural and logical difficulty of understanding code, reveals another clear personality trait.

GPT-5-minimal, one of the most verbose models, also produced the most intricate solutions, with a total cognitive complexity score of 111,133. This is more than twelve times the complexity of the code from the concise OpenCoder-8B, which scored 13,965. This metric serves as a proxy for the model's problem-solving philosophy. A high-complexity score suggests a personality that favors building elaborate, multi-layered solutions. A low score indicates a more linear, straightforward approach. While complex solutions may be necessary for difficult problems, they also create a larger surface area for bugs and increase the cognitive load on human developers who must maintain the code over time.

Communication and documentation

A third personality trait is the models' communication style, revealed through their documentation habits. The density of comments in the generated code indicates whether the model's style is to explain its work or to assume its logic is self-evident. Claude 3.7 Sonnet proved to be a creative commenter, with a comment density of 16.4%. At the other end of the spectrum, GPT-5-minimal proved to be a poor documentarian, with a comment density of only 2.1%. This feature has real-world consequences for team collaboration and maintainability. A well-commented codebase can onboard new developers more quickly and simplify debugging. The fact that models exhibit such consistent but different commenting behaviors underscores that they are not neutral code generators—they are opinionated authors with distinct communication styles.

These foundational metrics are not just output statistics—they are the behavioral signatures of an AI's underlying personality, setting the stage for a deeper analysis of their strengths and weaknesses.

The following table presents the foundational data for this analysis, compiling these key metrics from 4,442 identical programming tasks to establish a quantitative baseline for each model's unique signature.

Table 7: Comparative Code Generation Metrics Across LLMs

LLM Model	Lines of Code (LOC)	Comments (%)	Cyclomatic Complexity	Cognitive Complexity
GPT-5-minimal	490,010	2.10%	145,099	111,133
Claude-Sonnet-4	370,816	5.10%	81,667	47,649
Claude-3.7-Sonnet	288,126	16.40%	55,485	42,220
GPT-4o	209,994	4.40%	44,387	26,450
Llama 3.2 90B	196,927	7.30%	37,948	20,811
OpenCoder-8B	120,288	9.90%	18,850	13,965

The coding archetypes of leading LLMs

With a full view of their individual personality traits, we can define "coding archetypes." Just as a hiring manager assesses a human candidate's resume, we can build a narrative dossier for each LLM, using a wealth of data to bring its personality to life. This approach moves beyond simplistic rankings to provide a nuanced understanding of each model's relative strengths, weaknesses, and ideal use cases.

The following matrix provides a high-level summary of these personalities, combining quantitative metrics with qualitative archetypes to serve as a reference for the detailed profiles that follow.

Table 8: LLM coding archetypes

LLM	Coding archetype	Functional skill (pass rate %)	Issue density (Issues/ KLOC)	Verbosity (LOC)	Cognitive complexity	Dominant flaw type (% of total issues)
GPT-5-minimal	The baseline performer	75.37%	26.65	490,010	111,133	94.87% code smells
Claude Sonnet 4	The senior architect	77.04%	19.48	370,816	47,649	92.2% code smells
Claude 3.7 Sonnet	The balanced predecessor	72.46%	22.82	288,126	42,220	92.9% code smells
GPT-4o	The efficient generalist	69.67%	26.08	209,994	26,450	90.5% code smells
Llama 3.2 90B	The unfulfilled promise	61.47%	26.20	196,927	20,811	89.9% code smells
OpenCoder-8B	The rapid prototyper	60.43%	32.45	120,288	13,965	92.0% code smells

The baseline performer [GPT-5-minimal]

This is the entry-level reasoning mode. It delivers strong performance that is superior to most non-reasoning models. Its personality is defined by having a more “traditional” risk profile compared to more advanced models. It produces common and well-understood flaws, such as a significant rate of “Path-traversal & Injection” vulnerabilities (20%) and basic “Control-flow mistake” bugs. At the same time, it introduces a new class of risk with its high verbosity and complexity, leading to the highest proportion of CRITICAL code smells of any model.

The senior architect [Claude Sonnet 4]

This LLM codes like a seasoned and ambitious architect tasked with building enterprise-grade systems. It exhibits the highest functional skill, successfully passing 77.04% of the benchmark tests. Its style is verbose and highly complex, as it consistently attempts to implement sophisticated safeguards, error handling, and advanced features, mirroring the behavior of a senior engineer.

This very sophistication creates a trap: teams may feel the code is safer because it looks advanced, while in reality it likely introduces more complex, high-severity bugs like resource leaks.

The very sophistication of the model creates a lot of opportunities for higher-risk bugs that plague complex, stateful systems. Its unique bug profile reveals a high propensity for difficult concurrency and threading bugs (9.81% of its total bugs) and a significant rate of resource management leaks (15.07% of its bugs). The model's strength—its focus on generating sophisticated code—is linked to its weakness.

The balanced predecessor [Claude 3.7 Sonnet]

This model represents a capable and well-rounded developer from a prior generation, exhibiting strong functional skills with a 72.46% benchmark pass rate. Its most defining personality trait is its communication style—it is an exceptional documentarian, producing code with a remarkable 16.4% comment density—nearly three times higher than its successor and the highest of any model evaluated. This makes its code uniquely readable and easier for human developers to understand.

But here's the catch with the balanced predecessor: while it appears more stable and less reckless than its more ambitious successor, it is by no means a "safe" model. It still introduces a high proportion of **'BLOCKER'** vulnerabilities (56.03%) and suffers from the same foundational flaws as the other models.

The efficient generalist [GPT-4o]

This LLM is a reliable, middle-of-the-road developer. Its style is not as verbose as the "senior architect" nor as concise as the "rapid prototyper"—it is a jack-of-all-trades, a common choice for general-purpose coding assistance. Its code is moderately complex and its functional performance is solid.

Its distinctive personality trait, however, is revealed in the type of mistakes it makes. While generally avoiding the most severe **'BLOCKER'** or **'CRITICAL'** bugs, it demonstrates a notable carelessness with logical precision. This is reinforced by its single most common bug category: **control-flow mistakes, which account for a remarkable 48.15% of all its bugs (refer to Table 4).**

This paints a picture of a coder who correctly grasps the main objective but often fumbles the details required to make the code robust. The code is likely to function for the intended scenario but will be plagued by persistent problems that compromise quality and reliability over time.

The unfulfilled promise [Llama 3.2 90B]

Given its scale and backing, this model represents what should be a top-tier contender, but its performance in our analysis suggests its promise is largely unfulfilled. Its functional skill is **mediocre**, with a pass rate of 61.47%, only marginally better than the much smaller open-source model we tested.

However, the model's most alarming characteristic is its remarkably poor security posture. The model exhibits a profound security blind spot, with an **alarming 70.73%** of the vulnerabilities it introduces being of **'BLOCKER' severity**—the highest proportion of any model evaluated. This security profile suggests that without an aggressive external verification layer, deploying this model in a production environment carries substantial risk.

The rapid prototyper [OpenCoder-8B]

This LLM is the brilliant but undisciplined junior developer, perfect for getting a concept off the ground with maximum speed. Its style is defined by conciseness, producing the least amount of code (120,288 LOC) to achieve functional results. This makes it an ideal choice for hackathons, proofs-of-concept, and rapid prototyping where time-to-first-result is the primary goal.

But, while the immediate productivity gain is obvious, it comes at the cost of the highest issue density, burying the project in technical debt that throttles long-term productivity and maintainability.

This model is a technical debt machine, exhibiting the highest issue density of all models at 32.45 issues per thousand lines of code. Its most prominent personality flaw is a notable tendency to leave behind dead, unused, and redundant code, which accounts for 42.74% of all its code smells.

This is a classic sign of rushed, iterative development without the discipline of cleanup. While perfect for a prototype, its code is a minefield of maintainability issues that would require a significant refactoring effort by a senior human developer or a robust governance tool, before it could be considered for production.

Why “more capable” can be riskier

Comparing newer models uncovers the most surprising finding of this analysis: a model “upgrade” can mask increases in real-world risk. The very process of making a model “more capable” can also make it more reckless, or shift the risk profile from common, well-understood flaws to more nuanced, and potentially harder-to-detect, implementation challenges.

As the table below illustrates, while the newer Claude Sonnet 4 shows a distinct improvement on performance benchmarks over its predecessor (+6.3%), this gain is paid for with a marked increase in the severity of its mistakes. The bugs and security vulnerabilities it creates are more likely to be of ‘BLOCKER’ severity.

The introduction of GPT-5 demonstrates a more complex trade-off. While it is more functionally correct than most of its peers and dramatically reduces the proportion of ‘BLOCKER’ severity vulnerabilities, it also introduces a new, more complex risk profile. As the model attempts more sophisticated solutions, it generates a much higher rate of advanced “Concurrency / Threading” bugs than any other model.

Table 9: The Evolving Risk Profile of More Capable Models

Metric	Claude 3.7 Sonnet (Older)	Claude Sonnet 4 (Newer)	GPT-5-minimal
Benchmark pass rate	72.46%	77.04%	75.37%
% of Vulnerabilities that are ‘BLOCKER’	56.03%	59.57%	**35.00%**
% of Bugs from ‘Concurrency/ Threading’	1.44%	9.81%	**20.00%**

This fundamental shift in the error profile means that the issues in lower-reasoning models are often easier to spot because they are more common and straightforward. The result is a new generation of models where increased capability shifts the risk profile from common, well-understood flaws to more nuanced, and potentially harder-to-detect, implementation challenges.

Conclusion: A new mandate for evaluating the leading LLMs

Functional performance benchmarks are a vital measure of an LLM's core problem-solving capabilities. Our findings are not intended to diminish these achievements, but to enrich them with additional context. As this report has shown, it is also important to study the crucial non-functional attributes—security, engineering discipline, and maintainability—that ultimately govern the total cost and risk of AI-assisted development.

This deeper analysis is revealing: all LLMs share common strengths and weaknesses, and each possesses a unique personality. The advent of new models like GPT-5 represents a fundamental turning point. They are not a silver bullet, but powerful new tools that come with a significant trade-off: they shift risk from common, well-understood flaws to more subtle, complex, and potentially dangerous ones.

Regardless of whether code is written by developers or an LLM, the "trust but verify" approach has never been more critical. For this new generation of models, verification must be even more rigorous. Developers must resist the false sense of security that comes from code that appears cleaner on the surface, while hiding systemic issues like concurrency bugs and severe technical debt underneath. As we accelerate into a world where most code is written with AI assistance, harnessing the power of these models responsibly requires expanding our view beyond the performance benchmark, to a richer, more nuanced view of the leading LLMs and their unique personalities.

Sonar

sonar.com