International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 10 Issue: 11 | Nov - 2026 SJIF Rating: 8.659 ISSN: 2582-3930

Quantum Inspired AI Code Optimizer: A Verified, Evidence-Based Platform for Optimizing and Translating Programs

Devaraj F V[1], Shamith R Heggod[2], Shreehari G K[3], Shreerama A Hegde[4], Suhas B[5]

[1]Assistant Professor, Department of Computer Science and Engineering, Jawaharlal Nehru New College of Engineering, Shivamogga, Karnataka, India

[2,3,4,5]Seventh Semester Students, Department of Computer Science and Engineering, Jawaharlal Nehru New College of Engineering, Shivamogga (Affiliated to Visvesvaraya Technological University, Belagavi), Karnataka, India

---

Abstract - Programs written first are often slow, and the tools that are available either only advise the programmer or change the code without checking it. This paper presents the Quantum Inspired AI Code Optimizer (QIACO), a web platform for Python, C++, C, Java and Go. The platform checks the syntax with the host compiler, derives time and space complexity from the syntax tree, rewrites 100 families of well-known algorithms with a master abstract syntax tree (AST) engine, and searches combinations of rewrite rules with a quantum-inspired evolutionary algorithm that is supported by a neural inefficiency detector and a Q-learning agent. A candidate is accepted only after differential testing against the original in an isolated process, and every request ends in a clear evidence-based state, so that code that is already good, code that cannot be verified and code that fails validation are reported honestly. Of 16 programs sent to the running system, 11 were improved and validated; in a benchmark of 200 requests on 100 programs in two languages, 153 returned a validated optimization. Of 77 automated back-end tests, 73 passed.

Key Words: Code Optimization, Time Complexity, Quantum-Inspired Evolutionary Algorithm, Q-Learning, Differential Testing, Program Translation.

1. INTRODUCTION

A student who has written a correct but slow program has three options, and each has a shortcoming. A compiler optimizes the machine code but never changes the algorithm, so a program with two nested loops over the same array is still quadratic after compilation. A language model can suggest a better algorithm, but its answer may behave differently from the original in cases that nobody tried. A human reviewer is slow and is not available for every program. What is needed is a tool that proposes a better algorithm and proves, as far as testing can prove, that it is safe.

The Quantum Inspired AI Code Optimizer was built to close this gap. It is a web application with a FastAPI back end and a React and TypeScript front end. The user pastes a function or uploads a file, chooses the input language and, if wanted, another output language, and presses Optimize Code. The system answers with a clear state and never returns changed code that has not passed validation. The word quantum-inspired refers only to the search method: the system runs on an ordinary computer and makes no claim of a quantum speed-up.

The motivation for the project follows from five practical problems observed with existing tools, summarised in Table 1.

Table -1: Problems addressed and the corresponding design response

Problem with existing tools	Response of QIACO
Compilers (for example GCC at -O2) never change the algorithm, so an $O(n^2)$ program stays $O(n^2)$.	Structural rules and verified algorithm templates replace the algorithm, for example a nested loop by a hash table.
Language models write code that looks correct but may behave differently.	Every candidate is run against the original on generated inputs in an isolated process; a candidate that differs is rejected.
A tool that only compares the input with the output cannot tell a program that is already good from one it failed to improve.	Clear evidence-based states are returned; Already Optimized is given only when every rule has been evaluated and none applies.
Tools handle one language, so moving a program to another language needs manual rewriting.	Five languages are supported, and the optimized code can be returned in another output language through an intermediate representation.
Learning components are often presented as if they decide the result.	A neural detector and a Q-learning agent only advise; the validator decides, and their real evaluation numbers are reported.

The remainder of this paper is organised as follows. Section 2 reviews related work. Section 3 describes the proposed system. Section 4 reports the results of the automated tests and of the measured example programs. Section 5 concludes the paper.

2. LITERATURE SURVEY

A set of earlier works was reviewed to establish the approaches that are available for each component of such a system: the search over candidate programs, the generation of candidates, the learning guides, the checking of candidates and the translation between languages.

Han and Kim [1] introduce the quantum-inspired evolutionary algorithm (QIEA), in which every individual is a string of quantum bits, each holding a probability amplitude instead of a fixed 0 or 1. An individual is observed to obtain a concrete solution, and rotation gates move the amplitudes towards the best solution found. The algorithm reached good solutions of the knapsack problem with a small population and runs on a conventional computer. It is the model of the search used in the proposed system, where each bit decides whether a rewrite rule is applied.

Watkins and Dayan [2] prove the convergence of Q-learning, in which the value of an action in a state is improved from the observed reward, and Sutton and Barto [3] describe reinforcement learning as a Markov decision process. The proposed system uses the tabular form of Q-learning to learn which rule to try first.

Schkufza, Sharma and Aiken [4] replace the exhaustive search of superoptimization by a stochastic search over machine-code sequences, guided by test cases and a cost function, and Bunel et al. [5] add a learned proposal distribution. Both show that tests and a learned guide can steer a search towards fast code, but both work on short instruction sequences and not on the algorithms of a source-code program.

Madaan et al. [6] and Chen et al. [7] study language models for performance-improving edits and for code generation. Generated programs can give real speed-ups, but not every edit is correct, and filtering by tests raises the success rate. Li et al. [8] generate many candidate programs and filter them by execution, which is close to the structure of the proposed system, in which candidate rewrites are filtered by the validator.

Lachaux et al. [9] translate functions between C++, Java and Python and show that translated code which looks correct can be wrong, so that execution is the reliable judge. Aho, Lam, Sethi and Ullman [10] describe the classical optimizations of a compiler, which preserve the algorithm by construction and therefore improve only constant factors. McKeeman [11] and Claessen and Hughes [12] introduce differential testing and property testing on generated inputs, the two ideas on which the validator of the proposed system is based.

Taken together, the reviewed literature shows that the components required for a trustworthy code optimizer exist separately, but not in one interactive tool that (i) changes the algorithm by structural rules and verified templates, (ii) searches rule combinations with a quantum-inspired algorithm guided by learned components, (iii) supports several languages with honest capability levels and (iv) refuses to show any candidate that has not been validated and says so when no safe improvement exists. This combination is the contribution of the present work.

3. PROPOSED SYSTEM

The primary objective of the system is to convert a program that a user pastes into a validated program with a lower cost, without any candidate reaching the user unless it has been shown to behave like the original. Fig. 1 shows the four layers of the architecture: the browser front end with a code editor, language selectors, result tabs and a dialog for the state Already Optimized; the FastAPI back end with eight typed endpoints, user authentication and the status engine; the analysis, search and learning core; and the execution layer with the sandbox, the validator, the benchmark runner and the translator.

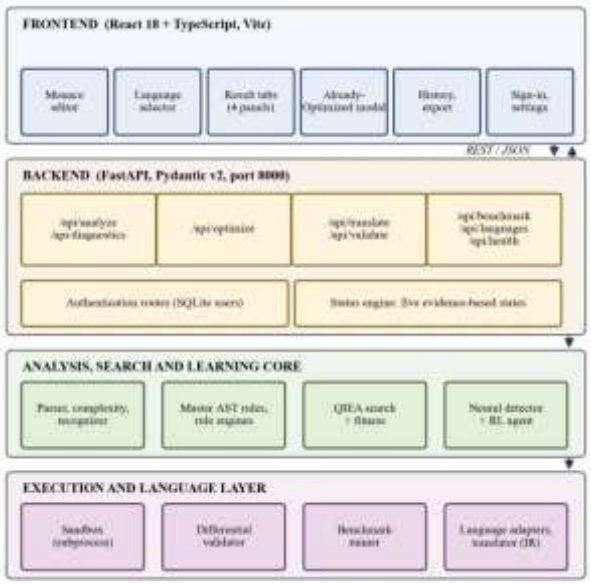


Fig -1: Architecture of the Quantum Inspired AI Code Optimizer

3.1 Pipeline stages

Table 2 lists the eight stages of the pipeline for a request, the component responsible for each stage and the type of the stage. The pipeline of Fig. 2 starts with a syntax check by the real compiler or interpreter of the host; a syntax error ends it with the state Analysis Incomplete and the line of the error. The master AST engine then tries to recognise the program as one of 100 algorithm families. The complexity analyzer derives the time and space class with its reasoning and a confidence label, and the structural recognizer finds patterns such as nested loops, bubble sort, repeated linear search or string concatenation in a loop from the syntax tree, never from names. The remaining stages follow in order. Rules are selected and flagged by the ten Python rules, the engines for the other

languages, the learned rules and the neural detector, which only advises; the Q-learning agent orders the selected rules, and the quantum-inspired search combines them. Every candidate is then tested by the validator, and the status engine takes the final decision.

Table -2: Stages of the pipeline, responsible component and type

#	Stage	Responsible component	Type
1	Syntax check and diagnostics	Host compiler or interpreter; Python ast module	Deterministic
2	Complexity and pattern analysis	Complexity analyzer with reasoning; structural recognizer	Deterministic
3	Master AST rules	Rewrites of 100 algorithm families, no stored code	Deterministic
4	Rule selection and flags	Ten Python rules, engines, learned rules; neural detector (advisory)	Deterministic / learned
5	Rule ordering	Tabular Q-learning agent	Learned, advisory
6	Search over rule combinations	Quantum-inspired evolutionary algorithm (QIEA)	Search-based
7	Validation	Differential tests in a sandboxed subprocess	Deterministic
8	Decision	Status engine	Deterministic

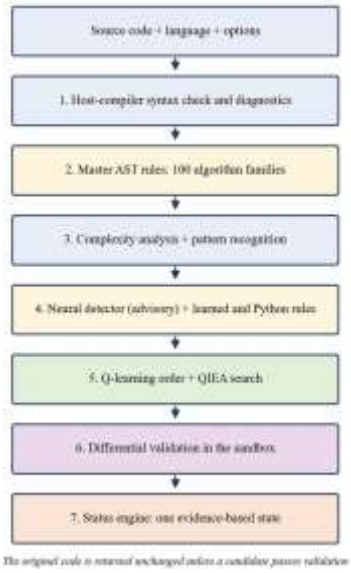


Fig -2: Stages of the optimization pipeline

3.2 Rules and algorithm templates

The master AST engine rewrites 100 algorithm families for C, C++, Java, Python and Go. It stores no answer code: it reads the function and its parameters from the syntax tree and synthesises a verified, better implementation in the language of the user or in the chosen output language, so a renamed or reformatted program is still recognised. For Python, ten further rules are evaluated against the syntax tree: bubble sort to the built-in sort, repeated list search to a set or a dictionary, string concatenation to a join, loops to comprehensions, memoization of recursion, removal of redundant computation, built-in functions in place of manual loops, generators for large data and default handling of dictionary keys. A separate engine handles C and C++, and a polyglot engine covers Java and Go. An optional language-model module can derive a complexity or an optimization when the local rules fail, and a learned-rules store keeps a structural fingerprint of every accepted rewrite so that the same pattern is answered locally the next time. In all experiments of this paper the model keys were empty. When another output language is requested, the optimized code is translated through a universal intermediate representation.

3.3 Quantum-inspired search with learned guides

When two or more rules apply, or when the user asks for Deep Optimization, the rules are searched as combinations (Fig. 3). As the figure shows, the population is initialised with equal amplitudes, and the loop of observation, evaluation and rotation repeats until the generation limit is reached, after which the best rule combination is returned.

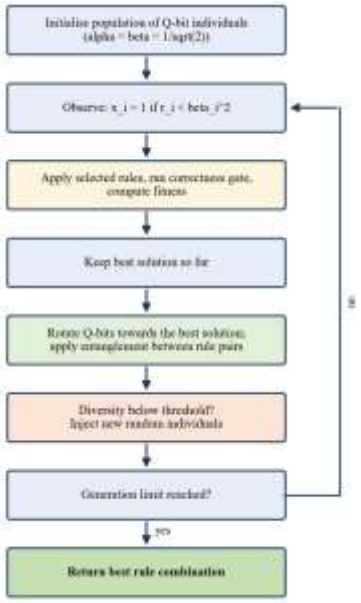


Fig -3: Flowchart of the quantum-inspired evolutionary search

Each applicable rule has one quantum bit with amplitudes (α, β) and $\alpha^2 + \beta^2 = 1$. A solution X is observed by setting $x_i = 1$ with probability β_i^2, the rules of X are applied, and the fitness is computed as

$$F(X) = G(X) \cdot (0.30\Delta C + 0.25\Delta S + 0.15\Delta M + 0.15R - 0.15K) \quad (1)$$

where $G(X)$ is 1 only if the transformed program passes the correctness gate and 0 otherwise, ΔC, ΔS and ΔM are the gains in complexity class, speed and memory, R is readability and K a risk penalty. The

qubits are then rotated towards the best solution by a rotation gate with a step of 0.06 rad, entangled pairs are coupled, and fresh individuals are injected when the diversity of the population is too low. The search is seeded and therefore reproducible. A neural network with 16 syntax-tree inputs, 64 ReLU units and 10 sigmoid outputs flags likely inefficiency patterns, and a tabular Q-learning agent ($\alpha = 0.1$, $\gamma = 0.9$) learns which rule to try first from rewards of +1.5, +0.5, −0.1 and −1.0. Both learned components are advisory.

3.4 Validation and the status engine

The validator is the stage that decides whether a candidate may be shown. It extracts the signature of the function, generates unit, edge-case, boundary and property tests (empty and single-element inputs, duplicates, negative numbers, values up to 100 000, sorted and reverse-sorted sequences), and runs the original and the candidate on identical copies of the inputs in a separate process after a static security scan. The process has a wall-clock limit of 5 s, a clean environment without secrets, a disposable directory and an output cap; memory and CPU limits are set where the operating system provides them. Return values, changes to arguments and exception types must agree.

The status engine (Fig. 4, Table 3) turns the evidence into exactly one state. The state Already Optimized is given only when parsing succeeded and every rule has evaluated its preconditions and found none applicable; the original code is preserved byte for byte and a dialog says that no claim of global optimality is made. Two further states, Optimization Fallback Required and Complexity Fallback Required, are returned when no local rule matched or the static analysis gave no complexity and no model is configured; they also leave the code unchanged.

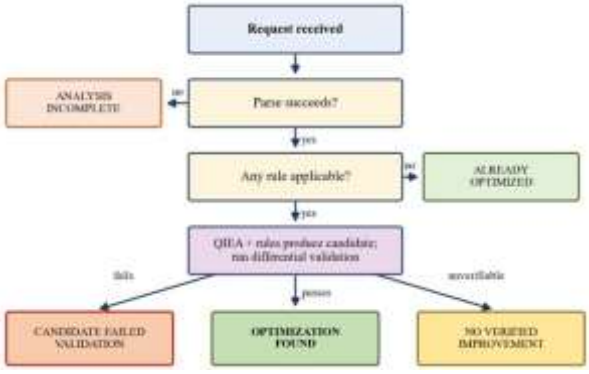


Fig -4: Decision of the status engine

Table -3: The result states

State	Trigger	Code returned
Optimization Found	A candidate passed all validation tests	Optimized code with before and after complexity
Already Optimized	All rules evaluated, none applicable	Original, unchanged
No Verified Improvement	Candidate only partly verified	Original, with explanation
Analysis Incomplete	Syntax error, unsupported language or time-out	Original, with the errors
Candidate Failed Validation	A candidate disagreed with the original	Original; candidate rejected
Fallback Required	No local rule matched or no complexity could be derived, and no model is configured	Original, unchanged

3.5 User interface

Fig. 5 shows the result page for a C++ two-sum program: the status badge, the applied rule, the line Differential Validation: VALIDATED with 5 of 5 tests, the optimized code and cards that show the algorithm identified, the time complexity ($O(n^2)$ to $O(n)$), the space complexity ($O(1)$ to $O(n)$) and the number of code lines. The cards state the price of a faster algorithm. Further tabs show a side-by-side diff, the asymptotic comparison and the explanation with the full search trace. The history of the last 50 analyses is kept in the browser, and a report can be exported as JSON or Markdown.

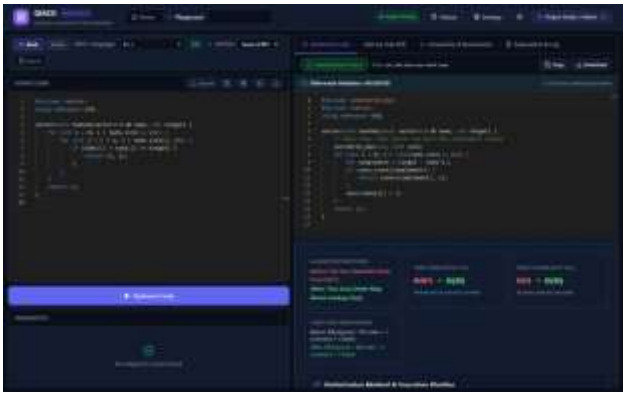


Fig -5: Result page: C++ two sum, $O(n^2)$ to $O(n)$, validated

4. RESULTS

The system was implemented in Python with FastAPI and in TypeScript with React. All experiments were run on a Windows 11 laptop with Python 3.13, the MinGW g++ and gcc compilers and a Java compiler; the Go compiler was not installed, and the language-model keys were empty, so every result comes from the local engine.

The back end has 77 automated tests in ten files that cover analysis, the REST endpoints, the sandbox and benchmark runner, C++ optimization, 22 varied

programs, the fallback of the language-model provider, the rules, the quantum and learning components, the status engine, and translation. In one run of 64 seconds, 73 passed and 4 failed; the four failures are assertions in the C++ test file that still expect the rule names and the spelling of the earlier engine. In addition, 16 example programs were sent to the optimize endpoint of the running server: ten in Python, four in C++, and one each in Java and Go. Table 4 lists the state, the time complexity reported before and after, and the validation tests passed.

Table -4: Results for 16 example programs sent to the running system

No.	Program	Lang.	State	Time complexity	Tests passed
1	Bubble sort	Python	Optimization Found	$O(N^2) \rightarrow O(N \log N)$	5/5
2	Repeated search in list	Python	Optimization Found	$O(n) \rightarrow O(1)$	9/9
3	String concatenation in loop	Python	Optimization Found	$O(n) \rightarrow O(n)$	25/25
4	Recursive Fibonacci	Python	Optimization Found	$O(2^N) \rightarrow O(N)$	5/5
5	Manual list building	Python	Optimization Found	$O(n) \rightarrow O(1)$	25/25
6	Manual sum	Python	Already Optimized	O(n) (unchanged)	-
7	Dictionary default handling	Python	Fallback Required	O(n) (unchanged)	-
8	Two sum (nested loops)	Python	Optimization Found	$O(N^2) \rightarrow O(N)$	5/5
9	Already optimal (sorted)	Python	Already Optimized	O(1) (unchanged)	-
10	Syntax error	Python	Analysis Incomplete	-	-
11	C++ bubble sort	C++	Optimization Found	$O(N^2) \rightarrow O(N \log N)$	5/5
12	C++ two sum	C++	Optimization Found	$O(N^2) \rightarrow O(N)$	5/5
13	C++ Fibonacci	C++	Optimization Found	$O(2^N) \rightarrow O(N)$	5/5
14	C++ breadth-first search	C++	Already Optimized	$O(n^2)$ (unchanged)	-
15	Java bubble sort	Java	Optimization Found	$O(N^2) \rightarrow O(N \log N)$	5/5
16	Go bubble sort	Go	Optimization Found	$O(N^2) \rightarrow O(N \log N)$	5/5

Eleven programs were improved and validated, with every validation test passing, among them bubble sort in Python, C++, Java and Go ($O(n^2)$ to $O(n \log n)$), recursive Fibonacci in Python and C++ ($O(2^n)$ to $O(n)$) and two sum in Python and C++ ($O(n^2)$ to $O(n)$). Two further Python programs were rewritten by the list-comprehension rule; this removes the cost of append, which is a constant-factor gain, and the O(1) printed by the static estimator after the rewrite should not be read as a change of class. Three programs, the call to *sorted*, a manual sum and a breadth-first search over an adjacency matrix, were left unchanged with the state Already Optimized. For the breadth-first search (Fig. 6) this is correct, because any traversal that reads a matrix must read it. For the manual sum it shows a limit of the rule catalogue, since the built-in sum would be shorter. One word-counting function received the fallback state because no local rule produced a verified rewrite and no model was configured, and its original function was preserved. A program with a syntax error returned Analysis Incomplete with the position of the error.

A larger benchmark used the 100 programs of the project (searching and sorting, number and string problems, matrices, graphs, dynamic programming, linked lists, trees and interview problems) in Python and in C++, 200 requests in all, each in its own process with a limit of 40 seconds (Table 5). 153 requests (76 percent) returned a validated optimization, 31 were left unchanged as already optimal, and 9 Python programs reached the fallback state. 6 requests did not finish within the limit, one in Python and five in C++ (dynamic-programming and graph programs), and one C++ program returned Analysis Incomplete. A request that cannot finish is the most serious weakness of the present build, and in the web application it blocks the server until it ends.

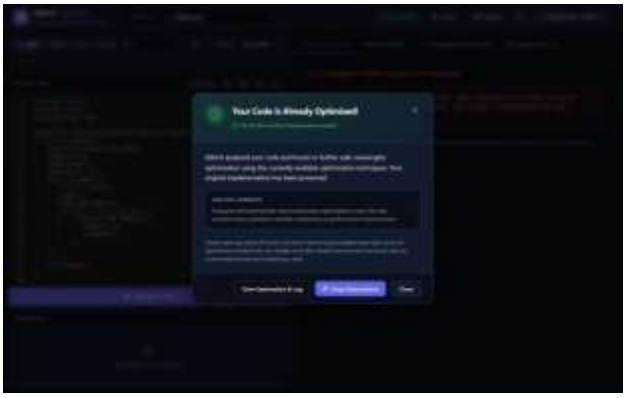


Fig -6: Already Optimized dialog for a breadth-first search over an adjacency matrix

Table -5: Outcome of the 200 benchmark requests

Outcome	Python (100)	C++ (100)	Both
Optimization found, validated	80	73	153
Already optimized	10	21	31
Fallback required	9	0	9
No answer within 40 s	1	5	6
Analysis incomplete	0	1	1

The neural detector was trained on 160 generated programs and tested on 40 held-out programs (Table 6). Its precision is 1.00 but its macro recall is only 0.20, because only two of ten patterns were found on every test sample. The detector is therefore a weak guide, and it is used only to write flags to the search log. The Q-learning table in the project records 429

episodes and has learned a preference for the bubble-sort rule (value 11.6) over the set rule (4.7) and the join rule (2.4) for a program without special structure; the preference orders the rules but never replaces validation.

Table -6: Evaluation of the neural detector on the held-out test split

Measure	Value	Measure	Value
Training programs	160	Macro precision	1.00
Test programs	40	Macro recall	0.20
Epochs	15	Macro F1 score	0.20
Patterns (labels)	10	Exact-match accuracy	0.275

The limitations of the present system are as follows. Complexity is estimated statically and is stated with a confidence label; it is not a proof. Verification by generated inputs is evidence and not a proof of equivalence, and an equivalent algorithm may differ in cases that were not generated, for example the pair returned when no solution exists in two sum. Programs outside the 100 families and the ten Python rules receive a fallback state when no model is configured. Some requests do not finish, four unit tests are out of date, the Go compiler was not installed, and on Windows the memory limit of the sandbox is not enforced.

5. CONCLUSION

The Quantum Inspired AI Code Optimizer demonstrates that a master AST engine and a quantum-inspired search over rewrite rules, supported by structural analysis, a neural detector and a Q-learning agent and followed by differential validation in a sandbox, can return optimized programs in several languages and can state honestly when no safe improvement exists. Eleven of 16 measured programs were improved and validated, three were correctly left unchanged, and 153 of 200 benchmark requests returned a validated optimization.

Future work includes a hard time limit for every request, an update of the failing tests, deeper deterministic rewrites for Java and Go, a larger and more varied training set for the neural detector, a combination of the static estimate with measured growth, stronger isolation of the sandbox on Windows, and symbolic checks that extend the validation beyond generated inputs.

ACKNOWLEDGEMENT

We express our sincere gratitude to our guide, Mr. Devaraj F V, Assistant Professor, Department of Computer Science and Engineering, for the suggestions and support provided during every stage of this work. We thank our project coordinators, Nandeesh and Shreedevi, for their continuous encouragement and guidance. We also convey our deep sense of gratitude to the Head of the Department and to the Principal of Jawaharlal Nehru New College of Engineering, Shivamogga, for providing the resources necessary to carry out this project.

REFERENCES

[1] K.-H. Han and J.-H. Kim, “Quantum-inspired evolutionary algorithm for a class of combinatorial optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 6, pp. 580–593, 2002.

[2] C. J. C. H. Watkins and P. Dayan, “Q-learning,” *Machine Learning*, vol. 8, no. 3–4, pp. 279–292, 1992.

[3] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, MA, USA: MIT Press, 2018.

[4] E. Schkufza, R. Sharma and A. Aiken, “Stochastic superoptimization,” in *Proc. ASPLOS*, 2013, pp. 305–316.

[5] R. Bunel, A. Desmaison, M. P. Kumar, P. H. S. Torr and P. Kohli, “Learning to superoptimize programs,” in *Proc. ICLR*, 2017.

[6] A. Madaan et al., “Learning performance-improving code edits,” in *Proc. ICLR*, 2024.

[7] M. Chen et al., “Evaluating large language models trained on code,” *arXiv:2107.03374*, 2021.

[8] Y. Li et al., “Competition-level code generation with AlphaCode,” *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[9] M.-A. Lachaux, B. Roziere, L. Chanussot and G. Lample, “Unsupervised translation of programming languages,” in *Proc. NeurIPS*, 2020.

[10] A. V. Aho, M. S. Lam, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Boston, MA, USA: Addison-Wesley, 2006.

[11] W. M. McKeeman, “Differential testing for software,” *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[12] K. Claessen and J. Hughes, “QuickCheck: a lightweight tool for random testing of Haskell programs,” in *Proc. ICFP*, 2000, pp. 268–279.