Preprints.org

Review

Not peer-reviewed version

A Systematic View on AI-Based Code Generation

Elena V. Ravve *

Posted Date: 17 November 2026

Keywords: artificial intelligence; automated code generation and maintenance; large language model; graph neural network; test driven design; distances between codes; descriptive complexity theory

Preprints.org is a free multidisciplinary platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC, OpenAlex.

Copyright: This open access article is published under a Creative Commons CC BY 4.0 license, which permit the free download, distribution, and reuse, provided that the author and preprint are cited in any reuse.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.

Review

A Systematic View on AI-Based Code Generation

Elena V. Ravve

Software Engineering Department, Braude Academic College of Engineering, Snunit St. 51, Karmiel, 2161002, Israel; cselena@braude.ac.il

Abstract

In this review, we consider and combine the most recent results from different fields of computer science and software engineering, related to automated code generation using AI-based tools. First, we provide an historical retrospective in order to establish similarity between industrial revolution and our days. This approach allows adaptation of the historical experience to our days challenges. Next, we provide some results from classical descriptive complexity theory relevant to the scope of our review. In addition, we consider two approaches from Artificial Intelligence (AI): Large Language Model (LLM) and Graph Neural Network (GNN); and the recent results, which provide severe limitations on computability and complexity of both of them. Then, we emphasize the crucial role of automated translations from natural languages to formal languages, used in the previously mentioned results. For our best knowledge, such translations do not exist as for now. However, we show a list of results from different branches of science, which provide some evidence that they should be doable. Moreover, we point out the importance of Test Driven Design (TDD) in automated code generation using AI-based tools. Code maintenance is not less important than code generation from scratch. That is why, any change, made in code must be as local, as possible. The last fact requires precise definition of distances between versions of the desired code at each TDD cycle. Finally, we define a complete workflow, which combines all the features: results from descriptive theory, translation from natural languages to formal languages, code distances as well as combination of TDD, LLM and GNN altogether, that is new. The contribution is dedicated to software engineers and programmers with limited knowledge in computer science and especially in descriptive complexity theory as well as computer science experts with limited experience in practical code writing. The review is based on classical results of computer science as well as on the most recent contributions, some on them even did not pass a complete review process and official publication in scientific papers or proceedings. We also used personal communications with the relevant researches. Some of them are explicitly maintained in the acknowledgement section. We included particular sources to the review according to our understanding to their importance to the purpose and the framework of our paper.

Keywords: artificial intelligence; automated code generation and maintenance; large language model; graph neural network; test driven design; distances between codes; descriptive complexity theory

1. Historical Retrospective

1.1. Industrial Revolution: The First Computer and the First She Programmer

In our story, we skip the simplest manual instruments like the abacus, used for calculation since ancient times. We only mention French mathematician Blaise Pascal [1], who in 1642 invented a rotating-wheel calculator and Gottfried Wilhelm Leibniz [2], who later expanded the device to handle multiplication and division. Hi also developed formal logic in early 1700's, which became the theoretical basis of all computer science.

[1] A French mathematician, physicist, inventor, philosopher, and Catholic writer (1623-1662).
[2] A German mathematician, philosopher, scientist, and diplomat (1646-1716).

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

We jump directly to the beginning of the era of industrial revolution. The first changes from hand work on local farms to machine work in large factories started in Great Britain around 1760's with massive use of new steam engines and effective methods to make cloth. The corresponding engineering (calculation) tools were needed urgently.

In 1792 the French government declared creation of the Cadastre, a set of logarithmic and trigonometric tables, and Gaspard de Prony [3] was appointed to supervise the mission. 18 volumes of the tables were prepared manually and never published. In general, in this period, the notion of a "computer" was used for a person, who manually composed the mathematical tables for different purposes.

In 1819, Charles Babbage [4] found the unpublished tables. In July 1822, he sent a letter to the Royal Society proposing to replace the "computer" by a mechanical device that should be able to do the same job. In the same 1822, Charles Babbage presented his *Difference Engine*, see Figure 1, as a fixed mechanical sequence, which is designed to produce table of polynomial functions. It means that this first device was *not programmable*: it could not read instructions and just performed a pre-defined set of additions and subtractions.

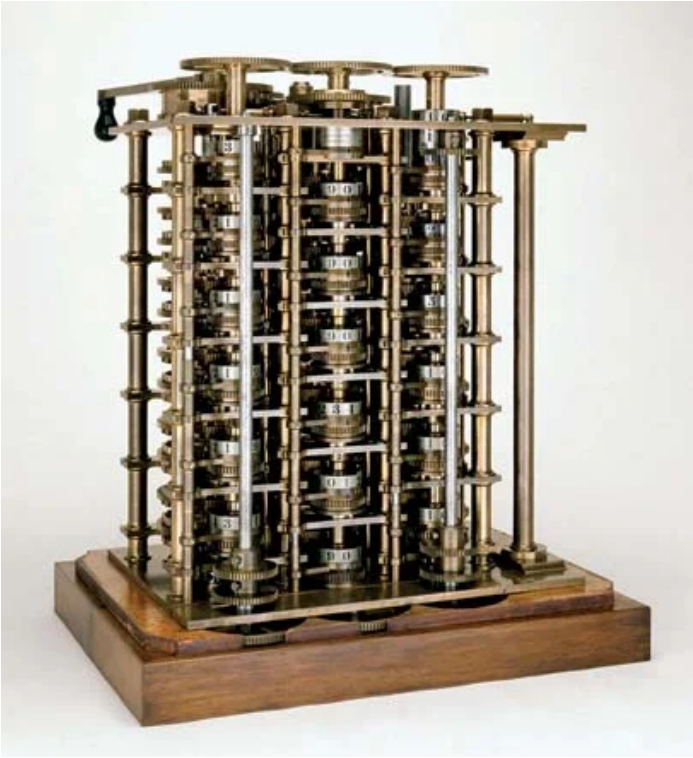


Figure 1. Difference Engine, [1].

The input was set manually into the machine's mechanical columns and consisted of the initial numerical values and parameters. Then, the operator turned a hand crank to start the calculation. The output of the calculated mathematical values was displayed on mechanical columns, printed on paper, or stamped into soft metal plates.

Few years later, in 1837, Charles Babbage designed his still mechanical *Analytical Engine*, see Figure 3, that is in fact the first general-purpose *programmable* computer in the sense as we use the notion now, based on arithmetical and logical unit operations and local memory. In this case, the

[3] A French mathematician and engineer, who worked on hydraulics and produced mathematical tables and texts on mathematical physics (1755-1839).

[4] An English mathematician, philosopher, inventor and mechanical engineer (1791-1871).

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

input was already provided using manually punched cards, inspired by Joseph-Marie Jacquard [5] loom, Figure 2. The punched cards in fact provided code and data presented as 0's and absence of 0's.



Figure 2. Jacquard Loom, [2].

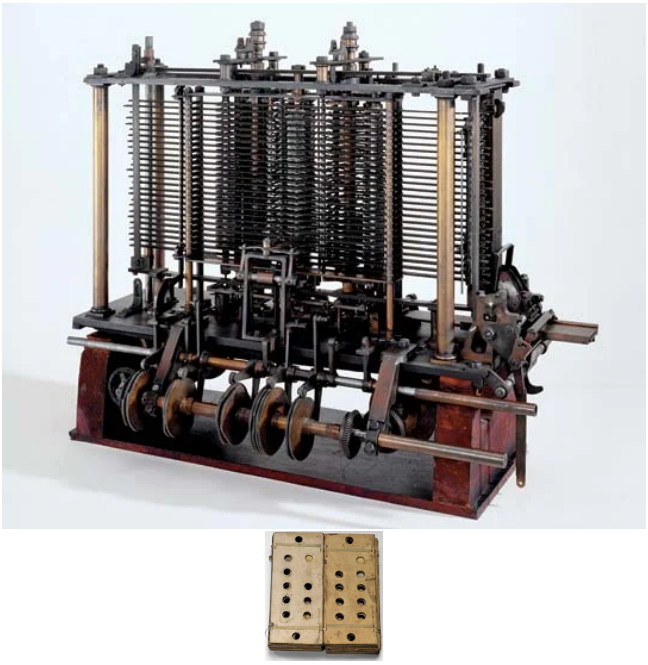


Figure 3. Analytical Engine [3] and manually punched cards.

A mechanical input device read the holes on the punched cards. The read information was then transferred and stored in the mechanical memory. The stored information was then transferred to a mechanical processing unit to perform the coded instructions on the corresponding values. The

[5] A French inventor of the Jacquard loom, which served as the impetus for the technological revolution of the textile industry (1752-1834).

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

calculated mathematical results were presented on a built-in printer or a curve plotter, or stored in punched cards or pressed metal stereotype.

In this way, any arithmetic or complex mathematical calculation may be done including but not limited to tabulated and evaluated equations up to seventh-order polynomials or higher; repetitive mathematical sequences and approximations, square roots, complex mathematical sequences including Bernoulli numbers and Gaussian elimination. Specialized punch cards could be used in order to repeat computations or alter execution paths based on intermediate results.

Ada Lovelace, [4], is the world's first computer programmer, who wrote an *algorithm* [6] to calculate Bernoulli numbers, which was then codded in the corresponding way to be executed on the Analytical Engine.

Due the mechanical nature of the implementation it suffered of multiple practical limits such as: very small size of the internal memory, very long computational time, lack of precision and so on. That is why any question about efficiency of the computation was not relevant. Moreover, even correctness of the coding was not an issue. However, the most important problem, mentioned already by Ada Lovelace is:

Result 1. *The Analytical Engine has no ability to originate or create independent ideas; it could only execute the exact instructions given to it.*

It means that the computation should be invented by a human being as an algorithm and only then codded in the suitable way.

To summarise the section; from now, the notions "computer" is not used for a person, who makes calculations, but rather for a device that makes the calculations. We presented two such mechanical devices - one non-programmable, the second one - programmable. For the Analytical Engine we observe a well defined flow: from a problem to be solved via invention of the corresponding algorithm and its codding (programming or in general software engineering) to execution of the codding on the corresponding engine that should provide the final results (hardware part of the solution), see Figure 4.

[6] A step-by-step set of rules or instructions used to solve a problem. The word "algorithm" comes from Latinization of al-Khwarizmi. Muhammad ibn Musa al-Khwarizmi was a 9th-century Persian mathematician and scholar from Baghdad. He wrote a book around 825 CE that explained how to use Hindu-Arabic numerals (0 through 9).

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

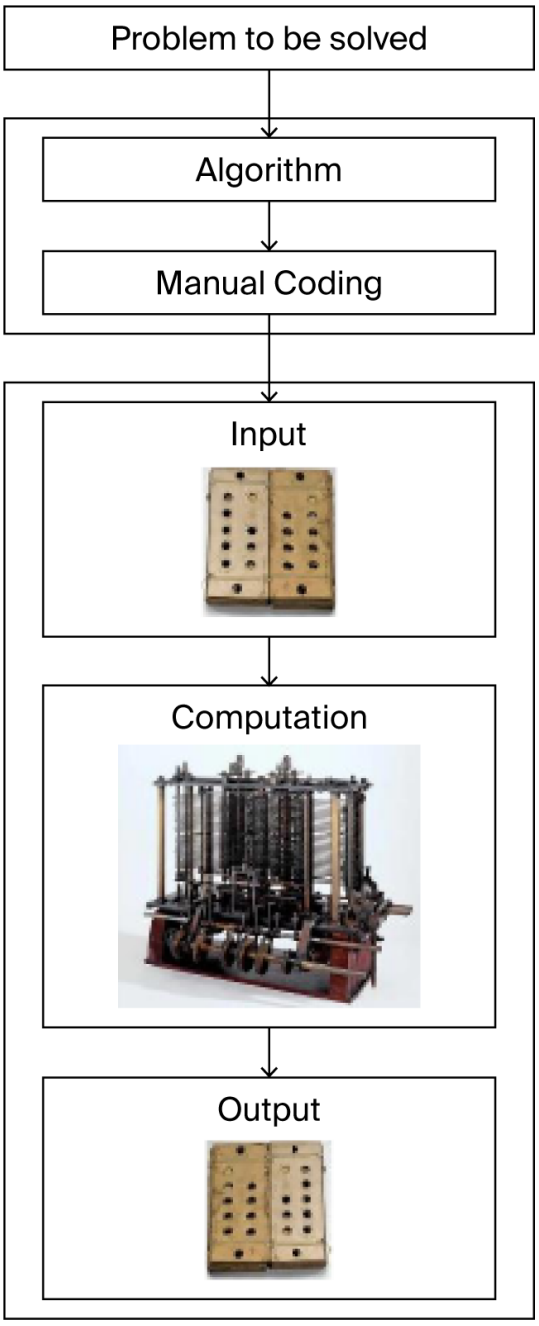


Figure 4. Analytical Engine: from a problem to its solution.

The main practical limitations came mostly from the mechanical nature of the available implementation. So, neither question about complexity or even correctness of the codding were not, in fact, relevant. However, even for these devices the question about their theoretical computational power already erased.

1.2. Modern History: Massive Use of Computers, Programmer as a Profession

The practical limitations of the first computational engines became to be less relevant when the electricity came to the everyday use in late 1870's and the 1880's. In 1882, Thomas Alva Edison [7] opened the Pearl Street Station in New York City. Then, Nikola Tesla [8] and George Westinghouse [9] introduced practical use of alternating current systems. Eventually, in 1930's-1950's: governments of

[7] An American inventor and businessman (1847-1931).
[8] A Serbian-American engineer, futurist, and inventor (1856-1943).
[9] An American inventor, engineer, and entrepreneurial industrialist (1846-1914).

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

all developed countries invested money in bringing power lines to rural areas. Electricity became to be a standard.

In parallel, George Boole published his foundational ideas in *The Mathematical Analysis of Logic* in 1847, [5]. Mathematicians like William Jevons, Ernst Schröder, and Charles Sanders Peirce refined Boole's system, transforming it closer to modern set theory. Henry M. Sheffer proved in 1913 that any Boolean function can be built using only NAND operations. Marshall Stone in 1930's proved that any Boolean algebra is isomorphic to a field of sets, bringing it into mainstream modern mathematics. So, both industrial and mathematical platforms for development of modern computers were built in 1930's.

In 1939, George Stibitz from Bell Laboratories demonstrated the first relays' based adder. The simple circuit proved that Boolean logic may be applied to the design of computers. The new era has started. For more information, visit [6].

And what about programs for the computers? As we already mentioned above, in 1843, Ada Lovelace wrote the first published algorithm for Charles Babbage's mechanical Analytical Engine, which then was manually codded by punched cards.

In 1940's Konrad Zuse developed Plankalkül, considered the first high-level programming language. In 1947, Kathleen Booth developed assembly language at the University of London, replacing raw binary code with readable mnemonics. In 1957, John Backus led a team at IBM to create FORTRAN, the first widely used commercial, general-purpose language for science and math. In 1958, ALGOL and LISP were created. In 1959, COBOL was introduced by Grace Hopper for business and administrative systems. In 1964, BASIC was created by John Kemeny and Thomas Kurtz to make computer programming accessible to non-specialists. In 1972, Dennis Ritchie created the C programming language at Bell Labs for operating system development. In 1983, Bjarne Stroustrup released C++, adding object-oriented features to C. In 1991, Guido van Rossum released Python, emphasizing code readability and general-purpose utility. In 1995, Java and JavaScript launched, transforming web interactivity and enterprise software. Computer programmer, software engineer, or data scientist became to be professions.

In general, program languages may be classified according to different criteria. For example, one such a classification divides programming languages into:

- **Imperative or Procedural**, which are focused on how a program operates by giving the computer a step-by-step sequence of commands that change the program's state. Examples: C, Pascal, Fortran. Sometimes also Object-Oriented Programming languages like Java, C++, C#, Python are included in this category or are considered as a separated one;
- **Functional**, which treat computation as the evaluation of mathematical functions. Examples: Haskell, Lisp, Erlang, Elixir.
- **Declarative or Logic-based**, which are focused on what the program should accomplish without explicitly specifying the step-by-step control flow to achieve it. Examples: SQL (for database queries), HTML/CSS (for layout), Prolog (logic-based rules).

While you have a lot of computers, computer systems, and computational processes you want to study these new entities. Two main questions should be answered first: what problems may be solved using computers in general or on some particular kind of computers? The question follows immediately from Result 1. Next, if a problem can be solved, then how the efficiency of the solution may be defined and then estimated. Welcome to the computer science.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

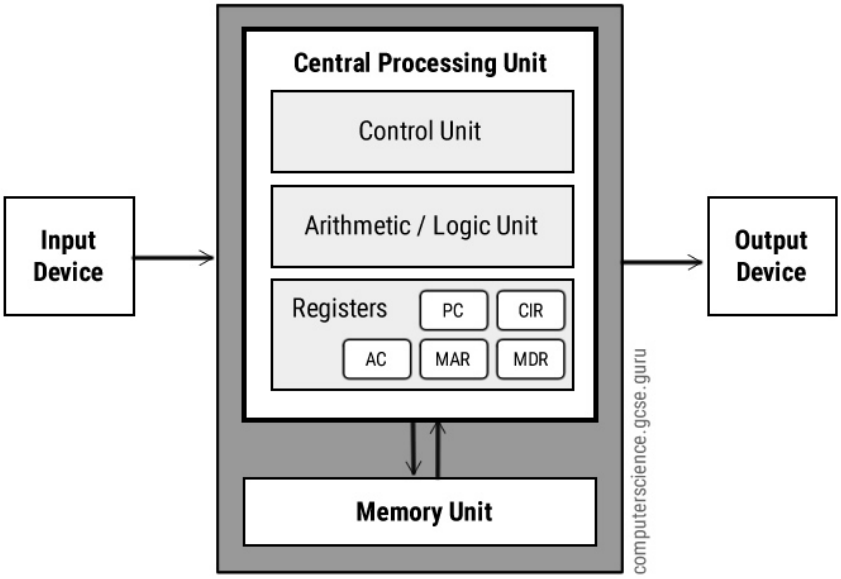


Figure 5. von Neumann architecture or Princeton architecture, [7].

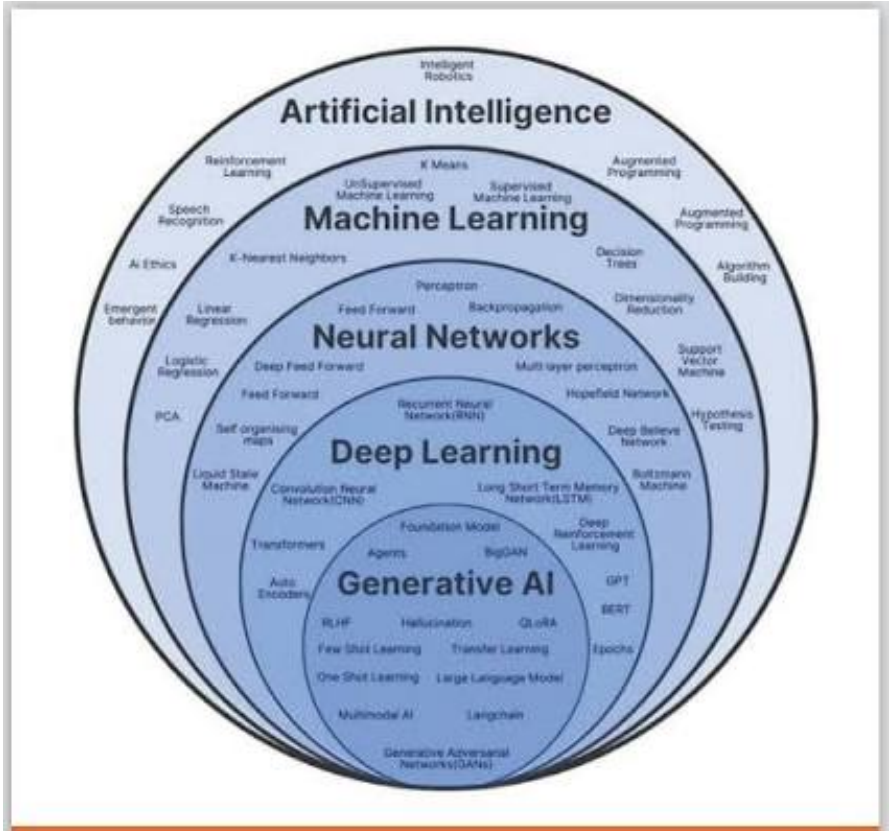


Figure 6. Models of *AI*, [8].

In 1945, [9], mathematician and physicist John von Neumann presented his general model of a computer (also known as the von Neumann architecture or Princeton architecture), which includes, see Figure 5:

- Arithmetic and Logic Unit (ALU) that performs mathematical calculations and logical comparisons;
- Processor Registers inside the CPU (such as the Program Counter (PC));

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

- Control Unit (CU) that directs operation timing, manages instruction fetching, and controls data flow.

The model defines the common abilities of all programming languages:

- execution of arithmetical and logical operations as well as
- manipulations on the content of Program Counter: sequential run of a program, un-conditional jump (like, `goto`) and conditional jumps (like `if` and different variations of loops).

The model allows definition of so-called *intermediate languages* which are low-level, machine-independent code or data structures that a compiler uses internally to bridge the gap between high-level source code (like C++, Java) an raw hardware machine code. Examples of *intermediate languages* are Three-Address Code, Microsoft Common Intermediate Language and so on. Each one of them in different syntax provides exactly the same minimal set of arithmetical and logical operations as well as manipulations on the content of Program Counter.

2. Some Results from Classical Computer Science and Model Theory

When we have in fact infinitely many different computational systems, it is just impossible and there is no any practical reason to investigate each one of them. The commonly used scientific approaches in such situations is modeling and classifications. Classification may be done according to different criteria. In this way, you receive different equivalence classes. Modeling allows simplification and abstraction, which eliminate useless details to focus on most relevant components, and provide extension of the received results of investigations to other real entities under consideration in order to predict their behaviour.

2.1. Models of Computations

For our particular purposes we consider only tree kinds of computational models: Boolean circuits, finite automata and Turing machine.

2.1.1. Boolean Circuits

As we already mentioned in Section 1.2, Boolean circuits were invented by G. Boole in 1847; and Claude Shannon was the first, who applied them to electronic switching circuits in 1937. Input of a Boolean circuit are binary values (0 or 1), the same are outputs. Boolean circuits are composed of different logical gates according to some chosen basis connected into a directed acyclic graph. The Boolean circuit is aimed to provide the corresponding values according to a given Boolean function.

Result 2. *For any Boolean formula, composed from the corresponding basis, the corresponding Boolean circuit may be composed.*

A Boolean circuit, as a two-dimensional object, is characterized by its size: the total number of gates used in the circuit as well as its depth: the length of the longest feasible path from any input node to the final output, representing processing time or delay.

2.1.2. Finite Automata

In 1943, Warren McCulloch and Walter Pitts introduced the first conceptual model of finite-state systems [10] .They used these networks to describe **nervous activity and model biological brain functions logically**.

[10] A Finite State Machine (FSM) and a Finite Automaton (FA) are practically the same underlying mathematical model of computation and are often used interchangeably. However, FSM is the preferred term in engineering and practical system design (focusing on states, inputs, and outputs), while Finite Automaton is preferred in theoretical computer science (focusing on recognizing strings or formal languages).

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

A finite automaton is an abstract machine used to model computation. It consists of a finite number of states, reads input symbols from a finite type that moves in one direction. According to an actual state and the read symbol, the machine may transfer to another state based on a set of rules.

Edward F. Moore (1956) and George H. Mealy (1955) defined finite automata equipped with outputs, known today as Moore and Mealy machines. Michael O. Rabin and Dana Scott (1959) introduced Nondeterministic Finite Automata (NFAs) and proved their equivalence to Deterministic Finite Automata (DFAs). Then, infinitely many variations of automata were defined. In 1962, the Swiss mathematician Julius Richard Büchi invented an extension of FA to the case of infinite inputs. Deterministic Büchi automata are strictly less expressive than non-deterministic ones.

A tree automaton processes tree-structured data instead of linear strings, mapping node labels and child states to a parent state. Bottom-up finite tree automata were invented around 1965 independently by John E. Doner and by James W. Thatcher and Jesse B. Wright. Top-down tree automata and infinite tree extensions were later introduced by researchers like Michael O. Rabin and M. Magidor. Nondeterministic and deterministic bottom-up automata have equal expressive power, whereas deterministic top-down automata are strictly weaker. Infinite tree automata was proposed by Michael O. Rabin. One of the latest and most general extension of tree automata was introduced in [10].

The immediate question: what problem can not be solved using automata?

Result 3.

- *Finite automata cannot solve problems that require unlimited memory or counting: checking if a word reads the same forwards and backwards, verifying if a string has an exact equal number of some letters, matching pairs of brackets or parentheses that go deep to any level, and so on.*
- *Deterministic Büchi automata cannot recognize whether some letter occurs only finitely often in an infinite word. Nondeterministic Büchi automata are much more powerful but still have certain limitations.*
- *Standard finite tree automata cannot solve problems requiring the test of equality or dis-equality between arbitrary distant subtrees. Because they process trees locally (bottom-up or top-down), they lack the memory capacity to verify whether two separate subtrees in a large input are identical.*

The complexity of automata splits into two main concepts: state complexity (the number of states needed to describe a machine or language) and computational complexity (the time and memory resources required for an automaton to process problems), [11]. If automaton is finite, then the run time is $O(n)$, when n is the length of the input. For infinite input the last consideration is not relevant. The question about the needed memory is relevant, for example, for automata with stack. For more information use [12] and [13]. Different variations of automata still remain essential in compiler construction, hardware design, and pattern matching.

2.1.3. Turing Machine

The Entscheidungsproblem: in 1928, mathematician D. Hilbert asked if mathematics was "*decidable*", it means, whether an algorithmic method could determine if any mathematical statement was true or false. To answer this, Alan Turing conceptualized an imaginary "automatic machine" in 1936. He envisioned a device with an infinite tape of cells, a read/write head, and a finite state table of instructions. Turing used this abstract model to prove that:

Result 4. *The Entscheidungsproblem has no solution: certain mathematical problems cannot be solved by any algorithmic procedure.*

Problems that cannot be solved by a Turing machine are called undecidable problems: deciding whether an arbitrary computer program will finish running (halt) or loop forever on a given input; finding a matching sequence of dominoes with top and bottom strings from a finite set; deciding if a given set of shapes can tile an infinite two-dimensional plane, and so on. There are infinitely more

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

unique problems than there are possible computer programs or Turing machines. Trying to solve them creates a self-contradictory loop, similar to the statement "*This sentence is false*", which breaks any logical code trying to evaluate it.

A *recursively enumerable* (also called *computably enumerable* or *Turing-recognizable*) set is one, for which a Turing machine can list all valid members given enough time, or halt and accept any valid input. However, if an item does not belong to the set, the program may run forever instead of stopping.

A Deterministic Turing Machine (DTM) follows a single, unique computation path for any input because each state-symbol pair leads to strictly one action. A Non-Deterministic Turing Machine (NTM) can branch into multiple paths simultaneously, accepting an input if at least one path reaches an accept state.

Turing proved that a single "universal" machine could simulate the behavior of any other specific machine by reading a program stored on tape. During World War II, Turing applied his mechanical computation knowledge to build code-breaking machines at Bletchley Park.

2.2. Complexity of Computation

In the previous sections we provided the main relevant results about computability using difference models, see Results 2, 3, 4.

2.2.1. Boolean Circuits

As we already mentioned in Section 2.1.1, a Boolean circuit is characterized by its size: the total number of gates used in the circuit as well its depth: the length of the longest path from any input node to the final output, representing processing time or delay.

When we have several basic computational units and one memory, four possible scenario to access the memory for read and write operations are possible. Ability to use some the basic computational units in parallel simultaneously provides surprising results:

Result 5. *If you are using a model, which allows concurrent read from the memory and concurrent write to the memory, then you can find the minimum element in an array of length n in $O(1)$ time using $O(n^2)$ basic computational units.*

The best balance between $O(1)$ computational time and the use of $O(n^2)$ basic computational units was found by Nick Pippenger, who in early 1980's formally introduced term "*Nick's Class/Complexity*" (NC) on Boolean circuits with polynomial size and polylogarithmic depth, which mathematically models efficient parallel computation. A problem with input of length n belongs to NC if it runs in time $O((logn)^c)$ using $O(n^k)$ basic computational units for some constant c and k. NC^0 corresponds to Boolean circuits of a constant depth.

AC is similar to NC but allows unbounded fan-in: maximum number of inputs to a single basic computational unit (gate). TC (Threshold Circuit) complexity defines the computational power of Boolean circuits built with unbounded fan-in AND, OR, NOT, and majority (threshold) gates, organized into polynomial-size and small-depth DAG's. It classifies parallel computation, where threshold operations handle counting and arithmetic efficiently. Again, size is polynomial in the input length n. And, depth is $O((logn)^c)$, while TC^0 represents constant depth. Using TC^0 one may compute functions like sorting, integer multiplication, division, and counting.

Result 6. *The following holds: $NC^i \subseteq AC^i \subseteq TC^i \subseteq NC^{i+1}$.*

It means that every problem in NC can be done in TC, AC, and vice versa when collapsing the full hierarchies ($NC = TC = AC$). $AC^0 \subsetneq TC^0$, whether $TC^0 \subseteq NC^1$ is a strict inclusion is one of the main open problems in circuit complexity.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

2.2.2. Finite Automata

As we reported in Section 2.1.2, the complexity of automata splits into two main concepts: state complexity and computational complexity. If automaton is finite, then the run time is $O(n)$, when n is the length of the input.

2.2.3. Turing Machine

Computational complexity using a Turing machine measures the amount of resources: such as time (number of steps) and space (number of tape cells). A theoretical machine requires to solve a problem as a function of the input size n. Binary encoding using 0's and 1's is a standard for complexity theory.

Time complexity counts the total number of transitions or steps a Turing machine makes before it halts and accepts or rejects an input string of length n.

Deterministic time corresponds to $O(t(n))$ steps of a DTM, non-deterministic time corresponds to $O(t(n))$ steps of a NTM, as introduced in Section 2.1.3. Polynomial time (P): the class of problems solvable in $O(n^k)$ time for some constant k. These are considered as tractable or feasible computations. Non-deterministic polynomial time (NP): the class of problems decidable by a non-deterministic Turing machine in polynomial time. Exponential time $EXPTIME$ corresponds to problems requiring time that grows exponentially. It is proven that P is strictly smaller than $EXPTIME$. The same for NP is an open problem.

Space complexity counts the maximum number of tape cells the Turing machine visits and uses during its computation on an input of length n. $LOGSPACE$ problems are solvable while using logarithmic space, $O(log(n))$. Experts generally agree that logarithmic space computation is a key model for highly efficient memory use. Any algorithm that runs in logarithmic space is guaranteed to finish in polynomial time

Deterministic space counts distinct cells marked or read on the work tape. Savitch's theorem, cf. [14], shows that any job done in non-deterministic space $f(n)$ can be done in deterministic space $(f(n))^2$. Polynomial space corresponds to problems solved using polynomial amount of tape cells relative to input size n. It means that $PSPACE$ equals $NPSPACE$.

Result 7.
The following holds: $P \subseteq NP \subseteq PSPACE \subseteq EXPTIME$.
Moreover: $P \subsetneq EXPTIME$.
In addition: $NC \subseteq P$[11].

Combining Results 6 and 7, we finally receive:

Result 8. $AC^0 \subsetneq TC^0$, $TC^0 \subseteq NC^1 \subseteq NC \subseteq P$ *or* $TC^0 \subseteq P$.

2.3. Descriptive Complexity Theory

Descriptive complexity is a subfield of computational complexity theory and finite model theory that characterizes complexity classes by the type of formal languages or logics [12] needed to express the problems within them. Instead of measuring resource usage like time or space on a Turing machine, it measures the expressive power of logical formulas required to define a problem to be solved. Inputs are treated as finite relational structures (like strings, graphs or databases). Major complexity classes match specific extensions of First-Order Logic (FOL) of Second-Order Logic (SOL), as shown on Figure 8, [16].

The classical computational complexity theory assumes that inputs are ordered. However, mathematical logic naturally describes abstract structures like graphs or relational databases, which are not

[11] It is widely suspected, but unproven, that NC is a strict subset of P.
[12] We assume that the reader is familiar with the basic logical notions as presented in [15].

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

naturally ordered. For such a case, some simulation of an ordered presentation of the given information is used. The classical descriptive complexity typically assumes a linear order relation ($\leq$) is present in the vocabulary by default.

2.3.1. Boolean Circuits

As we already mentioned in Section 2.2.1, Boolean circuits are characterized by three main complexity classes: NC, AC and TC.

In descriptive complexity theory, a language is recognizable by a family of polynomial-size, constant-depth circuits with unbounded fan-in AC^0 if and only if it is definable by a FOL formula armed **arbitrary relation and predicate symbols**, [17]. AC^0 cannot express simple counting properties, such as whether the number of input bits equal to 1 is even or divisible by a number m (e.g., the parity function), which are not expessible in FOL.

FOL with counting $FO + C$ is the two-sorted extension of FOL over relational structures that has variables ranging over the non-negative integers, bounded arithmetic on the integer side, and counting terms that give the number of assignments satisfying a formula. Built-in relations are just arbitrary relations on the non-negative integers that the logic can access, independently of the input structure. Over ordered input structures, $FO + C$ with built-in relations captures the circuit complexity class (non-uniform) TC^0[13].

FOL augmented with majority operators yield logspace-uniform TC^0, [18]. TC^0 circuits use **MAJORITY** (threshold) gates, which correspond directly to adding **majority counting quantifiers** to the underlying FOL. TC^0 can perform addition, multiplication, division, and sorting of numbers, can count elements and compute majority functions (e.g., checking if 1s outnumber 0s). TC^0 can not solve inherently sequential or deep hierarchical problems like general context-free grammar parsing, graph connectivity or parity-heavy tasks requiring deep cascading logic without threshold gates.

In complexity theory, uniformity requires a single, fixed algorithm handle inputs of any arbitrary size. Non-uniformity allows a different implementation for each input length, meaning the solver for size n might bear no algorithmic relationship to the solver for size $n + 1$. Uniform complexity captures our intuitive idea of a real-world computer program. A single set of instructions works for input sizes $n = 1, 2, 3, \ldots$. In general, in non-uniform complexity, for length n, a specialized circuit can solve a problem, while how that circuit was generated does not matter, and it can even be uncomputable. The following holds:

Result 9. *FOL defines the class of logspace-uniform AC^0, the languages recognized by polynomial-size circuits of bounded dept; and FOL augmented with majority operators yield logspace-uniform TC^0.*

2.3.2. Finite Automata

Regular languages are another option for formalization of languages, [12]. A regular language is a set of strings defined over a finite alphabet that can be expressed using regular expressions. Kleene's theorem (1956) states that:

Result 10. *A language is regular if and only if it can be defined by a regular expression, recognized by a finite automaton, or represented by a transition graph.*

The theorem shows that these three different ways of describing a formal language have equal power. N. Chomsky established formal grammars, integrating regular languages into the Chomsky hierarchy, [19]. A formal grammar is a mathematical model defined by a 4-tuple $G = (V, T, S, P)$, where V is non-terminals (variables), T is terminals (alphabet), S is the start symbol, and P is production rules. The Chomsky hierarchy classifies formal grammars and their corresponding abstract machines into four levels of increasing complexity, see Figure 7:

[13] Non-uniform TC^0 can compute functions, which are not Turing-computable.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

Result 11.

1. *Regular grammar that restricts production rules to form simple linear substitutions corresponds to finite automaton;*
2. *Context-free grammar that allows a single non-terminal on the left-hand side to expand, regardless of context corresponds to pushdown automaton;*
3. *Context-sensitive grammar, where production rules cannot shrink the length of the string; the expansion depends on surrounding context, corresponds to a Turing machine with a restricted memory tape;*
4. *Recursively enumerable grammar with no restrictions on the production rules corresponds to Turing machine.*

Note that Results 10 and 11 are not only declarative and proclaim existence of the corresponding automata or (version of) Turing machine but rather provide a way to construct them. In practice, K. Thompson [14] and Unix developers utilized regular expressions and implementations of finite automata for text-processing tools like grep and lexical analyzers like *LEX* [15].

Similar results hold also for different extensions of finite automata. For example: Büchi automata and monadic second-order logic are equivalent in expressive power, when defining languages of infinite words. The foundational result is known as Büchi's Theorem, [20,21]. In [10], a weighted version of monadic second-order logic introduced. It is proven that, for commutative semirings, the behaviours of weighted automata are precisely the formal power series definable with particular sentences of the weighted logic.

2.3.3. Turing Machine

Coming to the main results of the descriptive complexity theory, related to the Turing machine, we summarize, see Figure 8 :

Result 12.

- *Immerman-Vardi Theorem of 1982, 1986 established that polynomial time equals first-order logic plus a least fixed-point operator on ordered structures [22,23];*
- *Fagin's Theorem of 1974 proved that NP is precisely the class of properties expressible in existential second-order logic, [24];*
- *Second-order logic with a transitive closure operator (commutative or not) yields PSPACE, [25,26];*
- *Second-order logic with a least fixed point operator gives EXPTIME, [16,27];*
- *Polynomial hierarchy corresponds to second-order logic [28–30].*

[14] Ken Thompson is co-creator of the Unix operating system at Bell Labs in 1969 alongside Dennis Ritchie.
[15] Lexical analyzer tool for the first phase of a compiler.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

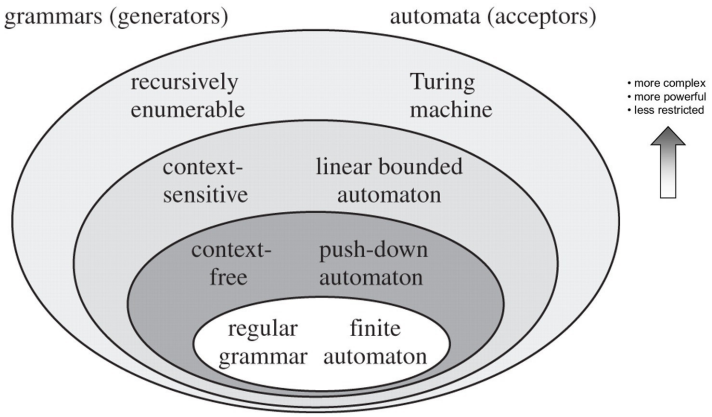


Figure 7. Chomsky hierarchy, [31].

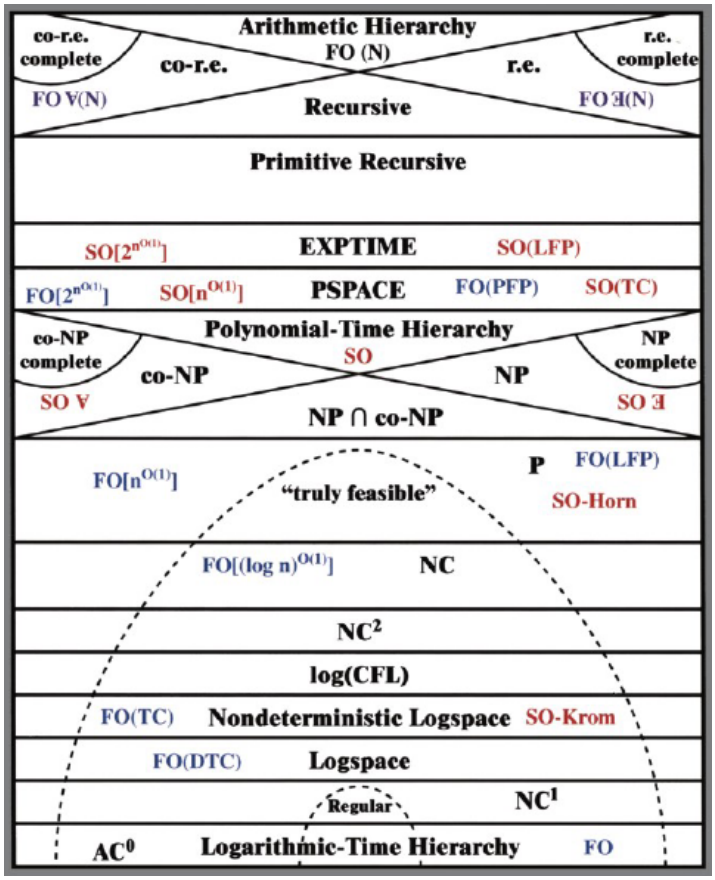


Figure 8. Descriptive complexity, [16]

2.4. Automatic Code Generation

As we already mentioned in Section 2.3.2, most results (but not all) in descriptive complexity theory are not only declarative and proclaim existence of the corresponding automata or (version of) Turing machine but rather provide a way to construct them.

In fact, let us consider one on the most popular application: compilers. Compilation converts human-readable source code into a machine-executable file. The compilation task consists of several steps. The first ones are lexical analysis and syntax analysis.

The first step in the compilation process is lexical analysis. During this stage, the source code is read and converted into a sequence of tokens. A token is a basic unit of meaning, like keywords (`if`, `while`), variables, constants, operators ($+$, $-$), and so on. Lexical analysis uses regular expressions to define token patterns and implementations of finite automata to scan source code and recognize those

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

tokens efficiently. LEX is a tool that automates the process of code writing for the lexical analyzer. It means that the lexical analyzer is not written by a programmer (a human being) but rather by a tool.

The input to the Lex tool is a specification file containing formal regular expressions to be recognized and some portion of C code actions. Lex reads your specification file and generates a C source file. Compiling the last C file you receive the desired lexical analyzer. The automatically generated lexical analyzer, when executed, reads a character from the input stream, matches patterns using finite automata implemented using Kleene's theorem, and executes your C actions whenever the corresponding regular expression is recognized.

Syntax analysis uses formal grammar rules to check if a sequence of tokens has a valid structure. A syntax analysis automatic generator (parser generator) is a tool that reads a formal grammar specification and automatically creates a program (a syntax analyzer) to check token structures and build parse trees. In order to automatically generate the syntax analyzer one should supply a formal grammar file. The generator ingests an incoming stream of discrete tokens. It applies top-down or bottom-up parsing algorithms to validate structural rules and output an *Abstract Syntax Tree* if the process succeed or an error report otherwise.

Popular syntax analysis generators are:

- **GNU Bison**: A widely used, general-purpose parser generator that converts $LALR(1)$ and GLR grammars into C or $C++$ code;
- **ANTLR**: A powerful framework for reading, processing, and executing structured text or grammars, supporting target outputs in $Java$, $Python$, $C\#$, and $JavaScript$;
- **Flex**: Frequently paired with Bison, this fast lexical analyzer generator processes text patterns into tokens before syntax evaluation;
- **JavaCC**: A specialized parser generator designed specifically for building syntax analyzers within $Java$ applications;
- **PLY (Python Lex-Yacc)**: A pure Python implementation of standard Lex and Yacc parsing tools for building custom language parsers.

We emphasize that the main idea is to use a formal specification given by regular expressions, grammars or logics and (using the corresponding results of descriptive complexity theorems) and to produce code of the corresponding automaton or another formalism. One of the latest contribution of the style described in [32], where an automatic code generator for screen based systems is introduced. In this case, the specification is defined in a specially developed graphical language. After the specification is defined by a user, the GUI related code is produced automatically. If specification of a problem to be solved may be expressed in a formal language and the corresponding theorem of descriptive complexity holds, then the implementation flow is rather shown on Figure 9.

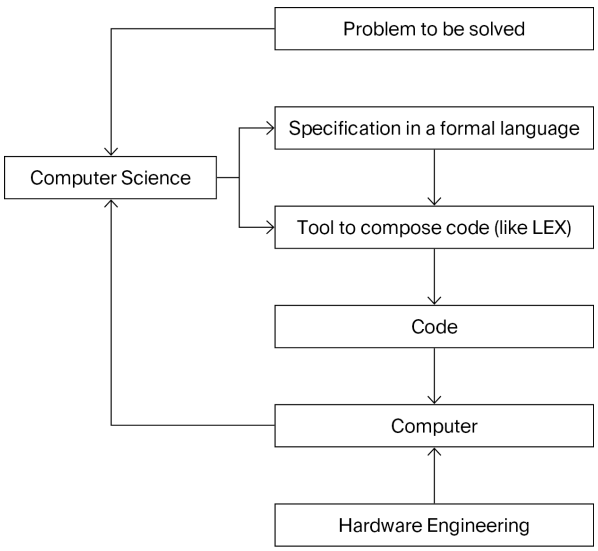


Figure 9. Implementation flow, based on results of descriptive complexity theory.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

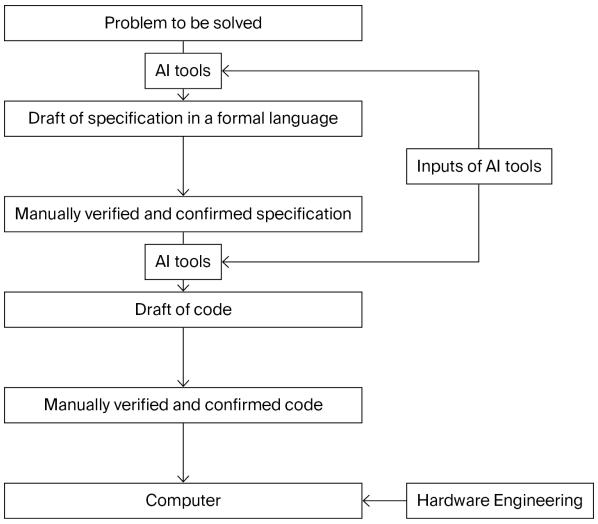


Figure 10. Implementation flow, based on AI tools.

2.5. Correctness and Maintenance of Code

As we already mentioned in Section 2.3.1, our intuitive idea of a real-world computer program is captured by uniform results of descriptive complexity theory. If a code is generated using the flow of Figure 9, then it is expected to be correct upto bugs in the corresponding tools (like LEX), which as a rule work properly. Correctness of a manually written code must be checked.

Code writing is a challenge but even more important challenge is code maintenance. We do not expect that some small change in specification (say, adding a new screen to an application) leads to total rewriting of the corresponding code. Locality of code changes when specification is slightly updated is a must condition for effective code maintenance.

Precise definition of a local code change is not so simple as it may sounds. It leads to a more general question about similarity of different code implementations. Similarity measures for source codes contain but not limited to: similarity in binaries (through re-optimization), equivalence checking and semantic differencing, n-grams based methods, similarity using dynamic approaches, structure-based static methods, machine learning and clustering techniques as well as similarity measures, developed for code synthesis testing and so on, [33].

2.6. Intermediate Summary

In Section 2, we recall some basic results of classical computer science:

2.6.1. Some Results of Descriptive Complexity Theory

- We considered three models of computation engines: Boolean circuits, finite automata, (variations of) Turing machine;
- For each one of the models, there exists a well defined separation between problems, which can be solved under this model and which can not be;
- If a problem can be solved under limitations of a specific model then questions about complexity become to be relevant;
- Two kinds of complexity are defined for each one of the models: time complexity and space complexity;
- Then the following question arises: *Which problems (solvable under the given model) can be proceed in which complexity?*;
- To answer the last question, different formal languages are defined: started from Boolean formula and regular expressions till different logics;
- Relations between formal definitions of problems and complexities of the corresponding solvers are rather provided on Figs. 7, 8.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

2.6.2. Automatic Code Generation

Moreover, from the practical point of view, automatic code generation is not a new feature and it is widely used in writing code for, for example, compiler, based on results of descriptive complexity theory, as presented on Figure 9.

In addition, the final executable code is presented as a string of 0's and 1's. And compilers translate a code, written in a high-level language (like C or Java), into the string. The high-level code and the executable code are equivalent from the point of view of the relation between the program input and its output.

3. AI Revolution: Does Coding Become to Be a Tool?

3.1. General Remarks

Artificial intelligence (AI) is a field of science, focused on building systems that can perform tasks normally requiring human intelligence, such as learning, reasoning, and problem-solving. The main applications of AI are data processing, where AI systems look at large amounts of information instead of just following strict rules; pattern recognition, which try to find trends in data to make choices or predictions; and machine learning systems, which improve on their own over time by practicing with new data.

The history of artificial intelligence spans from early philosophical and mathematical theories, starting even from A. Lovelace's claim of Result 1, which was extended in the 1950's to modern generative models and agentic systems. In 1950, A. Turing published his landmark paper introducing the Turing test to measure machine intelligence; and in 1956, J. McCarthy coined the term "artificial intelligence" at the Dartmouth Conference, establishing AI as a formal academic discipline.

Then, results and interest dropped significantly, when early systems failed to meet grandiose predictions due to hardware limits. In 2017, Google researchers introduced the Transformer architecture, revolutionizing natural language processing; and in 2022, the public release of *ChatGPT* accelerated worldwide adoption of generative AI and large language models.

However, from the computer science point of view, AI-based systems remain in the same situation as computers in 1822, when Charles Babbage presented his *Difference Engine*, see Figure 1, as a fixed mechanical sequence, which is designed to produce table of polynomial functions. It means that AI-based systems should be considered as tools (engines), which are *not programmable*. They are able to read queries (data), however, they can not read instructions. They just use a pre-trained mechanism to answer the corresponding questions. The user can not affect the information, used in the training phase as well as the training process itself. Moreover, the ability to control or affect the run of an AI-based is very limited if any.

AI systems rely on statistical patterns and probabilities rather than rigid, deterministic rules. AI relies on empirical engineering, optimization over partially structured data, and statistical pattern matching rather than deductive proof or closed-form mathematical guarantees. AI models lack a comprehensive mathematical theory to predict their performance before training; developers must rely on trial and error. Generative models can invent facts, miscalculate data, or misinterpret complex contexts while sounding completely confident. An AI's output is only as good, clean, and unbiased as the data it was trained on.

Natural language and real-world inputs sit in a messy meso-scale zone where clean mathematical models do not yet exist. Large language models and neural networks operate by predicting the next token or adjusting weights via gradient descent. They optimize for what "*looks right*" statistically rather than what is logically sound, unlike human errors in mathematics that follow logical missteps. AI errors can be bizarrely illogical yet hidden beneath a fluent, authoritative presentation. Outputs of AI tools must be treated as drafts and must be verified and finally confirmed.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

Remark 1. *The same notion* token *was used in Section 2.4 in context of compilation theory. Here, we use the same word* token *in context of AI with absolutely different interpretation. Please, do not confuse these two concepts.*

3.2. AI and Code Writing

AI code writing uses generative models to automate programming, turning natural language prompts into functional code snippets, complete functions, or automated tests. Generative models of *AI* are artificial intelligence systems designed to create new, original content: such as text, images, audio, and code—by learning patterns from existing training data.

Models analyze massive datasets using unsupervised learning to understand underlying data distributions and probabilities. Instead of "*knowing*" facts like humans, they use mathematical equations to predict the most likely next element or pixels. *AI* based code generation tools use machine learning and natural language processing to translate user prompts into functional syntax, autocomplete lines, and refactor entire projects. Training relies on reducing errors between real outcomes and predictions to ensure generated outputs look realistic.

In practice, *Cursor* is an *AI*-first code editor, which features smart completions, inline editing, and multi-file agent capabilities. *GitHub Copilot* is a classic assistant that offers real-time code suggestions directly. *Claude Code* is an anthropic's command-line agent that reads your entire repository, plans fixes, runs tests, and commits code to *Git* [16]. A systematic literature review on adoption of *AI* tools in software development, which covers period between 2015 and 2025 may be found in [34].

3.2.1. Modeling *AI* Tools

In general, unfortunately, we do not have one well defined model of *AI*, like it was defined by John von Neumann in [9] for computers, see Fig 5. *AI* models are categorized according to different criteria such as:

- Learning Methodology: Machine Learning vs. Deep Learning;
- Technical Architecture: Rule-Based & Expert Systems, Classic Machine Learning Models, Deep Neural Networks;
- Deep Neural Networks: Natural Language Processing, Large Language Models, Computer Vision Models, Generative AI Models, Predictive & Analytical Models;
- Size and Deployment Type: Flagship / Large Models, Small / Task-Specific Models, Open-Weight vs. Closed Models;
- Theoretical Capability & Intelligence Level: Artificial Narrow Intelligence, Artificial General Intelligence, Artificial Superintelligence and so on.

The characterization that is relevant to our review is rather presented on Figure 6.

Especially, current situation with *AI* tools for code generators is pretty similar to that of the beginning of the descriptive complexity theory era. In the case of the descriptive complexity theory, the given landscape was: a pile of different computational engines; and the main question was *What kinds of computations may be executed with these engines?*. The answer was provided by developing three main models of the computational engines with different variants: Boolean circuits, automata and Turing machine; definition of complexity measures using notions of *space* and *time* as well as introduction of different formal languages (different logics, regular languages and so on). Then, relation between requirements, formulated in different formal languages, and effective construction of the corresponding engines in the terms of relevant models was established.

Now, we have a pile of different *AI* based tools for code generation and the questions are:

1. *How can we estimate the ability of the tools to produce trustable code?*
2. *What is the complexity of the tools' activation?*

[16] *Git* is a free, open-source distributed version control system designed to track changes in source code during software development.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

Large language model are characterized but infinite input-output space. The general structure of neural networks naturally connects them to Boolean circuits and hints to use the approaches and tools, developed in descriptive complexity theory.

We expect that the reader is familiar with the basic information about Large Language Model (*LLM*) as presented, for example, in [35]; as well as neural networks as presented, for example, in [36]. Neural networks (*NN*s) are computational models inspired by the human brain that form the foundational architecture of modern *AI* tools. They are composed of: input layer, hidden layers, output layer, weights and biases as well as activation functions. The main types of *NN*s are: Feedforward Neural Networks, Convolutional Neural Networks, Recurrent Neural Networks and Deep Neural Networks.

3.2.2. Large Language Model: Computability, Complexity and Formal Languages

An *LLM* is a type of artificial intelligence trained on massive amounts of text to understand, summarize, translate, and generate human-like language. A token is the basic, fundamental unit of text—such as a word, part of a word, or punctuation mark— that a *LLM* reads, processes, and generates.

Large Language Models have moved past experimental hype and delivered measurable, high-impact success stories across enterprise automation, customer service, healthcare, and specialized domains. For example, an AI-powered customer support assistant named *Clara* used LLMs to support human agents. *Clara* generated $30 million to $40 million in profit/value by drastically reducing customer wait times and deflecting routine queries without requiring full human intervention.

However, the success is not absolute. Five fundamental limitations of computation power of *LLM* are recently reported in [37], see Figure 11:

1. **Long context window**: practical use is curtailed by training on finite windows, inputs that exceed the window, positional-encoding overlap, and computational constraints;
2. **Reasoning**: adherence to rules/logic, exploitation of reasoning patterns, and cross-step consistency remain brittle;
3. **Hallucination**: prompt and sentence-level contradictions—amplified by language complexity, induce factual errors;
4. **Retrieval quality**: database and evidence selection are filtered by retrieval metrics, yet degrade under query ambiguity and attention distraction during integration;
5. **Multimodality**: cross-modal inputs introduce architectural colonization effects, epistemic pitfalls, and scaling/deployment challenges.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

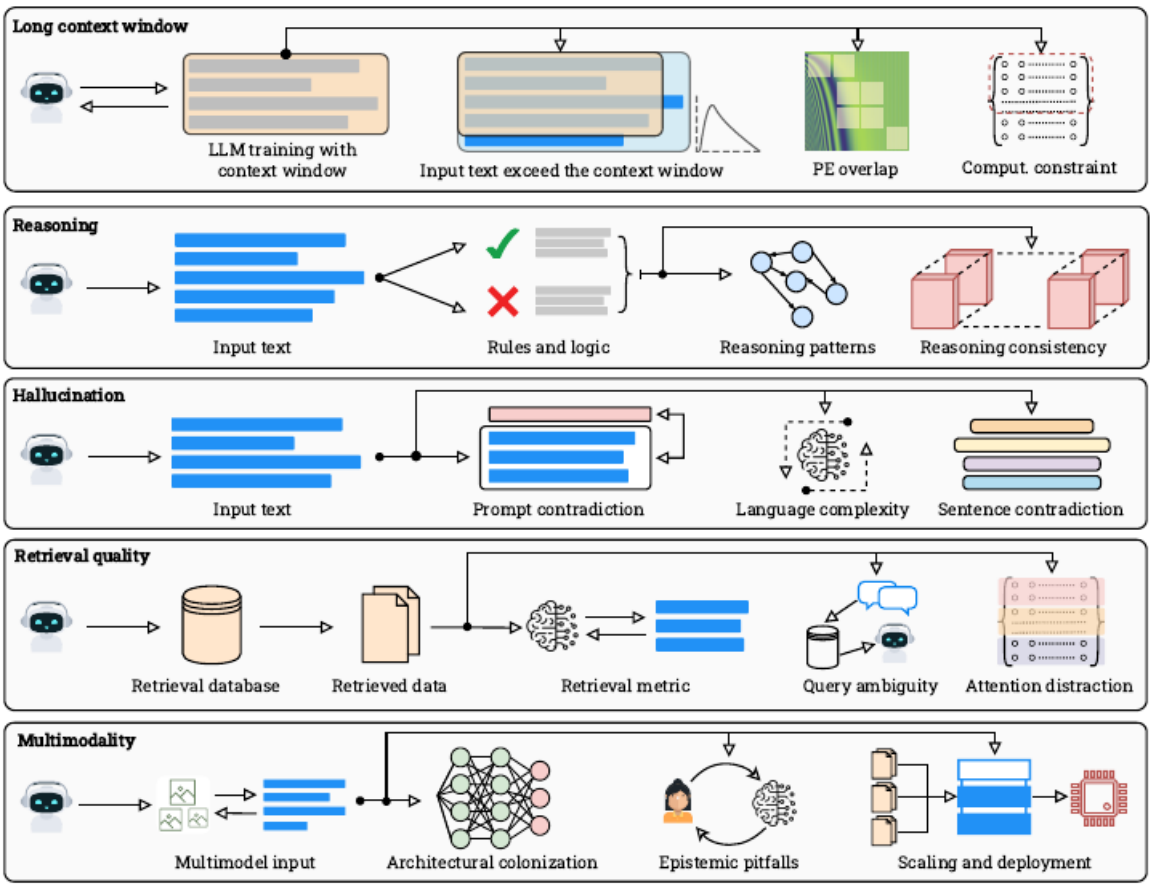


Figure 11. Five interacting fronts that bound *LLM* reliability, [37].

This paper, [37], presents a unified, proof-informed framework that formalizes the innate theoretical ceilings of *LLM* scaling:

1. **Computability and uncomputability** imply an irreducible residue of error: for any computably enumerable model family, diagonalization guarantees inputs on which some model must fail, and undecidable queries (e.g., halting-style tasks) induce infinite failure sets for all computable predictors.
2. **Information-theoretic and statistical constraints** bound attainable accuracy even on decidable tasks, finite description length enforces compression error, and long-tail factual knowledge requires prohibitive sample complexity.
3. **Geometric and computational effects** compress long contexts far below their nominal size due to positional under-training, encoding attenuation, and softmax crowding

The main conclusion of the paper is: *LLM failures scale with capability because they stem from the very theoretical roots that enable language modeling itself. Each failure mode reflects a projection of the same underlying triad: computational undecidability, statistical sample insufficiency, and finite information capacity.* :

1. **Computability theory** shows that no enumerable class of models can correctly answer all computable queries [38,39];
2. **Uncomputability** demonstrates that certain problems lie beyond the reach of any algorithm, [40];
3. **Statistical learnability** reveals that even learnable functions require sample complexity that often exceeds practical limits.

Moreover, in [41], *LLM*s trained on transformer architectures over very large corpora using classic language modeling tasks (next sentence prediction) were investigated. Rather than focusing on a

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

single brand-name product (like $GPT-4$ or $BERT$ in isolation), the paper models and evaluates the theoretical learning constraints of autoregressive and encoder-based transformer language models as a general class.

Instead of treating language models purely as empirical black boxes, the framework analyzes the formal properties of the functions/concepts that language models try to acquire from data based on key theoretical and mathematical assumptions. The presented framework directly corresponds to the case of automatic code generation. From [41], we obtain the following result:

Result 13. *LLMs are unable to learn concepts beyond the first level of the Borel hierarchy* [17].

Moreover,

Result 14. *[43] The first level of the Borel hierarchy consists of the open* [18] *sets and closed* [19] *sets in a topological space, which correspond directly to safety/liveness properties and recognizable behaviors in formal ω-language theory:*

- *A language is open iff it is recognizable by a weak alternating automaton or corresponds to a liveness property (something good eventually happens, prefix-closed properties);*
- *A language is closed iff it corresponds to a safety property (nothing bad ever happens).*

Combining Results 13 and 14, we receive:

Result 15. *LLMs are unable to learn concepts beyond safety/liveness properties and recognizable behaviors in formal ω-language theory.*

In Section 2.3.2, we considered Chomsky hierarchy shown on Figure 7. The Chomsky hierarchy classifies formal languages by their complexity, whereas the Borel hierarchy classifies topological complexity of sets (such as sets of infinite words or ω-languages) using open and closed sets.

On the standard Chomsky hierarchy, which classifies languages of finite words based on their generative grammars, safety and liveness properties do not fit directly because they are strictly defined over infinite words. However, when the Chomsky hierarchy is extended to languages of infinite words recognized by ω-automata, safety and liveness properties slice orthogonal to the hierarchy, rather than sitting inside a single tier, [44].

When studying languages of infinite words (ω-languages) for context-free grammars, researchers map them onto the Borel hierarchy to measure their topological complexity. Deterministic context-free ω-languages sit low in the Borel hierarchy, whereas general context-free or counter automata languages can be much higher or even non-Borel (analytic).

While Result 15 significantly restricts the ability of LLM, the expressiveness of natural languages must be restricted to the precision of formal languages as well. Such an attempt was reported in [45], where $Formal-LLM$ was introduced, see Figure 12. The initial purpose of the tool was to control the LLM-based agent's plan generation. To do so, agent developers construct a context-free grammar (CFG) as the formal language to represent the constraints for the agent.

The CFG is then automatically translated into a pushdown automaton (PDA). When LLM conducts planning, it is prompted to follow the state transition defined by the automaton. This is realized by limiting the LLM-based agent's choices at each step to the valid actions defined by the PDA at its current state, which helps to guarantee that the constraint is satisfied in the final generated plan.

[17] The Borel hierarchy is a way of organizing and classifying Borel sets in a topological space (like the real numbers) by how complicated they are to build from basic open and closed sets, [42].
[18] A set is open if, roughly speaking, around every point in the set you can "zoom in" and stay entirely within the set.
[19] A set is closed if its complement is an open set.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

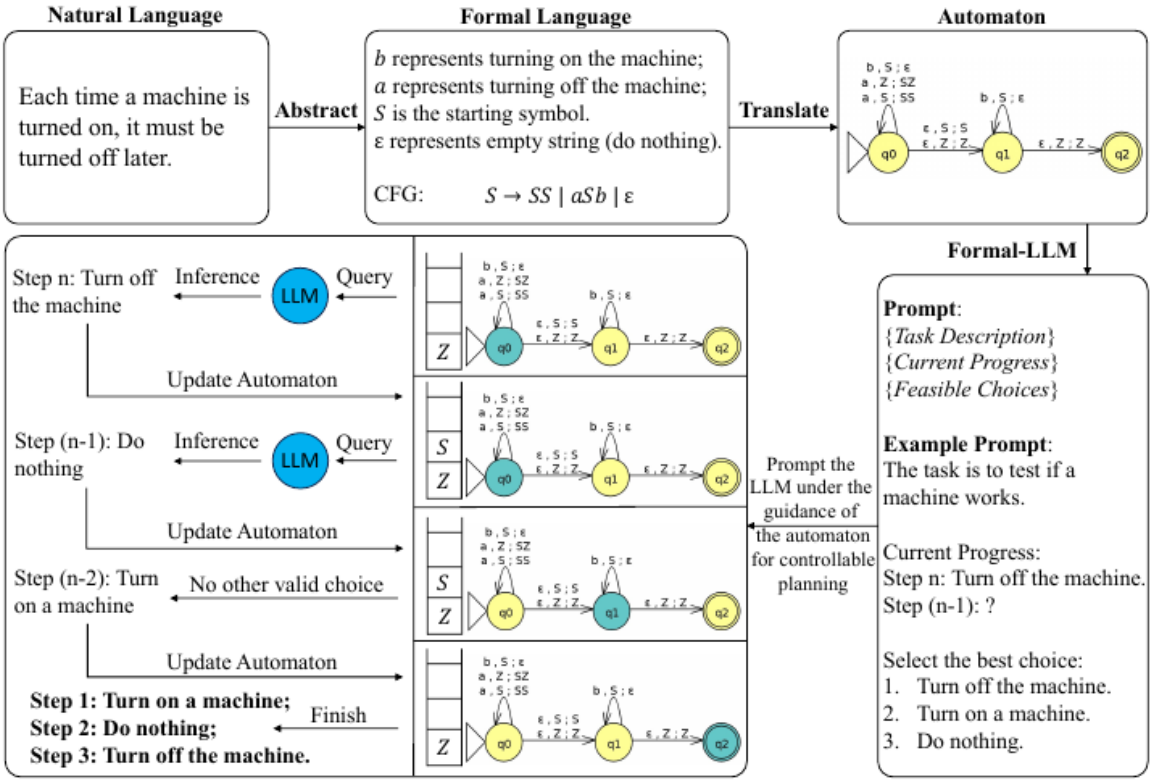


Figure 12. The Formal-LLM workflow, [45].

3.2.3. Graph Neural Network: Computability, Complexity and Formal Languages

In this section, we consider Graph Neural Network *GNN*, as described in [46]. In fact, for example, texts may be presented as graphs, associating indices to each character, word, or token, and representing text as a sequence of these indices. This creates a simple directed graph, where each character or index is a node and is connected via an edge to the node that follows it.

A Graph Neural Network (*GNN*) is a specialized type of deep learning model designed to process data structured as graphs, consisting of entities (nodes) and their relationships (edges). Nodes (vertices) represent individual entities; edges (links) represent relationships or connections between nodes; node/edge features correspond to metadata or attributes attached to the nodes and edges.

The most relevant to the review answer to the computability power of *GNN* was given in [47], using the main approach of the descriptive complexity theory. Here, we take verbatim:

> *We analyse the power of graph neural networks (GNNs) in terms of Boolean circuit complexity and descriptive complexity.*
>
> *We prove that the graph queries that can be computed by a polynomial-size bounded-depth family of GNNs are exactly those definable in the guarded fragment GFO + C of first-order logic with counting and with built-in relations. This puts GNNs in the circuit complexity class (non-uniform) TC^0....*
>
> *Moreover, we show that queries computable by a single GNN with piecewise linear activations and rational weights are definable in GFO + C without built-in relations. Therefore, they are contained in uniform TC^0.*

We recall that non-uniform TC^0 can compute functions, which are not even Turing-computable [20] ; while uniform TC^0 (using Result 8) is under P.

[20] In [48], it was even proved that *GNN*s with random initialisation can approximate all functions on graphs. The caveat of this result is that it is non-uniform and that input graphs of size n require *GNN*s of size exponential in n and depth linear in n that is not, in fact, feasible.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

One of the most recent result of [49] provides logical characterizations of all the functions that can be computed by certain GNN formalisms, not just the intersection with FOL. To do this, logics beyond first-order (so-called *Presburger* logics), but which are still amenable to analysis, are used.

Investigation of the expressive power of neural networks (NNs) from the perspective of descriptive complexity was extended in [50] by giving a logical characterization for general NNs with unlimited topologies and unbounded running time. The characterizations are also based on Boolean networks. It is proven that each NN translates to a Boolean program, which simulates the time series of the NN with each input; and vice versa, Boolean programs can be simulated by NNs.

When a graph grows in terms of nodes and edges, standard GNN training faces severe scalability issues, [51]: memory and time complexity, as well as exponential neighbor blast, which means the number of expanded neighbors grows exponentially in the average degree, [52].

Result of [53] establishes upper bound for the expressive power of GNNs with random node features in the uniform setting: GNNs with random features can at most capture properties which follow a zero-one law. The main question of the paper is *How do GNNs behave as the number of graph nodes become very large?*. Under mild assumptions, it is shown that when one draws graphs of increasing size from the Erdös-Rényi model, the probability that such graphs are mapped to a particular output by a class of GNN classifiers tends to either zero or to one.

When the input data growths in a GNN, growth happens not only across computational complexity but also across architectural degradation and the performance scaling law, [54].

An inherent flaw occurs when a GNN trained on a small graph is exposed to a much larger graph at test time GNNs do not naturally generalize to larger input sizes. A model can easily find a perfect "global minimum" solution on small graphs but completely fail on larger ones if the local connectivity patterns or feature distributions change.

To capture longer-range dependencies in a larger input graph, you need more layers. However, as layers increase, node features tend to converge and become identical, erasing the uniqueness of individual nodes. The situation is called *oversmoothing*. Another problems arises when compressing information from an exponentially expanding neighborhood into a fixed-size node embedding vector. This creates an information bottleneck called *oversquashing*.

As you feed more data into a GNN, the model's accuracy follows predictable *Neural Scaling Laws*, [54]: GNN performance scales predictably against the total number of nodes or edges; assuming the model capacity is scaled up alongside data growth, the error rate drops; the model depth (number of layers) plays a disproportionately massive role in how effectively a GNN scales with larger data volumes.

We already saw that if we follow the general approach of the classical descriptive complexity theory then the input of GNN is expected to be expessible in some formalism, like $GFO + C$ or Presburger logics. Even if the desired input, initially formulated in a natural language (say, English) may be expressed in a desired formalism, it is not a trivial task to find such a translation. Structured reasoning over natural language inputs remains a core challenge in artificial intelligence, as it requires bridging the gap between unstructured linguistic expressions and formal logical representations. A key challenge for translation tasks is the collection of a large corpus of paired language and translated specifications.

Fortunately, we have a promising result in [55], where a parser, which is trained with pairs of sentences and executions as well as an executors presented. The result was extended in [56–58], where it was shown that Large Language Models demonstrated few-shot learning capabilities in many natural language tasks and can be used to overcome data-complexity challenges. The cited papers are dedicated to translations to LTL, which is relevant to different applications, in particular, robotics.

A more general approach is presented in [59], where a novel **bi-level framework** that maps language to logic through a two-stage process: high-level task abstraction and low-level logic generation was presented. Experiments on multiple realistic reasoning benchmarks demonstrate that the approach significantly outperforms existing baselines in accuracy, with high accuracy gains.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

In particular, natural language to FOL translation via code generation was presented in [60]. It is based on in-context learning and employs code snippets to bridge the gap between natural language and first-order logic and reported good results. See also [61].

In order to combine the presented result and those of [47] they should be adopted to the relevant logics like $GFO + C$ or Presburger logics, which seems to be doable.

3.3. Intermediate Summary

In Section 3, we presented the latest results about the use of LLM as well as GNN, which seems to be the most relevant kind of NNs, for code generation.

3.3.1. Computability and Complexity

For LLM, five interacting fronts that bound LLM reliability were investigated in [37]. These pathologies persist even under massive data, suggesting deeper computational and statistical origins. And such behaviors are not transient artifacts of optimization or data curation but manifestations of intrinsic theoretical barriers, constraints imposed by computability, information theory, and learnability itself.

In fact, LLMs are unable to learn concepts beyond safety/liveness properties and recognizable behaviors in formal ω-language theory. While Result 15 significantly restricts the ability of LLM, the expressiveness of natural languages must be restricted to the precision of formal languages as well. Such an attempt was reported in [45], where $Formal - LLM$ was introduced, see Figure 12.

For GNN, there exists a well defined separation between problems, which can be solved under this model in feasible complexity and which can not be due to [47]. We cite again the most relevant result: *...queries computable by a single GNN with piecewise linear activations and rational weights are definable in GFO + C without built-in relations. Therefore, they are contained in uniform* TC^0.

3.3.2. Automatic Code Generation

We presented the latest results about automatic translation from natural languages like English into formal logics likes FOL or LTL, when possible. The evaluation of the relevant tools reports very good results, but still their translations should be controlled and finally confirmed manually. Moreover, for our best knowledge, there is no such a tool for $GFO + C$ and Presburger logics, why it seems to be doable.

In general, as for now, we demonstrated the following flow: an input for code generation engine is automatically translated into the corresponding formal presentation (like $GFO + C$ or Presburger logics), when possible (due to natural limitations of the expressive power of the formal presentation); the input goes to the corresponding AI-based tool that generates the desired code automatically. Then the code must be finally tested and its correctness must be confirmed, see Figure 10.

4. AI-Added Code Writing: In the Beginning Was the Specification [21]

In order to prevent any mis-understanding, in the paper, we deal only with functional specifications. Un-functional specifications are definitely out of the scope of the review. Each time a word *specification* is met it must be understood as *a functional specification*.

Coming back the the main stream of our review: on the one hand, we may give to an AI-based tool a specification and (to hope to) receive the corresponding implementation in code. On the other hand, due to specifications, formulated in an un-precise natural language like English, *garbage in - garbage out* principal still holds. AI-based tools are not precise by definition; and when they are activated on a *free-style* formulated input, the obtained result is strongly expected to be meaningless.

The last consideration emphasizes the crucial importance of precise formulation of what we want from the AI-based tool, aimed to generate the desired code. While in the classical approach of Figure 9, a specification, formulated in a natural language comes to a programmer, who is expected *to understand*

[21] *In the beginning was the Word, and the Word was with God, and the Word was God*, [62].

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

it and to translate it into a programming language; in the case of AI-based tools of Figure 10, we, in fact, have a problem with *understanding*, as the tools are based on learning via examples rather than on learning via understanding. That is why results of AI-based tools may be and must be manually finally confirmed in our particular case of implementation of code writing. It means that we hardly may eliminate a programmer from the implementation flow completely.

On the other hand, as we already mentioned, we face severe scalability issues when, for example, an input graph grows. However, especially this scalability problem is less relevant to our case. In fact, one of the scalability solutions is subgraph partitioning that is very natural for code writing. The massive graph is cut into smaller, isolated clusters. The AI-based tool trains on one cluster at a time. In the case of code writing the clusters are functions or methods, depending of the notions, used in the given programming language.

We use the known notion of a module in an hierarchically composed code and define it precisely as follows:

Definition 1. *Module in an hierarchically composed code is a cut of the hierarchy with the corresponding well defined interface, which describes its input(s) and possible outputs as well as its functionality, which is a relation between input(s) and possibly output(s) or other results of its run* [22].

The idea of automatic code generation using specifications, expressed in formal languages and especially graph-based languages, is not really new, cf. [63]. One of the most recently developed branch in the general field is automatic code generation for Graphical User Interface (GUI). The corresponding contributions are mostly based on Model-Driven Architecture (MDA) as described in [64] (see also for more details, for example, [65]) and intensively exploit Unified Modeling Language (UML), cf. [66], or similar formalisms.

As a rule, these approaches start from the assumption that the relevant UML diagram is given and it depicts the corresponding model, while, say, the UML diagram should be manually composed, cf. [67–72]. For example, in our papers, [32,73], we presented the complete process, starting from a graphical GUI description until GUI code generation for screen-based applications.

In order to generalize the approach, we propose to use the notion of a module from our Definition 1. The interactions between modules are defined using a graph-based presentation and their functionality is defined using the corresponding formal languages.

4.1. Code Writing First

GNNs are revolutionizing code generation by addressing a major flaw in traditional Large Language Models (LLM's): the inability to explicitly process the structured, relational nature of source code. While standard transformers view code as a flat sequence of text tokens, GNNs explicitly map syntactic and semantic structures.

One can use $GNNs$ for code generation by converting source code into graph structures: such as Abstract Syntax Trees (AST) [23] or control-flow graphs — so, the model can learn code syntax, relationships, and logic. Nodes represent elements in the code like variables, functions, operations, or classes. Edges represent relationships like data flow, control flow, or syntactic parent-child connections in the graph. Node features describe what each code token or variable does.

One of the first attempts to practically integrate GNNs advantages into code generating engines was reported in [75]. The approach, named $GNN-Coder$, combines the use of GNN's and Transformer model [24], as presented on Figure 13. The contribution studies how GNN-integrated Transformer can promote the development of semantic retrieval tasks by capturing the structural and semantic

[22] In general, a module may not have a well defined output. For example, when some file must be deleted.

[23] Abstract Syntax Tree is a hierarchical tree data structure used in computer science to represent the logical structure of a program's source code, introduced by John McCarthy in 1960s, [74].

[24] Transformers convert raw code into context-rich, continuously distributed vector representations (embeddings) that preserve semantic meaning across downstream tasks, [76,77].

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

features of code. This proposed approach shows ability to distinguish between code snippets, thereby enhancing retrieval accuracy in the general code generation flow, based on reuse of existing sources of code.

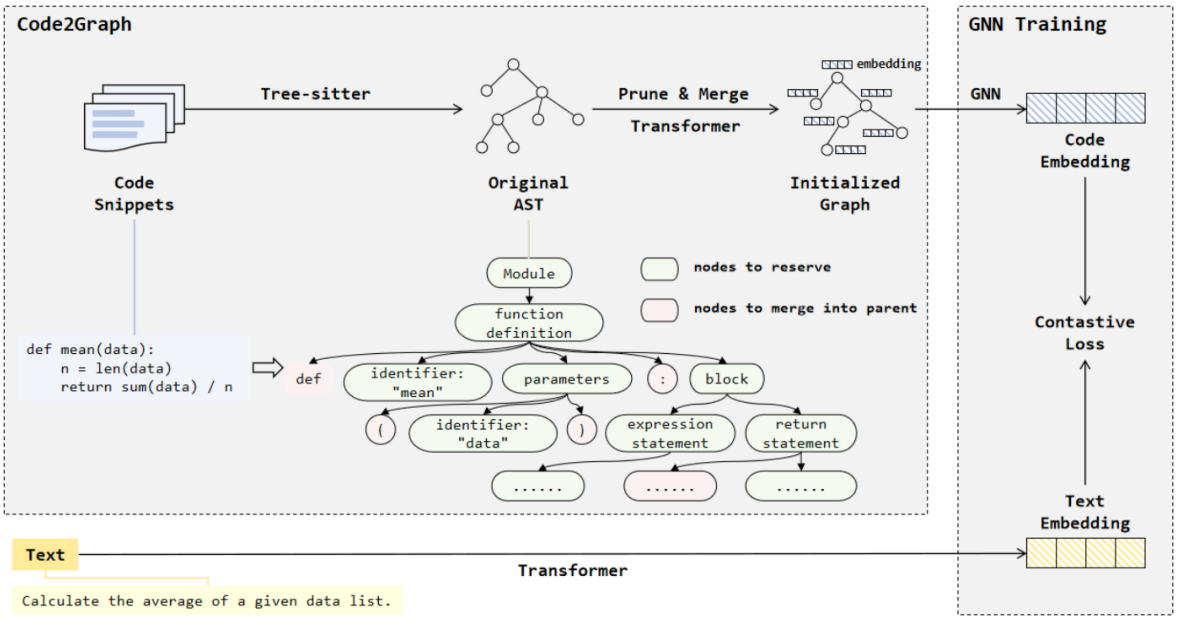
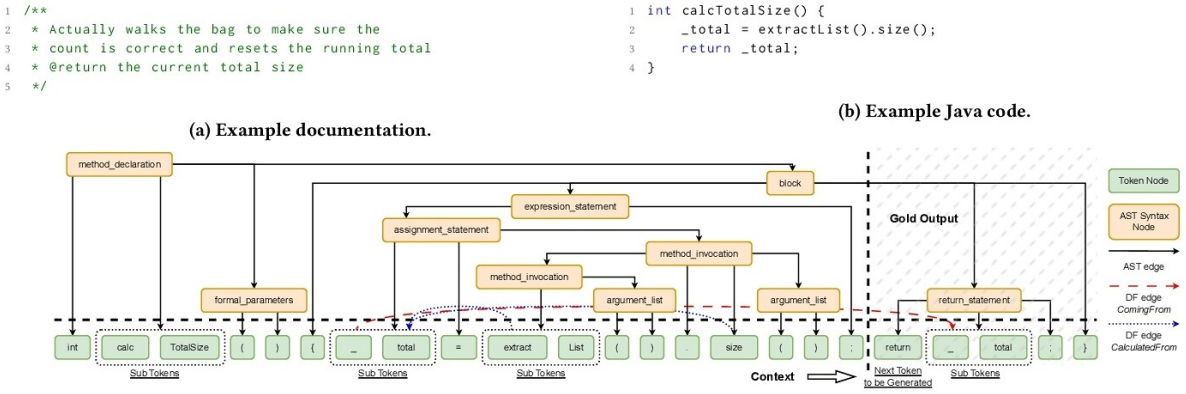


Figure 13. Overall architecture of GNN-Coder, [75].

Instead of operating entirely in isolation, modern architectures combine *GNN*s and *LLM*s to create *"structure-aware"* code generation engines, [78]. The introduced framework (*CGFuse*) enables token-level integration of graph-derived representations by infusing learned graph features directly into the intermediate layers of pre-trained language models. This approach allows combination of three main components: graphical presentation, specification as well as some code. See Figure 14.



(c) Code graph with AST (solid lines) and DFG edges (dashed/dotted). Syntax nodes s are orange, terminal nodes t are green.

Figure 14. A sample Java snippet with documentation (a), code (b), and its augmented code graph (c), [78].

4.2. Test Writing First: Incremental Code Generation and Test Driven Design (TDD)

As we mentioned above, one of the main challenges of *AI*-based code generation is the relation between the length of the provided information about the desired implementation and the complexity of the tool. One of the promising approaches to deal with the challenge is Test Driven Design (*TDD*).

4.2.1. *TDD*

Test Driven Design (or Development) (*TDD*) is a software development approach, where you write tests before writing the actual production code, [79]. In general, the approach remained to be

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

mostly theoretical and hardly used in practice till now. However, it seams to be very promising while combined with AI-based tools.

In fact, according to [80]: *software testing . . . is the process of applying a few well-defined, general-purpose test criteria to a structure or model of the software.* A test (in software testing) is a single, structured procedure, where you run a piece of code or an application with specific inputs to see if the actual output matches the expected result.

TDD usually follows the "Red-Green-Refactor" cycle, [81], see Figure 15:

1. Add a test to the test suite;
2. (Red) Run all the tests to ensure the new test fails;
3. (Green) Write just enough code to get that single test to pass;
4. Run all tests;
5. (Refactor) Improve the initial code while keeping the tests green;
6. Repeat.

As each test expects some particular output for the given input, the trivial implementation for TDD approach is just writing a *case*-like code, which for the given input just provides the expected output at step (3), above. In order to prevent such tricks, more information must be provided.

Minimal additional information is expected to include the full definition of the inputs and (possibly) outputs of the corresponding modules, as well as, the relation between inputs and outputs: functionality.

The tests may be written manually or by using any suitable tool, for example also AI-based. We consider a set of tests given in advance. Any question about completeness of the set is out of the scope of the review. More information about consistency may be found in particular in [32].

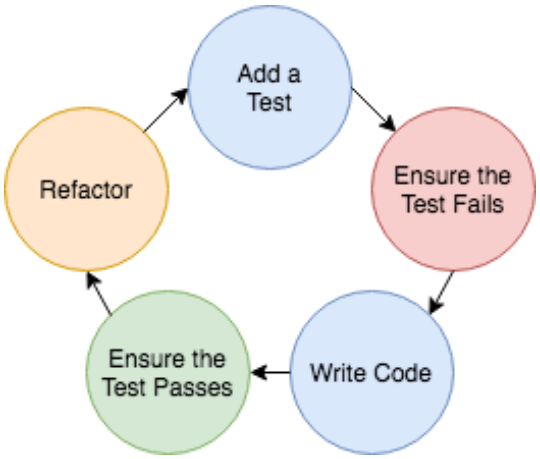


Figure 15. "Red-Green-Refactor" cycle. [81].

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

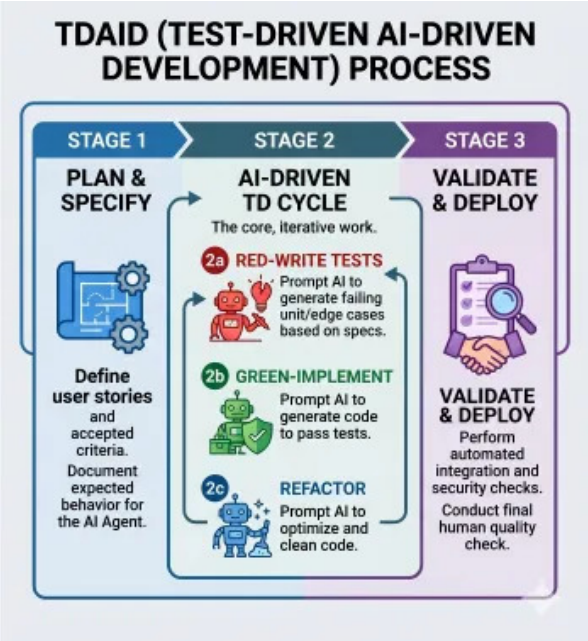


Figure 16. *TDD*, based on *AI* tools, [82].

4.2.2. *TDD*, *LLM* and *GNN*

By integrating computability, information theory, and statistical learning, in [37], it was demonstrated that certain classes of errors are not removable artifacts of training or architecture—but are necessary consequences of computation itself. This synthesis reframes *LLM* research: the challenge is not to eliminate failure, but to understand, bound, and allocate it optimally. So, let us start from tests.

Combination of *TDD* principles and *LLM*-based code generation was presented in [83]. The authors experimentally evaluated their hypothesis that providing *LLM*s like $GPT-4$ and *Llama* 3 with tests in addition to the problem statements enhances code generation outcomes. the authors experimented with established function-level code generation benchmarks such as *MBPP* and *HumanEval*. Their results consistently demonstrate that including test cases leads to higher success in solving programming challenges. They concluded that *TDD* is a promising paradigm for helping ensure that the code generated by *LLM*s effectively captures the requirements.

Self-debugging with self-generated tests for code generation was investigated in [84]. In the last paper, two distinct paradigms for the self-debugging process: post-execution and in-execution self-debugging, were introduced. The reported findings reveal that post-execution self-debugging struggles with the test bias introduced by self-generated tests, which can lead to misleading feedback. In contrast, in-execution self-debugging enables *LLM*s to mitigate this bias and leverage intermediate states during program execution.

One of the most recent developments of the combination of *TDD* and *LLM*s may be found in [85], where a $TDD-Agent$, which operationalizes the test-driven development paradigm for code generation, was introduced. The $TDD-Agent$ first prompts the model to generate executable tests, encouraging it to clarify expected behaviors before implementation, and then performs iterative dual-track refinement over both the generated code and tests using execution feedback. Overview of the TDD-Agent is rather provided on Figure 17.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

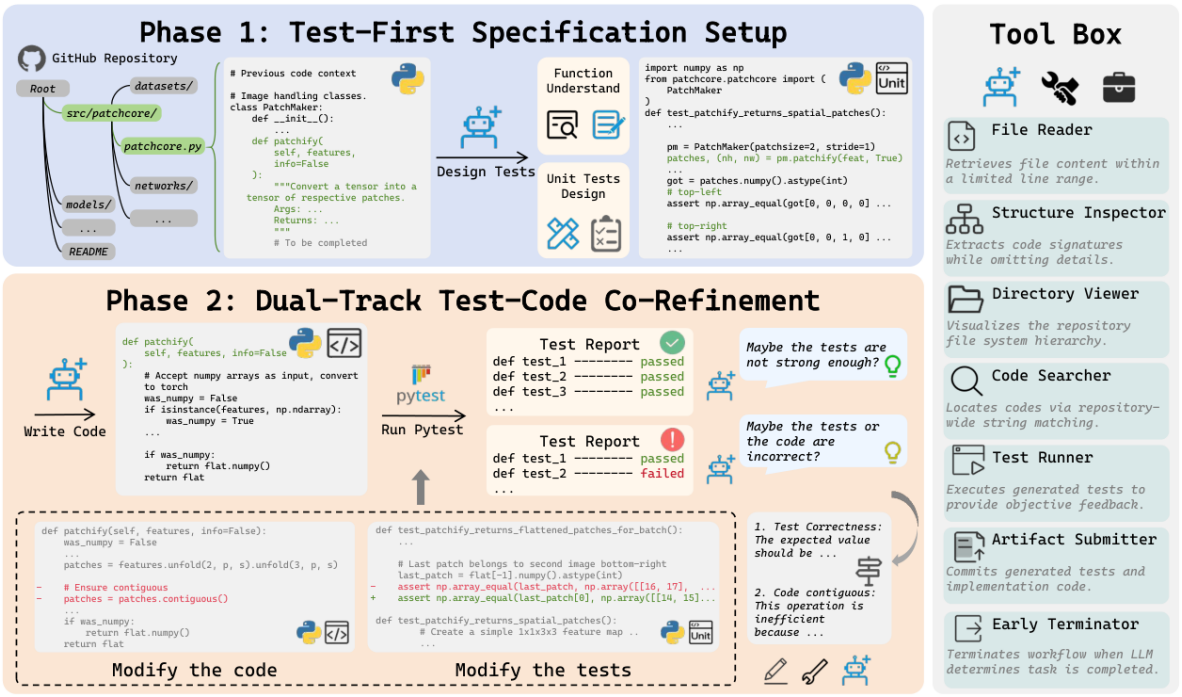


Figure 17. Overview of the TDD-Agent. (1) The LLM first designs and generates unit tests for the target function. (2) The LLM generates the complete implementation of the target function, and through execution and refinement, iteratively refines the tests and code. [85].

On the other hand, one can use *TDD* to build a *GNN* pipeline by writing tests for the following components before writing the actual code: graph data validation, permutation invariance, message passing verification and overfitting a single batch.

In fact, *GUARD*, a *GNN*-based tool for automated unit test case generation and code defect prediction, was introduced in [86]. *GUARD* first converts the source code into a *Program Dependency Graph (PDG)*, using *PDG* to represent the structure and semantic information of the code. Then, a defect prediction model based on a graph convolutional network (GCN) is constructed to predict the defect probability of each node (code basic unit). Finally, based on the prediction results, a test case generation strategy is designed to achieve automatic generation of targeted unit test cases, as shown on Figure 18.

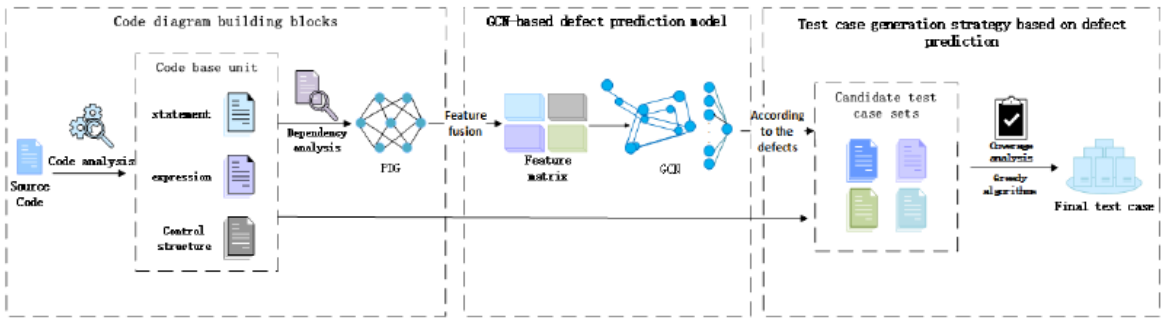


Figure 18. *GUARD* framework, [86].

As we already reported, modern architectures combine *GNN*s and *LLM*s to create *"structure-aware"* code generation engines, [78]. One of the most recent developments of the combination of *TDD* and *LLM*s may be found in [85], The $TDD-Agent$ of [78] operationalizes the test-driven development paradigm for code generation using *LLM*. *GUARD* of [86] is a *GNN*-based tool for automated unit test case generation and code defect prediction. We have not managed to find any reported approach that combines all of the three features: *LLM*, *GNN* and *TDD*.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

4.2.3. TDD-Based Incremental Code Generation Using AI

In this section, we present our generalized vision on the complete TDD-based incremental code generation and maintenance using AI, based on the result reported above.

1. **Computability limitations** As it was shown in [41,43,47]:
   - LLMs are unable to learn concepts beyond safety/liveness properties and recognizable behaviors in formal ω-language theory, see Result 15; and
   - ...the graph queries that can be computed by a polynomial-size bounded-depth family of GNNs are exactly those definable in the guarded fragment $GFO + C$ of first-order logic with counting and with built-in relations. This puts GNNs in the circuit complexity class (non-uniform) TC^0.... queries computable by a single GNN with piecewise linear activations and rational weights are definable in $GFO + C$ without built-in relations. Therefore, they are contained in uniform TC^0.
2. **From natural languages to formal languages** Based on the previous item, we may be interested in automated translations from natural languages into the required formal languages, like $GFO + C$ or Presburger logics (when it is possible) for un-experienced users. In this direction of further research, we have promising results of
   - [60], where natural language to FOL translation via code generation was presented; and
   - [56–58], which are dedicated to translations to LTL.

   However, we have not managed to find such translations to the required formal languages. In general, if this approach is chosen then the user must be informed whether the given mission may be executed. For example, if such a user gives some coding of a Turing machine and asks whether it halts on each input, he/she must be informed that the query can not be answered in any case.
3. **Formal language first** For professional users, who are interested in AI-tools for automated code generation and maintenance, we propose another approach that is again widely used for different software engineering purposes. In Section 2.4, we presented LEX as a tool, based on Result 10, which transforms some tokens' definitions (in the sense of compilation theory, see Remark 1), based on regular languages, into implementation of the corresponding finite-automata.
   In general, in Section 2.3, we mentioned logic-based languages, which are widely used in software engineering for dozens of year for different purposes. One of the classical examples is Prolog. In [73], we used PROMELA (PROcess MEta LAnguage), which was designed by Gerard J. Holzmann, [87], that is the input and modeling language for SPIN. In SPIN/PROMELA, LTL is used to mathematically prove safety and liveness properties. Compare to Result 15.
4. **Incremental code generation and maintenance** AI-based code generated from scratch seems to be a very important challenge. However, code maintenance seems to be even more challenging. As a rule live cycle of commercial software lasts dozens of year with relatively small changes from one version to the next one. That is why in our paper [88], which is expected to be published soon, we strongly advise to use TDD approach to code development automation with incremental code changes.
5. **Estimation of the maid changes: distances between codes** The exact definition of a change made to the code to be incremental strongly depends on the chosen distance metric. Survey of different distances between codes may be found, for example, in [33]. One of the most recent metric was presented in [75] for development of semantic retrieval tasks by capturing the structural and semantic features of code.
   The Mean Angular Margin (MAM), is a novel metric for quantifying the uniformity of code embedding distributions, providing a standardized measure of feature separability. The used AI tool should be directed to make only incremental code changes in each cycle of the TDD. We report our experience in the field in [88].
6. ***LLM, GNN* and *TDD* altogether** In this review, we presented

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

- Combination of *TDD* principles and *LLM*-based code generation as shown in [83]. One of the most recent developments of the combination of *TDD* and *LLM*s may be found in [85].
- Combination of *TDD* principles and *GNN* as shown in [86].
- Combination of *LLM* and *GNN* as shown in [78].

Based on our Definition 1, we propose to describe specification of modules' interaction using graphical presentation; and their functionality in text-based formal languages. Then, the complete specification should be integrated into *TDD* cycles.

Finally, we propose the following complete integrated workflow:

- Given full specification, based on our definition of modules as described above. Given a predefined set of tests.
- Given an *AI*-based code engine, trained to produce incremental code generation.
- At the zero step, based on the full specification and set of tests, which contains one chosen test, the first Red-Green-Refactor cycle of *TDD* is completed.
- At each next step, based on the full specification, one more test added to the set of already proceeded tests and the code, generated till now, the next Red-Green-Refactor cycle of *TDD* is completed.
- The loop ends while all tests are proceeded. The generated code goes to the final confirmation and deployment.

The approach significantly extends that presented on Figure 10.

1. We divide specification into two parts: I/O definition and functionality definition, which may be presented in different formulations; and
2. We include the already generated part of code as a starting point of the next *TDD* cycle in order to guaranty incrementality. Moreover,
3. We emphasize the crucial role of the metric, chosen for estimation of distances between codes, generated on the previous and actual *TDD* cycles.

5. Summary and Outlook: From a Zoo to a Zoology [25]

In this review, we built a bridge between the most recent results from different fields of computer science and software engineering, related to automated code generation using *AI*-based tools.

We use an historical retrospective in order to establish similarity between industrial revolution and our days. This approach allows adaptation of the historical successful experience to our days challenges. The approach allowed identification of gaps, which exist in both research and implementation.

The first observation is that from the computer science point of view, the modern *AI*-based systems remain in the same situation as computers were in 1822, see Figure 1. *AI*-based systems can be considered as tools (engines), which are *not programmable*. They are able to read queries (data), however, they can not read instructions. They just use a pre-trained mechanism to answer the corresponding questions.

The user can not affect the information, used in the training phase, as well as the training process itself. Moreover, the ability to control or affect the run of an *AI*-based tool is very limited if any. In order to reach at least the situation of 1837, when Charles Babbage designed *Analytical Engine*, see Figure 3, *AI*-based tools should become to be *programmable* in the sense as we use the notion now for computers.

On other hand, if we compare interaction between a user and the first computational engines to interaction between a user and the modern *AI*-based engines, the situation is just opposite. The first users of Babbage's engines used manually punched cards, as shown on Figs. 3, 4, that is, in fact, the most limited communication way. In the contrary, the modern *AI*-based tools provide free-style access to their engines that is even worse. A user may ask any question, even a question that is proven to be

[25] The title is taken from [89].

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

un-decidable in computer science. In this situation, the user can not estimate the trustability of the corresponding answer. In order to solve the problem, we propose the use of *prompt* languages:

- either the free-style query is translated to some formal language, if possible; and, than the user is informed what query is really answered, or
- the user from the beginning is limited by use of some (logic-based) language.

We emphasize that does not exist a general model of AI-based engines, like von Neumann model for computers, see Section 1.2. Next, we consider three classical models of computations: different variations of automata, Boolean circuits and the Turing machine. For each of them, we provide the corresponding results from classical computability and descriptive complexity theory, as summarized on Figs. 7, 8.

We consider two models of AI-based engines, see Figure 6: Large Language Model (LLM) and Graph Neural Network (GNN). We show the most recent results, which provide severe limitations on computability and complexity of both of them. We think that the results should be extended and adopted to full systematic hierarchies, which exist in classical computer science, see Figs. 7, 8.

In addition, we emphasize the crucial difference between the classical results of computability and descriptive complexity theory and the hopes, related to the use of AI-based tools for automated code generations. While the classical methods are precise and are based on proofs; those of AI are, in fact, approximations and strongly depends of different external factors, [37], see Figure 11. We explicitly declare that in this part of the review we do not generalize any result from descriptive complexity. We just precisely cite the corresponding results as they are stated.

Based on the limitations of computability of AI-based engine, we again point out the crucial role of automated translations from natural languages to formal languages, when we are talking about AI-based tools dedicated to un-professional user. We emphasize again that, for our best knowledge, such translations do not exist as for now for the cases of formal languages ($GFO + C$ or Presburger logics), which are relevant to the scope of the review. However, we show a list of results from different branches of science, which provide some evidence that they should be doable.

For professional users, we expect development of special-purpose logic-based prompt languages, which from the beginning limit the user to use some syntax, which may be effectively translated to the implementation of the corresponding logics. We provide examples of such languages used in other fields of software engineering.

As code maintenance is not less important than code generation from scratch, we underline the importance of Test Driven Design (TDD) in automated code generation using AI-based tools. That is why, any change, made in the code must be as local, as possible. The last observation requires precise definition of distances between versions of the desired code at each TDD cycle. This consideration seems to be a new in the field. We provide examples of metrics, including the recent ones, which are aimed to measure distance between codes via different ways.

Finally, we define a complete workflow, which combines all the presented features: results from the computability and the descriptive theory, translation from natural languages to formal languages, logic-based languages, code distances as well as combination of TDD, LLM and GNN altogether, that is new.

Acknowledgments: My heartfelt appreciation goes to my colleagues, whose insightful discussions and suggestions have greatly enriched my work. In particular, I am thankful to E. Kleiman, K. Korenblat, D. Lemberg and J. A. Makowsky, for their collaborative spirit and support. **Errata**: We explicitly declare that in this review some sentences may be taken verbatim from the corresponding sources in order to provide precise presentation. On the other hand, all the used sources are cited explicitly.

References

1. Britannica, E. Difference Engine. https://www.britannica.com/technology/Difference-Engine#/media/1/725539/19223, 2026. Accessed: 2026-08-08.
2. Technik, W.M.T.u. Jacquard Loom. https://en.wiktionary.org/wiki/, 2026. Accessed: 2026-08-08.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

3. Britannica, E. Analytical Engine. https://www.britannica.com/technology/Analytical-Engine#/media/1/22628/219565, 2026. Accessed: 2026-08-08.
4. Britannica, E. Ada Lovelace: British mathematician. https://www.britannica.com/biography/Ada-Lovelace, 2026. Accessed: 2026-08-08.
5. Boole, G. *The Mathematical Analysis of Logic: Being an Essay Towards a Calculus of Deductive Reasoning*; Macmillan, Barclay, & Macmillan: Cambridge, 1847.
6. Museum, C.H. Timeline of Computer History. https://www.computerhistory.org/timeline/computers/, 2026. Accessed: 2026-08-08.
7. ComputerScience.gcse.guru. von Neumann Architecture. https://www.computerscience.gcse.guru/theory/von-neumann-architecture, 2026. Accessed: 2026-08-08.
8. Dream, R. Introduction to the World of Artificial Intelligence. *American Pharmaceutical Review* **2025**, *28*, 22. Managing Director, HDR Company LLC.
9. von Neumann, J. First Draft of a Report on the EDVAC. Technical report, Moore School of Electrical Engineering, University of Pennsylvania, 1945. Contract No. W-670-ORD-4926, U.S. Army Ordnance Department.
10. Droste, M.; Gastin, P. Weighted automata and weighted logics. *Theoretical Computer Science* **2007**, *380*, 69–86.
11. Hartmanis, J.; Stearns, R. Automata-based computational complexity. *Information Sciences* **1969**, *1*, 173–184.
12. Pin, J.É., Ed. *Handbook of Automata Theory*; European Mathematical Society Publishing House: Zürich, Switzerland, 2021.
13. Droste, M.; Kuich, W.; Vogler, H. *Handbook of Weighted Automata*, 1st ed.; Springer Publishing Company, Incorporated, 2009.
14. Savitch, W.J. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences* **1970**, *4*, 177–192.
15. Abramsky, S.; Gabbay, D.M.; Maibaum, T.S.E., Eds. *Handbook of Logic in Computer Science*; Vol. 1–6, Oxford University Press: Oxford, 1992–2000.
16. Immerman, N. *Descriptive complexity*; Graduate texts in computer science, Springer, 1999.
17. Sipser, M. Borel sets and circuit complexity. In Proceedings of the Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing, New York, NY, USA, 1983; STOC '83, pp. 61–69.
18. Barrington, D.A.M.; Immerman, N.; Straubing, H. On Uniformity Within $\mathbf{NC}^1$. *Journal of Computer and System Sciences* **1990**, *41*, 274–306.
19. Chomsky, N. Three models for the description of language. *IRE Transactions on Information Theory* **1956**, *2*, 113–124.
20. Büchi, J.R. Weak Second-Order Arithmetic and Finite Automata. *Mathematical Logic Quarterly* **1960**, *6*, 66–92.
21. Büchi, J.R. On a Decision Method in Restricted Second Order Arithmetic. In Proceedings of the Logic, Methodology and Philosophy of Science. Stanford University Press, 1962, pp. 1–11.
22. Vardi, M.Y. The Complexity of Relational Query Languages (Extended Abstract). In Proceedings of the Proceedings of the 14th Annual ACM Symposium on Theory of Computing (STOC '82), 1982, pp. 137–146.
23. Immerman, N. Relational Queries Computable in Polynomial Time. *Information and Control* **1986**, *68*, 86–104.
24. Fagin, R. Generalized first-order spectra and polynomial-time recognizable sets. In *Complexity of Computation*; Karp, R.M., Ed.; American Mathematical Society, 1974; Vol. 7, *SIAM-AMS Proceedings*, pp. 43–73.
25. Immerman, N. Languages Which Capture Complexity Classes (Preliminary Report). In Proceedings of the Proceedings of the 15th Annual ACM Symposium on Theory of Computing (STOC '83), Boston, Massachusetts, 1983; pp. 347–354.
26. Immerman, N. Languages Which Capture Complexity Classes. *SIAM Journal on Computing* **1987**, *16*, 760–778.
27. Ebbinghaus, H.D.; Flum, J. *Finite Model Theory*; Springer-Verlag: Berlin, Heidelberg, 1995.
28. Fagin, R. Generalized First-Order Spectra and Polynomial-Time Recognizable Sets. *Complexity of Computation* **1974**, *7*, 43–73.
29. Stockmeyer, L.J. The Polynomial-Time Hierarchy. *Theoretical Computer Science* **1976**, *3*, 1–22.
30. Wrathall, C. Complete Sets and the Polynomial-Time Hierarchy. *Theoretical Computer Science* **1976**, *3*, 23–33.
31. For developers, D. Chomsky hierarchy. https://devopedia.org/images/article/210/7090.1571152901.jpg, 2026. Accessed: 2026-08-08.
32. Korenblat, K.; Ravve, E. Automatic Code Generator for Screen Based Systems. In Proceedings of the New Trends in Databases and Information Systems; Welzer, T.; Eder, J.; Podgorelec, Viliand Wrembel, R.; Ivanović, M.; Gamper, J.; Morzy, M.; Tzouramanis, T.; Darmont, J.; Kamišalić Latifić, A., Eds., Cham, 2019; pp. 253–265.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

33. Avitan, M.; Ravve, E.V.; Volkovich, Z. Assembly Function Recognition in Embedded Systems as an Optimization Problem. *Mathematics* **2024**, *12*, 658.
34. Reyes-Reina, D.; Sanchéz-Torres, J.M.; Rueda-Cáceres, I.M. Adoption of *AI* tools in software development: a systematic literature review. *Science of Computer Programming* **2026**, *254*.
35. Campesato, O. *Large Language Models: An Introduction*; De Gruyter / Mercury Learning and Information, 2024.
36. Zhang, A.; Lipton, Z.C.; Li, M.; Smola, A.J. *Dive into Deep Learning*; Cambridge University Press, 2023.
37. Mohsin, M.A.; et al. On the Fundamental Limits of LLMs at Scale. *arXiv preprint arXiv:2511.12869* **2025**.
38. Peng, B.; Narayanan, S.; Papadimitriou, C. On Limitations of the Transformer Architecture, 2024, [arXiv:stat.ML/2402.08164].
39. Karpowicz, M.P. On the Fundamental Impossibility of Hallucination Control in Large Language Models, 2026, [arXiv:stat.ML/2506.06382].
40. Melo, G.A.; Máximo, M.R.O.A.; Soma, N.Y.; Castro, P.A.L. Machines that halt resolve the undecidability of artificial intelligence alignment. *Scientific Reports* **2025**, *15*, 15591.
41. Asher, N.; Bhar, S.; Chaturvedi, A.; Hunter, J.; Paul, S. Limits for learning with language models. In Proceedings of the Proceedings of the 12th Joint Conference on Lexical and Computational Semantics (*SEM 2023), Toronto, Canada, 2023; pp. 236–248.
42. Kechris, A.S., The Borel Hierarchy. In *Classical Descriptive Set Theory*; Springer New York: New York, NY, 1995; pp. 167–178.
43. Dennunzio, A.; Formenti, E.; Provillard, J. Acceptance conditions for ω-languages. In Proceedings of the Proceedings of the 16th International Conference on Developments in Language Theory, Berlin, Heidelberg, 2012; DLT'12, p. 320–331.
44. Bombardelli, A.; Cimatti, A.; Tonetta, S.; Zamboni, M. Symbolic Model Checking of Relative Safety LTL Properties. In Proceedings of the Integrated Formal Methods - 18th International Conference, iFM 2023, Leiden, The Netherlands, November 13-16, 2023, Proceedings; Ábrahám, E.; Dubslaff, C., Eds. Springer, 2023, Vol. 14387, *Lecture Notes in Computer Science*, pp. 302–320.
45. Li, Z.; Hua, W.; Wang, H.; Zhu, H.; Zhang, Y. Formal-LLM: Integrating Formal Language and Natural Language for Controllable LLM-based Agents, 2024, [2402.00798].
46. Sanchez-Lengeling, B.; Reif, E.; Pearce, A.; Wiltschko, A. A Gentle Introduction to Graph Neural Networks. *Distill* **2021**.
47. Grohe, M. The Descriptive Complexity of Graph Neural Networks. *TheoretiCS* **2024**, *Volume 3*.
48. Abboud, R.; Ceylan, İ.İ.; Grohe, M.; Lukasiewicz, T. The Surprising Power of Graph Neural Networks with Random Node Initialization. In Proceedings of the Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21; Zhou, Z.H., Ed. IJCAI Organization, 2021, pp. 2112–2140.
49. Benedikt, M.; Lu, C.H.; Tan, T. Decidability of Graph Neural Networks via Logical Characterizations. *ACM Transactions on Computational Logic* **2026**, *27*, 1–65.
50. Ahvonen, V.; Heiman, D.; Kuusisto, A. Descriptive complexity for neural networks via Boolean networks. *Journal of Logic and Computation* **2026**, *36*.
51. Wan, X.; Xu, K.; Liao, X.; Jin, Y.; Chen, K.; Jin, X. Scalable and Efficient Full-Graph GNN Training for Large Graphs. *Proc. ACM Manag. Data* **2023**, *1*.
52. Li, X.; Qi, J.; Liu, H.; et al. ScaleGNN: Towards Scalable Graph Neural Networks via Adaptive High-order Neighboring Feature Fusion. In Proceedings of the Proceedings of the ACM Web Conference 2026, 2026, pp. 980–991.
53. Adam-Day, S.; Iliant, T.M.; Ceylan, I.I. Zero-One Laws of Graph Neural Networks. In Proceedings of the Advances in Neural Information Processing Systems, 2023, Vol. 36, pp. 70733–70756.
54. Liu, J.; Mao, H.; Chen, Z.; Zhao, T.; Shah, N.; Tang, J. Towards Neural Scaling Laws on Graphs. In Proceedings of the Proceedings of the Third Learning on Graphs Conference. PMLR, 2025, Vol. 269, *Proceedings of Machine Learning Research*, pp. 44:1–44:22.
55. Wang, C.; Ross, C.; Kuo, Y.; Katz, B.; Barbu, A. Learning a natural-language to LTL executable semantic parser for grounded robotics. *CoRR* **2020**, *abs/2008.03277*.
56. Liu, J.X.; Yang, Z.; Schornstein, B.; Liang, S.; Idrees, I.; Tellex, S.; Shah, A. Lang2LTL: Translating Natural Language Commands to Temporal Specification with Large Language Models. In Proceedings of the Workshop on Language and Robotics at CoRL 2022, 2022.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

57. Liu, Z.; Shah, A.; Bisk, Y. Grounding Complex Natural Language Commands for Temporal Tasks in Unseen Environments. In Proceedings of the Proceedings of the 6th Conference on Robot Learning; Asfour, T.; Borjorn, K.; Kober, J., Eds. PMLR, 2023, Vol. 229, *Proceedings of Machine Learning Research*, pp. 1645–1655.
58. Pan, J.; Chou, G.; Berenson, D. Data-Efficient Learning of Natural Language to Linear Temporal Logic Translators for Robot Task Specification. In Proceedings of the 2023 IEEE International Conference on Robotics and Automation (ICRA), 2023, pp. 11554–11561.
59. Yang, K.; Wang, H.; Yang, K. From Language to Logic: A Bi-Level Framework for Structured Reasoning, 2025, [arXiv:cs.AI/2507.08501].
60. Liu, J. Few-Shot Natural Language to First-Order Logic Translation via Code Generation. In Proceedings of the Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers); Chiruzzo, L.; Ritter, A.; Wang, L., Eds., Albuquerque, New Mexico, 2025; pp. 10939–10960.
61. Putra, R.R.; Basuki, R.S.P.; Cheng, Y.; Gao, P. NL2Logic: AST-Guided Translation of Natural Language into First-Order Logic with Large Language Models. In Proceedings of the Findings of the Association for Computational Linguistics: EACL 2026. Association for Computational Linguistics, 2026, p. 6035–6051.
62. *The Holy Bible: English Standard Version*; Crossway Bibles: Wheaton, IL, 2001. John 1:1.
63. Herrington, J. *Code Generation in Action*; Manning Publications Co.: Greenwich, CT, USA, 2003.
64. Object Managment Group, . MDA Guide Version 1.0.1. https://www.omg.org/news/meetings/workshops/UML_2003_Manual/00-2_MDA_Guide_v1.0.1.pdf. Accessed: June 2003.
65. Atkinson, C.; Kühne, T. Model-Driven Development: A Metamodeling Foundation. *IEEE Softw.* **2003**, *20*, 36–41.
66. Object Managment Group, . OMG Unified Modeling Language. https://www.omg.org/spec/UML/2.2/Superstructure/PDF. Accessed: 2009.
67. Kraemer, F. Engineering Android Applications Based on UML Activities. In Proceedings of the Model Driven Engineering Languages and Systems; Whittle, J.; Clark, T.; Kühne, T., Eds., Berlin, Heidelberg, 2011; pp. 183–197.
68. Monte-Mor, J.; Ferreira, E.; Campos, H.; da Cunha, A.; Dias, L. Applying MDA Approach to Create Graphical User Interfaces. In Proceedings of the 2011 Eighth International Conference on Information Technology: New Generations, 2011, pp. 766–771.
69. Sabraoui, A.; Koutbi, M.E.; Khriss, I. GUI code generation for Android applications using a MDA approach. *2012 IEEE International Conference on Complex Systems (ICCS)* **2012**, pp. 1–6.
70. Lachgar, M.; Abdali, A. Modeling and Generating the User Interface of Mobile Devices and Web Development with DSL. *Journal of Theoretical and Applied Information Technology* **2015**, *72*, 124–132.
71. Benouda, H.; Essbai, R.; Azizi, M.; Moussaoui, M. Modeling and Code Generation of Android Applications Using Acceleo. *International Journal of Software Engineering and Its Applications* **2016**, *10*, 83–94.
72. Benouda, H.; Azizi, M.; Esbai, R.; Moussaoui, M. MDA Approach to Automate Code Generation for Mobile Applications. In Proceedings of the Mobile and Wireless Technologies 2016; Kim, K.J.; Wattanapongsakorn, N.; Joukov, N., Eds., Singapore, 2016; pp. 241–250.
73. Korenblat, K.; Ravve, E.V. On Consistency of Graphically Defined Specifications. In *Model Theory, Computer Science, and Graph Polynomials: Festschrift in Honor of Johann A. Makowsky*; Meer, K.; Rabinovich, A.; Ravve, E.V.; Villaveces, A., Eds.; Trends in Mathematics, Springer Nature Switzerland, 2025; pp. 465–483.
74. McCarthy, J. Towards a Mathematical Science of Computation. In Proceedings of the Proceedings of the International Federation for Information Processing (IFIP) Congress. North-Holland, 1962, pp. 21–28.
75. Ye, Y.; Pang, P.; Zhang, T.; Huang, H. GNN-Coder: Boosting Semantic Code Retrieval with Combined GNN and Transformer, 2025.
76. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, L.; Polosukhin, I. Attention is all you need. In Proceedings of the Advances in Neural Information Processing Systems (NeurIPS), 2017, Vol. 30.
77. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, D.; Jiang, T.; et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In Proceedings of the Findings of the Association for Computational Linguistics: EMNLP 2020, 2020, pp. 1536–1547.
78. Tiftikci, M.; Sharifloo, A.M.; Mezini, M. Deep Graph-Language Fusion for Structure-Aware Code Generation. In Proceedings of the Proceedings of the 2026 IEEE/ACM Third International Conference on AI Foundation Models and Software Engineering, New York, NY, USA, 2026; FORGE '26, p. 67–71.
79. Beck, K. *Test-Driven Development: By Example*; Addison-Wesley Professional, 2002.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org | NOT PEER-REVIEWED | Posted: 17 November 2026 https://doi.org/10.20944/preprints202610.1999.v2

80. Ammann, P.; Offutt, J. *Introduction to Software Testing*, 2nd ed.; Cambridge University Press: Cambridge, 2016.
81. TestDriven. What is Test-Driven Development? https://testdriven.io/test-driven-development/, 2026. Accessed: 2026-08-08.
82. Medium. Deep dive into Test-Driven Development (TDD) with AI. https://medium.com/@ignatovich.dm/deep-dive-into-test-driven-development-tdd-with-ai-edf86f07a40c, 2026. Accessed: 2026-08-08.
83. Mathews, N.S.; Nagappan, M. Test-Driven Development and LLM-based Code Generation. In Proceedings of the Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, New York, NY, USA, 2024; ASE '24, p. 1583–1594.
84. Chen, X.; Tao, Z.; Zhang, K.; Zhou, C.; Zhang, X.; Gu, W.; He, Y.; Zhang, M.; Cai, X.; Zhao, H.; et al. Revisit Self-Debugging with Self-Generated Tests for Code Generation. In Proceedings of the Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers); Che, W.; Nabende, J.; Shutova, E.; Pilehvar, M.T., Eds., Vienna, Austria, 2025; pp. 18003–18023.
85. Yu, H.; Li, K.; Li, J.; Chai, H.; Yuan, Y.; He, R.; Wei, J. TDD-Agent: Test-Driven Reasoning for Code Generation, 2026, [arXiv:cs.SE/2608.16742].
86. Li, Z.; Wang, Z. GUARD: A GNN-Based Tool for Automated UnitTest Case Generation and Code Defect Prediction. In Proceedings of the 2025 6th International Conference on Computer Engineering and Application (ICCEA), 2025, pp. 819–824.
87. Holzmann, G.J. *The SPIN Model Checker: Primer and Reference Manual*; Addison-Wesley Professional, 2003.
88. Korenblat, K.; Ravve, E.; Lemberg, D.; Kleiman, E. AI-Based Incremental Code Development Automation using the TDD. in preparation.
89. Makowsky, J.A. From a Zoo to a Zoology: Towards a General Theory of Graph Polynomials. *Theory of Computing Systems* **2008**, *43*, 542–562.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.