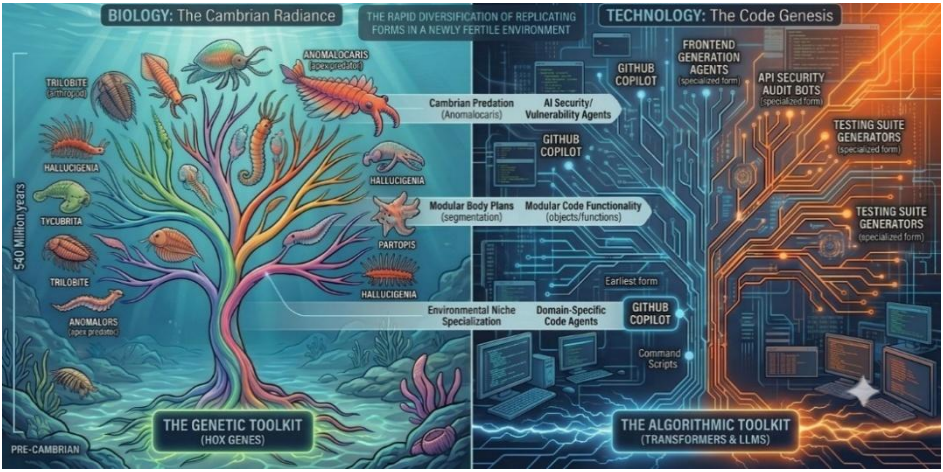
The CambrAIn Evolutionary Explosion



Paul Hampton likens the acceleration of Artificial Intelligence (AI) assisted software development tools to the Cambrian biological evolutionary explosion. Paul argues that the many tools on the market are not merely improving programming productivity, but changing the ecology of system creation itself. Paul examines market turbulence, the shift from in-editor assistance to persistent agents, rapid capability growth and the implications for safety-critical engineering.

An Analogy of Sorts

Over 500 million years ago, the Earth experienced an unprecedented and seismic change in the development of complex animal life called the Cambrian Explosion [1]. During this period, the fossil record shows a rapid expansion of complex animal life. Where previously, the primordial seas were home to soft-bodied species like jellyfish and sponges and other more basic lifeforms such as bacteria, the period saw the emergence of a vast array of new different species, and in fact, many of the major animal body plans represented today emerged during this period. A fundamental change was the development of hard mineralised skeletons and the earliest vertebrates – game changers in securing dominance in the evolutionary battlefield where trilobites and fish, amongst others, led the charge. And a battlefield it was, with new forms appearing, competing, specialising and disappearing.

I strongly believe that we are now experiencing an analogous explosion in the domain of AI assisted development environments, which seem to be emerging, multiplying and mutating with developers, enterprises, regulators and commercial markets acting as the selectors for their evolution. This will now drive a rapid expansion of applications through vastly accelerated development and the democratising of code development, where literally anyone, with a little technical setup, can now develop applications.

A New Dawn

In advance of my talk at the SCSC Seminar "How to Make the Most of AI: A Workshop for Safety Engineers" [2] in June this year, I decided to research AI code generation capabilities and started my journey by simply using ChatGPT, Claude and Gemini to review, fix and generate code. The results were remarkable with the tools *seemingly* displaying an intelligence about my objective and a guru level of language and software engineering knowledge. I was then introduced to Anysphere's Cursor [3] – an Integrated Development Environment (IDE), which took the capability to an entirely new level.

Workflows that once required weeks of scaffolding, experimentation, interface construction and error correction could now be compressed into a matter of days, and sometimes into a single sustained session. I could now write fully functional and tested applications with graphical interfaces in tens of minutes; mobile applications that might take at least 3 months to write, test and debug took a couple of days. Insights into previously difficult coding issues and problems could be resolved in a matter of seconds saving hours of head scratching.

A couple of workshops at the seminar demonstrated the capability. In timeframes measured in minutes, Cursor generated two working and tested applications including web-based user interfaces from pre-prepared single page specifications:

- A validator for Airbus A320 pitot airspeed sensors using a 2oo3 (two-out-of-three) software architectural pattern for three independent airspeed sensors deployed on the aircraft, providing safety-critical aircraft speed information to the flight control system
- A Monte Carlo simulation (see image opposite) to predict the safety margins of an Autonomous Train's braking system. This would run for the thousands of selected iterations taking a number of fixed values as well as assumed statistical distributions of other variable parameters



Monte Carlo simulation for a automated train braking system. Tested software generated by Cursor in minutes from 1 page spec.

The more capable tools now combine Large Language Models (LLM), repository indexing, command execution, test running, documentation search, code review and conversational planning. GitHub Copilot began as an assistant for completing lines and functions, but now spans chat, agent mode, issue to proposed code fix workflows, a command-line interface and integration with existing enterprise tools like Jira and Azure [4].

Cursor has gained attention because it places model interaction directly inside the editing loop, allowing codebase-aware changes, refactoring and explanation without constant context switching. Replit Agent, now in its fourth generation, extends the concept towards end-to-end application creation in a hosted environment [5]. A parallel body plan has become at least as important: terminal-native agents such as Anthropic's Claude Code and OpenAI's Codex, which live in the command line, run for extended periods, and can be wired into continuous integration.

A 2026 JetBrains survey of more than 15,000 professional developers found that 90% were using AI coding agents at work at least weekly, with 68% using them daily. Claude Code was the most widely adopted individual coding tool in the survey. [13].

The most remarkable aspect however, is the tools' (and I will use the word again) *seemingly* intelligent approach to fulfilling tasks. When given a project, Cursor will plan and logically break the problem down into manageable blocks and workflow and apply good engineering practices in designing the solutions and dealing with aspects that the user may not have thought of originally. The tool will then build the code, write tests and check results. When problems are reported it will *seemingly* intelligently hypothesise about possible causes, test these theories through ad-hoc code construction to arrive at the cause of issues and will then readily fix the problems if instructed.

"If you take the time to describe what you want (and don't want) in detail then there will be less 'schenaniganeering' as I might call it required at the end"

The process is extremely collaborative and iterative. Rarely will the initial outputs be exactly what you want, but the system will address any issues and new functionality can be easily added in a sentence or two.

In true Pareto style, you get a very long way very quickly initially and then there are what I might call "shenanigans": bending the prompts to get exactly what you want while stopping the AI undoing past successes. Also, dealing with all the new things you suddenly realised would be great to include – things you didn't know you wanted until you saw the end-to-end working executable.

In fact, how well outputs match your expectation boils down to how well you specify what you want and what you don't want (in Cursor you can define *rules* that all subsequent agents will adhere to, e.g. all code must comply with a given coding standard). If you take the time to describe what you want (and don't want) in detail then there will be less "schenaniganeering", as I might call it, required at the end.

I am finding now that the bottleneck is no longer the production of syntactically correct code, but the formulation of requirements, the detection of hallucinations, hidden assumptions, the management of dependencies, and the assurance that generated artefacts are fit for purpose. This last point is hugely important especially for safety-critical systems. I no longer need to write the code but I still need sufficient knowledge to understand the code that is generated.

From Watermill to Steam

The practical consequences are vastly accelerated productivity and the democratising of code development. I would liken it to the transition from water power to steam. Steam did not simply make mills faster; it changed where factories could be located, how labour was organised, and which businesses survived. AI in software development is having a similar systemic effect. It changes the economics of experimentation, reduces the penalty for discarding an approach, and enables small teams to attempt system-scale work previously reserved for larger organisations.

Rapidly developed prototypes are of course not certified products; acceleration at the front end does not alleviate the need for work in validation, security hardening, maintainability analysis and operational support, but the tools can help there too.



An old waterwheel: a casualty of the steam revolution. Now repurposed as a decorative landscape feature in Exmouth, Devon

In one of the workshops at the seminar in June, I asked, in the initial brief, for the IDE to write MISRA-C compliant code. I then asked it to review the generated code against MISRA-C and it found a number of violations – so as I said, it doesn't get things right first time. However, I then asked it to review the code against the specification and fix any issues, which it duly did. Similarly, security hardening if not specified at the outset will not be considered by the tool. However, once the code is written you can ask for a security review and it will report back a prioritised list of vulnerabilities, and will duly fix these if commanded.

An Evolving Tool Ecology

The current market resembles a biological radiation because many organisms are exploring adjacent niches. Some tools focus on individual programmers, others on enterprise governance, documentation, security or autonomous task completion. GitHub Copilot still benefits from deep integration with GitHub and Microsoft developer infrastructure, and now runs as an agent across editors, the GitHub workflow and the command line. Amazon Q Developer remains positioned around the Amazon Web Services ecosystem [6], while AWS has also introduced Kiro (which will replace Q Developer), a spec-first agentic environment that turns a prompt into requirements, design and tasks before code is written [14]. Google's Antigravity is an agent-first development environment, and the older Gemini CLI is being retired in its favour. Sourcegraph Cody, once a widely available assistant emphasising code search across large repositories, is now confined to enterprise deployments; the company's agentic successor, Amp, has been spun out as a separate organisation [7]. JetBrains, Atlassian and other established vendors are embedding assistants into familiar environments, and an emerging Agent Client Protocol is allowing different agents to inhabit the same editor shell.



Trilobites were not the largest Cambrian animals, but they were exceptionally well-adapted, enduring for over 270 million years

The niches continue to split. Cursor still competes by making the editor itself the centre of AI interaction, and has now launched Projects: a persistent coordinator that can direct large numbers of cloud and local agents over weeks of work, watch pull requests (request to merge code changes into a project's main repository) and continuous integration, and keep going after the laptop is closed [15]. Cognition's Devin, introduced as an AI software engineer intended to plan and execute multi-step development tasks, now also appears as Devin Desktop, the rebranded Windsurf editor, with an Agent Command Centre for supervising local and cloud agents [8]. Replit Agent 4 aims to turn natural-language ideas into deployable applications inside a shared cloud workspace. Qodo, formerly CodiumAI, has concentrated on test generation, code integrity and review assistance [9].

Poolside, Magic, Augment, OpenCode and others are pursuing large-scale coding models, open-source agents, context handling or enterprise automation. Some will become enduring platforms; others may become features inside larger ecosystems.

Survival will not be decided solely by model quality. Developers choose tools that fit habits, enterprises choose tools that satisfy governance, model suppliers choose which partners they will still serve, and safety-related organisations choose tools that leave auditable evidence. The fittest products may therefore be those that combine strong generation performance with traceability, data controls, integration into continuous integration and continuous delivery pipelines, and a credible story about intellectual property. In an ecology, the largest animal is not always the most resilient. The better adapted one usually is.

Extinction and Adaptation

The market has already produced fossils, and it is still producing them. Kite, once a prominent AI coding assistant, shut down in 2022 after concluding that the business could not be sustained despite years of technical effort [10]. DeepCode did not disappear in the same way, but was absorbed into Snyk, where its Machine Learning (ML) based code analysis became part of a broader security platform [11]. As mentioned, CodiumAI rebranded as Qodo, reflecting a shift from a narrow association with generated tests towards a wider code quality proposition. Amazon CodeWhisperer was folded into Amazon Q Developer, demonstrating how an assistant can be repositioned inside a larger enterprise AI brand and even this is now being replaced by Kiro. SpaceX's acquisition of Cursor was followed within a fortnight by OpenAI's decision to end managed access to its models inside that editor, with a proposed cut-off in November [16]. The selection pressure has not eased; it has intensified.



"These examples show that failure, acquisition and rebranding are not peripheral events. They are the selection mechanisms by which the field discovers viable forms"

These examples show that failure, acquisition and rebranding are not peripheral events. They are the selection mechanisms by which the field discovers viable forms. A tool may be technically impressive but commercially weak. Another may be less glamorous yet persist because it serves a regulated enterprise need. A tool may vanish as a product because its capability becomes a standard feature in a dominant platform, or because a supplier relationship breaks. This is exactly what one would expect during an early radiation: boundaries between species are unstable, and some organisms are transitional. Arguably, this might be classed as normal market forces, but it's the speed of these processes that is unprecedented.

The stronger candidates for enduring success currently appear to share three characteristics. First, they possess distribution through existing developer workflows, as GitHub, JetBrains and the Unix command line do. Secondly, they control or integrate with execution environments, as Replit, Amazon, cloud agent sandboxes and terminal agents do. Thirdly, they address repository-scale, persistent context rather than isolated snippets, as Cursor Projects, Claude Code and the remaining enterprise code-intelligence platforms attempt to do. None of these advantages is decisive, and today's apparent winners may still be displaced. However, these characteristics may prove important to long-term durability.

Recursive Acceleration

One consequential feature of the CambrAIn explosion is that AI code generation is being used to improve AI code generation itself and this is spanning virtually every tier of the modern AI development stack. In my experience (and for the many projects I've attempted), as long as the intent is well-specified, the tools I've used write far better and robust code than I could ever write and of course, at an incredibly fast rate. They also identify issues that I've not previously thought of, leading ultimately to higher quality code. It seems inevitable that better and more capable AI code generators will also evolve rapidly.

This recursive loop compresses the time between an underlying model improvement and its appearance in a practical development workflow. Newsletters such as TLDR AI show the resulting cadence: model releases, agent demonstrations, funding announcements, benchmark claims and product launches arrive weekly, not yearly [12]. The capability growth is already highly visible, and it has not stopped simply at larger context windows allowing reasoning over larger portions of a repository:

- Retrieval augmented generation has improved the relevance of answers by grounding them in project-specific files and documentation
- Agents can now run commands, inspect test failures and attempt repairs
- Code review assistants can suggest changes aligned with style rules
- Migration tools can transform older programming language versions or cloud service interfaces
- Terminal-native agents can run for hours, spawn sub-agents and sit inside continuous integration pipelines
- Persistent coordinators can retain project context for weeks, delegate to cloud agents, and act on signals from chat, pull requests and build failures
- Plugin and skills marketplaces let agents extend themselves, with all the supply-chain consequences that implies

This recursive acceleration also creates hazards. Benchmark-driven development can reward superficial success. Agents may learn to satisfy tests without satisfying intent. Generated code can propagate insecure "anti-patterns" at scale. Dependencies may be introduced without adequate supply-chain review. The same speed that enables exploration can multiply latent defects. Therefore, the central engineering challenge is to couple generative acceleration to verification acceleration. If the former outruns the latter, organisations will accumulate technical debt faster than they can understand it. If the understanding does not come along with the productivity, then we have a new debt: a comprehension debt. This is however not quite as bleak as it sounds as AI code generators are rather good at documenting and summarising what has been done.

There is also growing empirical literature, and it is less comforting than the marketing. A randomised trial by METR of experienced open-source developers using early-2025 tools found that they took around 19% longer when AI was allowed, even though they believed they had been sped up [17]. Later waves suggest that more agentic tools may now be accelerating some work, but the measurement is messy. My own experience remains one of dramatic speed-up on greenfield applications and workshop demonstrations. Perhaps both things can be true: new systems appear in minutes, while changes to large, high-assurance codebases can still be slowed by review, hallucination-checking and the cost of understanding what the agent just did.

From Coding to Systems Engineering

The deepest change is not that code appears faster. It is that the boundary between programming and systems engineering is moving. When an assistant can generate a database schema, application programming interface, user interface, deployment file and initial test suite from a description, the description itself becomes the critical artefact. Spec-first environments now make that explicit, writing requirements, design and task lists before they write code. Ambiguity in intent is converted into executable ambiguity. This is valuable for discovery, but dangerous where requirements are incomplete, inconsistent or safety significant.

A disciplined response is to treat AI assisted development as a transformation pipeline. Requirements, assumptions and hazards enter the pipeline; code, tests, documentation and deployment scripts leave it. The pipeline must be instrumented. Prompts, spec files, coordinator logs and plugin hashes should be preserved where they are substantively equivalent to design decisions. Model versions should be recorded. Generated changes should be reviewed against explicit acceptance criteria. Tests should not be accepted merely because the same assistant proposed them. Configuration files, infrastructure definitions and security policies deserve the same scrutiny as application code.

This viewpoint also changes education. The valuable engineer is not the person who can generate content fastest, but the person who can interrogate an output, detect a missing constraint, design a meaningful test, and recognise when a plausible explanation is wrong. Reading code becomes at least as important as writing it. Architectural judgement becomes more important, not less, because low-level implementation cost no longer dictates design choices as strongly as it once did. In short, AI is a "signal amplifier".

- Bring clear intent, strong architecture, disciplined review and it will help immensely
- Bring vague requirements, weak standards and trust without verification then disaster looms

Implications for Safety-Critical Practice

AI generated code can be used in safety-related developments today if it is subjected to the same, or stronger, assurance activities as human-produced code. Standards such as DO-178C in aviation and IEC 61508 for functional safety do not grant code moral status because a human typed it; they require evidence that appropriate processes, verification and independence have been applied. Some practitioners, however, read IEC 61508-3 as actively discouraging AI in the safety path, and current language models sit poorly with classical tool qualification: they are statistical, prompt-sensitive and not consistently reproducible. Tool qualification guidance, including the RTCA/DO-330 framework associated with RTCA/DO-178C, is also relevant when a tool could introduce or fail to detect errors, but qualifying an agent that writes, runs and patches code is a harder problem than qualifying a compiler.

The practical rule is straightforward: generated code is untrusted until justified. Human review remains essential, particularly for requirements interpretation, hazard controls and architectural decisions. Static analysis, formal methods where applicable, unit tests, integration tests, requirements traceability, coverage analysis and independent verification should be strengthened rather than relaxed.

Safety cases should explicitly state where AI was used, what controls were applied, and why residual risk is acceptable. The use of AI to generate part, or all, of the Safety case also needs to be addressed to avoid a rabbit hole of circular arguments. Organisations should also consider data governance, because sending proprietary or safety-sensitive code to external services may create confidentiality and configuration management concerns. Cloud agents, terminal agents and coordinators that watch workplace communication and team collaboration platforms or continuous integration widen that exposure. Reproducibility becomes a key consideration: will the same prompt generate the same or different code next week?

Perhaps the biggest challenges are loss of skills and "review fatigue": who wants to spend the rest of their career reviewing AI outputs?

Human factors considerations also need to be part of the mix. There are risks of automation bias, over-trust and complacency. Perhaps the biggest challenges are loss of skills and "review fatigue": who wants to spend the rest of their career reviewing AI outputs? (A theme further explored by Ben Fulford in his article).

There is, however, a developing practice in which checking is performed by a council of AI systems. Pull-request review bots, competing critic agents and multi-model checking are already shipping. Different models, tools and analysis engines can criticise one another's outputs, search for counterexamples, generate adversarial tests, compare requirements interpretations and produce evidence for human assessors. Such councils may eventually exceed individual human reviewers in breadth, patience and pattern recognition. Even then, safety responsibility will not disappear. It will move towards designing the council, validating its independence, understanding correlated failure modes, and deciding when its evidence is sufficient. Multiple AI reviewers do not necessarily provide independent assurance. Models trained on similar datasets or using similar architectures may exhibit correlated failure modes, and so, apparent agreement should not be mistaken for independent evidence.

The tools themselves have also become part of the attack surface. In September 2026 researchers disclosed a cluster of separate agent-hijacking flaws, including GitSpawn, in which a repository's local Git configuration could take over a coding agent, and Plugin4Shell, a zero-click remote code execution chain that bypassed plugin SHA-hashing in agent marketplaces [18].

Some vendors patched the affected implementations quickly. Other affected products were already being retired or had not yet received a corresponding client-side fix. For a safety organisation this is not a curious software-supply-chain anecdote. The coding agent now sits inside the trusted computing base: it can read the repository, run commands, install dependencies and, in some configurations, act on external signals without a human in the loop. Permitted-tool policies therefore need to cover plugin sources, auto-update, Model Context Protocol (MCP) servers[1], and whether a cloud agent is allowed to watch chat channels or build pipelines.

[1] An MCP is a standard-based bridge that mediates and controls access between AI systems and your local filesystems, databases, enterprise tools (like GitHub, Jira, or Slack), and internal enterprise APIs.

Conclusions

The Cambrian Evolutionary Explosion did not produce a finished animal kingdom overnight. It produced an extraordinary burst of experimentation: new forms appeared, ecological niches opened, competition intensified, and many evolutionary experiments disappeared while others succeeded. The resulting ecosystem was fundamentally different.

The CambrAIn Explosion may prove similar. AI-assisted development is moving from tools that help programmers write code towards agents that can interpret specifications, plan work, create and modify software, run tests and operate for long periods. This isn't just about making software faster – it is changing how development is structured and funded. As a result, small teams can now build and maintain complex systems that used to require massive engineering budgets. But speed of generation is not the same as speed of assurance. For safety-critical engineering, the central challenge is whether our ability to understand, verify, validate and govern what is generated can keep pace with accelerating production.

This changes the role of the engineer. Skills include specifying intent, identifying hazards and hidden assumptions, challenging plausible, incorrect outputs, designing verification and determining when evidence is sufficient. AI may reduce the amount of code engineers write, but it does not reduce their responsibility for the systems that code creates.

AI may also provide new assurance mechanisms: automated analysis, adversarial testing, competing verification agents, formal reasoning and continuous evidence generation. But these mechanisms must themselves be understood and controlled. Multiple AI systems agreeing with one another do not necessarily provide independent evidence. Our objective should not be to stop the CambrAIn Explosion, nor to accept it uncritically. It should be to ensure that assurance evolves at least as quickly as generation. The opportunity is enormous. So is the responsibility.

What should SCSC members do?

The SCSC community should engage with the CambrAIn Explosion early, critically and practically. Waiting for the market to stabilise may feel prudent, but it risks leaving assurance practice several years behind development practice. Conversely, adopting tools without disciplined governance would confuse productivity with dependability. The correct suggested approach is experimental conservatism: learn quickly, deploy carefully, and collect evidence. Organisations should run pilots on non-critical codebases, measure defect patterns, compare review effort, and identify which tasks are genuinely accelerated. They should also update development standards so that AI use is visible rather than informal. Three actions seem particularly urgent:

- Safety teams should create policies for permitted tools, data handling, prompt retention, model version recording, plugin sources, auto-update, and the use of cloud or terminal agents on safety-related repositories
- Assurance teams should develop review checklists for generated artefacts, including tests, documentation, spec files and coordinator logs
- Professional bodies should share incident reports, near misses and good practice patterns involving AI assisted development. Studies such as METR's are a start, but the field still needs empirical evidence from safety-related work, not anecdotes alone

References

[1] Wikipedia contributors, "Cambrian explosion", https://en.wikipedia.org/wiki/Cambrian_explosion, accessed September 2026.

[2] SCSC Seminar: How to Make the Most of AI: A Workshop for Safety Engineers, 25th June 2026, scsc.uk/e5101

[3] Anysphere, "Cursor: The AI Code Editor", https://www.cursor.com, accessed September 2026.

[4] GitHub, "GitHub Copilot", https://github.com/features/copilot, accessed September 2026.

[5] Replit, "Introducing Replit Agent 4: Built for Creativity", https://replit.com/blog/introducing-agent-4-built-for-creativity, accessed September 2026.

[6] Amazon Web Services, "Amazon Q Developer", https://aws.amazon.com/q/developer, accessed September 2026.

[7] Sourcegraph, "Cody FAQs", https://sourcegraph.com/docs/cody/faq, accessed September 2026.

[8] Cognition, "Introducing Devin Desktop", https://cognition.com/blog/introducing-devin-desktop, accessed September 2026.

[9] Qodo, "CodiumAI is now Qodo", https://www.qodo.ai/blog/introducing-qodo-a-new-name-the-same-commitment-to-quality, accessed September 2026.

[10] Kite, "Kite is saying farewell", https://web.archive.org/web/20221117130927/https://www.kite.com/blog/product/kite-is-saying-farewell/, accessed September 2026.

[11] Snyk, "Snyk acquires DeepCode", https://snyk.io/news/snyk-acquires-deepcode/, accessed September 2026.

[12] TLDR, "TLDR AI archives", https://tldr.tech/ai/archives, accessed September 2026.

[13] JetBrains, "AI Coding Agents: Adoption Trends", https://blog.jetbrains.com/research/2026/08/ai-coding-agent-adoption-2026/, accessed September 2026.

[14] Kiro, "Kiro: Move beyond AI coding to agentic engineering", https://kiro.dev/, accessed September 2026.

[15] Cursor, "Introducing Projects", https://cursor.com/blog/projects, accessed September 2026.

[16] OpenAI, "Our decision on Cursor following its acquisition by SpaceX", https://openai.com/index/our-decision-on-cursor-following-its-acquisition-by-spacex/, accessed September 2026.

[17] METR, "Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity", https://metr.org/blog/2025-07-10-early-2025-ai-experienced-os-dev-study/, accessed September 2026.

[18] Lyons, J., "AI coding agents' 0-click RCE flaw could hand attackers keys to the kingdom", The Register, 17 September 2026, https://www.theregister.com/security/2026/09/17/ai-coding-agents-0-click-rce-flaw-could-hand-attackers-keys-to-the-kingdom/5297335, accessed September 2026.

Image attribution:
top image: AI-generated (Gemini), The Cursor code editor logo: Courtesy of Anysphere, Inc., trilobite fossil: ID 75353597 | Cambrian © Eugen Thome | Dreamstime.com, Snyk and DeepCode integration graphic: Courtesy of Snyk Ltd.

Paul Hampton, Timegalore Ltd.



Paul is a Chartered Engineer with over 35 years' experience in IT. He has spent 20 of those designing and developing enterprise systems in sectors such as Energy & Utilities, Government, Criminal Justice and Health and 15 years in System Safety in a variety of capacities, including safety engineering, safety management, independent auditing and corporate governance and assurance. He has published a number of articles and papers and is a frequent presenter at industry seminars and symposiums. In 2019 Paul joined the SCSC Steering Group and Executive Committee and is the SCSC Newsletter Editor and Web Master.