GitHub Copilot Suggests Everything It Can Generate

by Nick Clark | Published March 27, 2026 | PDF

GitHub Copilot transformed code assistance by providing inline suggestions that often anticipate what the developer intends to write. The integration into development workflows is seamless, and the quality of suggestions in well-understood domains is genuinely useful. But Copilot generates every suggestion it can produce regardless of whether the developer has demonstrated proficiency in the patterns being suggested or whether the codebase's architecture supports the approach being proposed. The assistant has no model of the developer's demonstrated capability and no mechanism to gate suggestions by complexity or risk. Skill gating provides this structural governance.

What GitHub built

Copilot's code generation is contextually aware, drawing from the current file, open tabs, repository structure, and the conversation context in chat mode. The system generates completions, explains code, fixes errors, writes tests, and refactors code based on natural language instructions. The workspace agent mode extends this to multi-file operations. The engineering to make AI code generation feel integrated into the development workflow is substantial.

Suggestions are generated based on what the model predicts is appropriate given the code context. A junior developer working with a complex concurrency pattern receives the same suggestion that a senior developer would. The suggestion may be correct code. Whether the developer understands the code well enough to maintain, debug, and extend it is a different question that Copilot does not address.

The gap between generation and appropriate suggestion

The structural limitation shows up in developer skill development. A junior developer who accepts Copilot suggestions for patterns they do not understand accumulates a codebase they cannot maintain. They have not learned the patterns because the assistant provided them without requiring demonstrated understanding. When the generated code produces a bug in a concurrency pattern the developer does not comprehend, the developer cannot diagnose or fix it without generating more suggestions they do not understand.

For enterprise codebases, the concern extends to architectural consistency. Copilot may suggest a pattern that is technically correct but inconsistent with the project's established architecture. A skill-gated system would match suggestion complexity to the developer's demonstrated proficiency in that pattern category and to the codebase's structural requirements.

Why code review is not skill gating

Code review catches problems after they are written. Skill gating prevents inappropriate suggestions from being offered. A developer who accepts a complex suggestion they do not understand, commits it, and receives review feedback has already lost the learning opportunity. The code exists, the reviewer explains the problem, and the developer fixes it based on the review. Skill gating would have offered a simpler approach first, unlocking the complex pattern only after the developer demonstrated understanding of the prerequisites.

The progressive unlocking model is particularly relevant for code assistants. New patterns are introduced at increasing complexity, with each level gated by demonstrated proficiency in the previous level. The developer who is building genuine capability in concurrent programming sees concurrent suggestions that match their current level. Advanced patterns become available as proficiency is demonstrated through correct usage, successful debugging, and independent application.

What skill gating enables for code assistants

With skill gating as a structural primitive, Copilot maintains a proficiency model for each developer across pattern categories: concurrency, error handling, data structures, API design, security patterns. Suggestions are gated to the developer's demonstrated level in each category. Evidence gates for advancing to the next level include correctly modifying suggested code, successfully debugging pattern-specific bugs, and independently applying patterns that were previously suggested.

Structural starvation ensures that patterns requiring undemonstrated proficiency are not suggested. The developer does not see the complex approach until they have demonstrated the simpler prerequisite. This produces developers who understand the code they write rather than developers who maintain code they accepted from an AI assistant.

The structural requirement

Copilot's code generation capability is real and growing. The structural gap is between what the model can generate and what the developer should receive. Skill gating provides proficiency-matched suggestion, progressive capability unlocking, evidence-based gates for advancing to complex patterns, and regression detection that identifies when developers are accepting suggestions beyond their demonstrated capability. The code assistant that develops the developer alongside the codebase produces better outcomes than one that generates everything it can.

LLM & Skill Gating All 21 steps →

The model proposes. The agent decides.

Primary Technical Disclosure
○ AI-Mediated Curriculum and Progressive Capability Unlocking Using Semantic Performance States
Secondary Technical
○ LLM as Structurally Untrusted Proposal Generator○ Mutation-Validation-Arbitration Pipeline○ Hallucination Prevention via Structural Starvation○ Trust Weight Calibration and Decay○ Evidence-Based Capability Gating○ Certification Token Generation○ Narrative State and Personality Architecture○ Skill Regression Detection and Capability Revocation○ Arbitration as Semantic Event○ Structural Starvation Composability○ Multi-Turn Memory Isolation○ Curriculum Engine Progressive Unlock○ Multimodal Evaluation Pipeline○ Multimodal Anti-Gaming Substrate○ Professional Skill Gating Applications○ Embodied Skill Gating○ Biological Identity Skill Binding○ Security and Drift Detection Layer○ Validation Feedback Asymmetry
Applications (General)
○ Enterprise AI Progressive Deployment Through Earned Capability○ Educational Platform Competency Through Structural Certification○ LLM and Skill Gating for Medical Licensing○ LLM and Skill Gating for Legal Practice Certification○ LLM and Skill Gating for Aviation Pilot Training Systems○ LLM and Skill Gating for Financial Advisor Certification○ LLM and Skill Gating for Cybersecurity Skill Progression○ LLM and Skill Gating for Manufacturing Quality Systems
Applications (Specific)
○ Duolingo's AI Unlocks Content, Not Capability○ Khanmigo Tutors Without Skill Gates○ Coursera Certifies Completion, Not Competence● GitHub Copilot Suggests Everything It Can Generate○ Pearson Assesses Knowledge Without Gating Capability Progression○ Chegg Provides Answers Without Gating Understanding○ Grammarly Corrects Writing Without Gating Writing Skill○ Photomath Solves Problems Without Building Problem-Solving Skill○ Century Tech Adapts Content Without Structural Skill Gates○ Squirrel AI Diagnoses Gaps Without Gating Progression
LLM & Skill Gating overview →
AQ

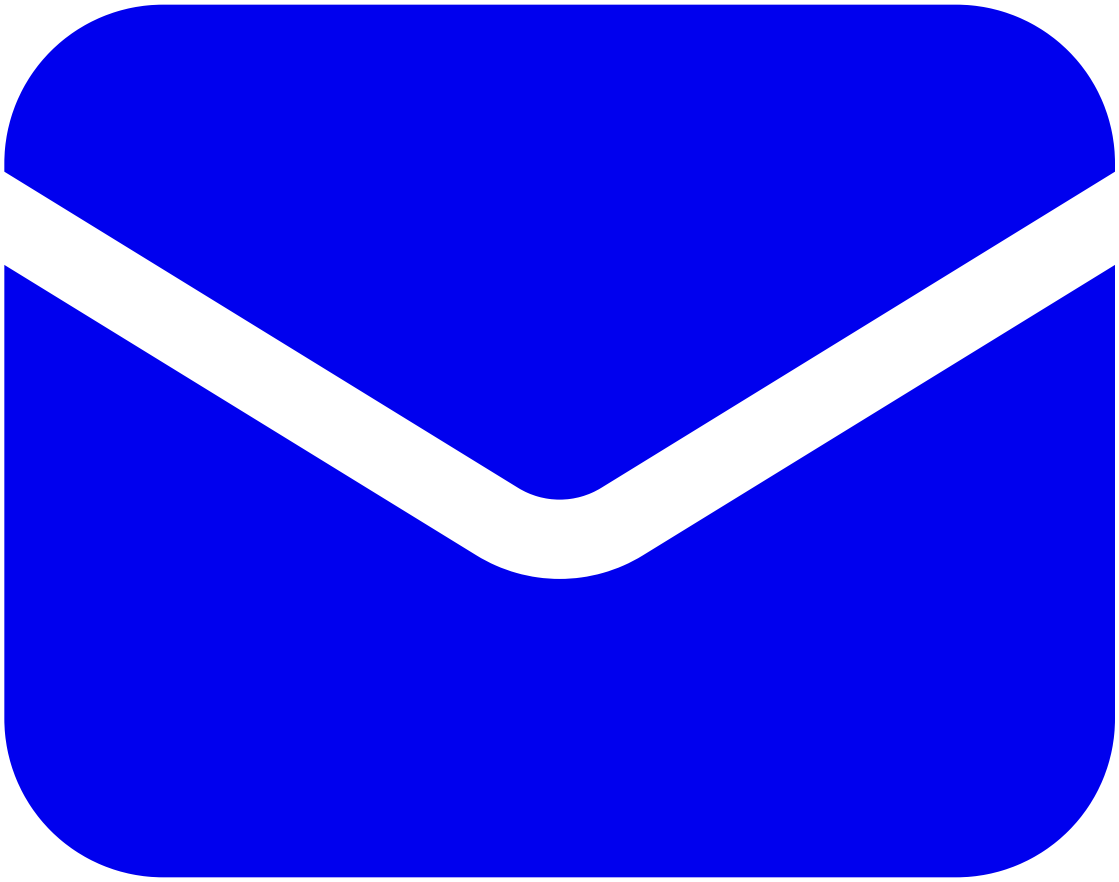
deterministic
autonomy

Legal

Subject to one or more pending U.S. and international patent applications, see Patents for the current list and status. No license, express or implied, is granted. Any use requires a separate written agreement—see Licensing. Patent applications referenced on this site are pending. Claim scope, if any, is subject to examination and may issue in altered form or not at all. See Legal for terms and conditions.

Last updated: 2026-03-03

- Opportunities
- Sitemap



-
- nick@qu3ry.net
- 72 28 14 36 01

Invented by Nick Clark | Founding Investors: Devin Wilkie