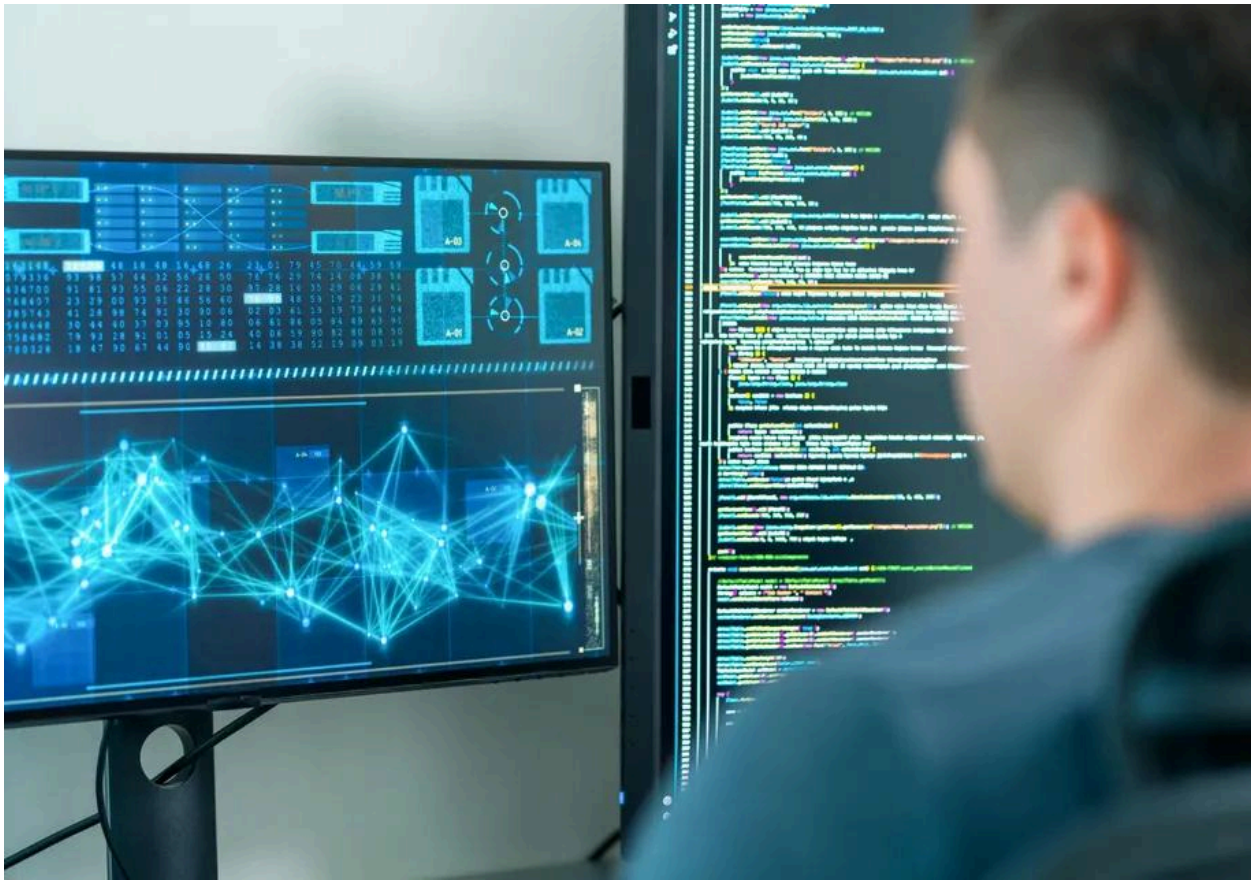
INNOVATION

Generative AI Is Just The Latest Chapter: 80 Years Of Programming Automation

Boris Kontsevoi Forbes Councils Member
Forbes Technology Council COUNCIL POST

Sep 08, 2026, 08:15am EDT

Boris Kontsevoi is a technology executive, President and CEO of Intetics Inc., a global software engineering and data processing company.



GETTY

Much of the current excitement around AI coding rests on convenient fiction: that machines have only just begun doing programmers' work. They have been doing it for more than 80 years. Generative AI is just the latest and most visible chapter in that long history of programming automation.

In the 1940s, programmers worked close to the hardware, entering numeric instructions and managing memory addresses directly. Assembly language changed that relationship. Developers could write symbolic names instead of raw numbers, and an assembler translated them into machine instructions. It remained machine-specific, but part of the translation had moved from the programmer to software.

Compilers extended that idea from individual instructions to entire programs. Formula Translation (Fortran), introduced in the 1950s, let scientists and engineers describe calculations in a language closer to mathematics, while the compiler produced the machine instructions. Its creators expected coding and debugging effort to fall to less than one-fifth, and early use bore that out. Common Business-Oriented Language (COBOL) applied the same principle to business data processing, making programs more readable and portable.

Once languages became easier to write, programmers faced the next inefficiency: rebuilding routines that already existed.

By the late 1960s, the idea of software as standardized, reusable parts was already taking shape. In the early 1970s, libraries made it practical. The Numerical Algorithms Group (NAG) and International Mathematical and Statistical Libraries (IMSL) supplied mathematical and statistical routines. EISPACK and LINPACK packaged complex matrix operations into callable Fortran subroutines. These packages allowed developers to build on tested components instead of starting from scratch. Unix applied the same principle at the program level, connecting small utilities through pipes to perform larger tasks.

In the 1980s, reuse moved into application development. Class libraries and graphical frameworks supplied windows, menus, text editors, data structures and interaction patterns that developers could reuse and adapt. Smalltalk-80 helped establish this model. C++ frameworks such as ET++ carried it further and reported source-code reductions of 80% or more compared with conventional graphical toolboxes. Computer-Aided Software Engineering (CASE) tools, report generators and database forms then applied the same logic to application structure, producing more of it automatically.

Fourth-generation languages, or 4GLs, took automation directly into business

software. Developers described the query, form or report they needed, and the platform handled more of the repetitive implementation. Structured Query Language, or SQL, applied the same principle to data retrieval. Informix 4GL and Oracle Forms packaged recurring work around data entry and reporting, while PowerBuilder's DataWindow handled much of the cycle of retrieving, displaying, validating and updating business data.

These platforms became common foundations for internal systems, shifting effort from repetitive data handling toward business rules, integration and process design.

The 1990s brought this model into everyday application development. Visual Basic, Delphi, PowerBuilder, Oracle Forms and other rapid application development environments let developers assemble interfaces, connect data and define events visually. The "controls" in these systems were real software components: grids, charts, buttons and specialized widgets that could be dropped into applications and reused. By 1996, Microsoft's ecosystem already offered more than 2,000 ActiveX controls from hundreds of independent vendors.

Reusable controls were only part of the 1990s automation wave. Integrated Development Environments (IDEs) also generated the application around them. Visual J++ 6.0 could create a single-table or master-detail database program, while Microsoft's foundation class library AppWizard produced the source, header, resource and project files for a Windows application. Developers could start from a generated application structure rather than construct it manually.

Their output was limited to structures predefined by templates and selected options. Generative AI did not invent code generation. It changed what developers could use as input.

By the 2000s, software development depended as much on reuse as on original construction. Open-source libraries, package managers, frameworks, application programming interfaces (APIs) and Stack Overflow made searching, selecting and adapting existing work part of development. Once Stack Overflow measured more than 40 million copy actions over two weeks in 2021. By 2025, GitHub reported more than 180 million developers and nearly 1 billion commits. Modern software was already built on shared code and accumulated knowledge at scale.

Cloud services, DevOps and low-code platforms carried the pattern beyond code throughout the 2010s. Teams stopped building every infrastructure capability internally. Compute, storage, identity, payments, messaging, monitoring and deployment became services or automated pipelines. The programmer's work shifted again, this time toward composition, configuration and integration.

Generative AI changes the relationship between intent and implementation. Earlier generators required a formal model, a template or a sequence of choices inside an IDE. Today, a developer can describe a requirement, identify an error or point the system to an existing codebase. Using that context, it can propose code, tests, documentation or a refactor. Stack Overflow's 2025 survey showed that 84% of respondents were already using AI tools or planned to do so.

What makes this stage different is the range of work it brings together. Search, code completion, generation, explanation, testing and refactoring can now happen within the same workflow. This can shorten the distance between a business requirement and an implementation, while leaving teams more time for design, integration and product decisions.

The more important shift is continuity across tasks. Previous tools generated a form, completed a line or transformed a predefined model. Generative AI can stay with the same problem, from interpreting the requirement through implementation, testing and documentation.

From the Theory of Constraints perspective, its real place is wherever it can relieve the bottleneck limiting the overall delivery process. If that constraint lies in requirements, testing, security or integration, accelerating code generation alone will not improve overall throughput.

The real advantage will come from how AI is built into the software delivery model, not from generated output alone. Its value becomes visible when delivery accelerates and engineering capacity expands without weakening architecture, quality standards or ownership.

Boris Kontsevoi

President & CEO of Intetics Inc.

Naples, Florida Area

Forbes | Councils
Member since 2018

Boris Kontsevoi is a founder and President of Intetics Inc., a leading global software engineering and digital transformation company. Under his leadership, a group of software engineers developed into a truly global technology company with multiple professional certifications, including ISO/IEC 42001 certification, for AI Management Systems, and industry awards, including the Global Outsourcing 100, Software 500, and Global Sourcing Association best of class company.

intetics
Where software concepts come alive™

INTETICS MEANS YOUR SUCCESS

Toll Free: +1 (877) SOFTDEV

US: +1 (239) 217-4907

DE: +49 (211) 3878-9350

UK: +44 (20) 3514-1416

www.intetics.com