TECHNICAL PAPER · B

Detection of AI-Generated Code and Embedded AI Agents in Source Projects: A Survey of Method Classes

A technical overview of the principal approaches to distinguishing machine-generated from human-written source code, and to identifying code that instantiates autonomous AI agents at runtime.

Draft · 27 May 2026

ABSTRACT

Two related but distinct detection problems arise in the analysis of contemporary source projects. The first is **provenance detection**: determining whether a given fragment of source code was written by a human or generated by a large language model. The second is **agent detection**: determining whether code, regardless of who or what wrote it, constructs or invokes an autonomous AI agent during execution. This paper surveys the principal classes of technique applicable to each problem. For provenance detection it covers supervised stylometric classification, zero-shot probability-curvature methods, perplexity scoring, and fine-tuned neural classifiers; it then sets out the measurable signals on which all of these rest — stylistic regularity, comment characteristics, token-level predictability, structural features, and idiom — together with documented accuracy ranges and failure modes. For agent detection it covers static identification of agent-framework dependencies, dynamic-execution constructs, and data-flow tracing. The paper is descriptive: it characterises the space of methods reported in the literature and does not prescribe a particular implementation.

Ask CodeDelta

01 Problem Statement and Scope

The proliferation of large language models capable of producing source code has introduced two questions that did not previously require automated tooling. Given a body of source code:

1. Which portions, if any, were generated by a language model rather than written by a human author? This is a question of *provenance*.
2. Which portions, if any, will instantiate or drive an autonomous AI agent when executed — for instance by calling a model-inference API in a loop, or by constructing an agent object from an agent framework? This is a question of *runtime behavior*.

These two questions are independent. Human-written code may construct an AI agent; AI-generated code may be entirely static and contain no agent logic. A complete analysis treats them separately, using different methods, and reports them as different findings.

This paper restricts itself to the detection methods themselves. It does not address what should be done with a detection result, nor does it make any claim about the prevalence, desirability, or risk of either AI-generated code or embedded agents. Those are questions of policy, not of technique.

02 Provenance Detection: The General Framing

Provenance detection is an instance of authorship attribution, a field that substantially predates large language models. Classical authorship attribution treats the problem as supervised classification: a body of code is converted into a feature vector, and a model trained on labeled examples predicts the author [1]. The same framing applies when the candidate "authors" are reduced to two classes, human and machine.

The foundational work in code authorship attribution established that programmers leave consistent, measurable stylistic traces in their code — a "coding fingerprint" derived from lexical, layout, and structural features [1]. The Code Stylometry Feature Set introduced in that work demonstrated that such features could attribute authorship among large candidate pools with high accuracy. Subsequent surveys have organized the feature space into broad categories [2]:

- **Lexical features** — frequencies of characters, keywords, operators, and identifiers; punctuation patterns; naming conventions [3].
- **Layout and formatting features** — indentation style, brace placement, whitespace habits, and commenting patterns.

- **Structural and syntactic features** — properties derived from the abstract syntax tree or control-flow graph, such as nesting-depth distribution and branching factor.

The application of this framing to the human-versus-machine question is recent but now well-represented in the literature, with several distinct methodological families.

03 Provenance Detection: Method Classes

3.1 Supervised stylometric classification

The most direct approach trains a conventional machine-learning classifier (logistic regression, support vector machine, random forest, or gradient-boosted trees are all reported) on engineered stylometric features extracted from a labeled corpus of human and machine code. A recent multilingual study reported a single classifier achieving an average F1 score of 84.1% across ten programming languages, trained on an open dataset of over 120,000 labeled snippets [3]. This family has the advantage of interpretability — the contributing features can be inspected — and the disadvantage of depending on a feature-engineering process whose effectiveness is dataset-dependent [2]. The choice of classifier also matters: a study constructing a labeled corpus of human-written and AI-generated Python code reported that, among the baseline detectors tested, a Bayesian classifier outperformed the alternatives [12].

A notable empirical finding from this family is that not all feature categories contribute equally. At least one study reported that abstract-syntax-tree and control-flow-graph structural metrics shifted classifier decision boundaries only marginally, while the presence or absence of comments was a comparatively strong signal for attribution [4]. This suggests that surface features can dominate deep structural features in practice, though the result is specific to the datasets and models studied.

3.2 Zero-shot probability-curvature methods

A second family avoids training a dedicated classifier. Building on the DetectGPT method developed for natural-language text [5], these approaches exploit a statistical property of model-generated text: such text tends to occupy regions of negative curvature in the generating model's log-probability surface. The method perturbs the candidate text (for code, by masking and refilling lines using a mask-filling model), measures the change in log-probability, and uses the curvature to discriminate generated from human content. The reported advantage is that it

requires no labeled training corpus; the reported limitation is sensitivity to the choice of perturbation model and to adversarial editing of the candidate [5].

3.3 Perplexity-based methods

A third family uses the perplexity of the candidate code under one or more reference language models as the discriminating signal. Code generated by a model tends to exhibit lower perplexity under that model (or a related one) than human-written code, because it was sampled from a similar distribution. Perplexity-based detection of AI-generated code assignments has been studied in the educational-integrity context [6]. This family sits between the previous two: it uses model probabilities like the curvature methods but reduces them to a simpler scalar statistic.

3.4 Neural sequence classifiers

A fourth family fine-tunes a pre-trained code language model (of the BERT or CodeT5 lineage) directly as a binary classifier, bypassing manual feature engineering. Studies in this family report accuracy exceeding the classical stylometric baselines — one reported a modified CodeT5 classifier reaching above 97% on its dataset [4]. The trade-off is reduced interpretability and increased computational cost relative to the engineered-feature approach.

04 Discriminating Signals: What Separates Machine from Human Code

Every method in Section 03, whatever its statistical machinery, ultimately rests on measurable differences between the two populations of code. It is worth setting out what those differences are, because they explain why detection is possible at all and why it is imperfect. The literature reports several signal categories that carry discriminative weight. None is decisive in isolation; their value is collective.

4.1 Regularity and low variance

Machine-generated code tends to exhibit lower variance across a range of stylistic dimensions than a human population does. Indentation and brace placement are more uniform; identifier-naming follows more consistent schemes; the distribution of function lengths and statement lengths is narrower. Human code, drawn from many habits, levels of experience, and moods, is comparatively irregular. The intuition is statistical: a single model sampling at moderate

temperature produces output clustered around its learned conventions, whereas a population of humans produces a wider spread. Surface-regularity features of this kind are among those shown to carry discriminative weight in code-stylometry studies [1][3].

4.2 Comment characteristics

The presence, density, and uniformity of comments is repeatedly reported as a strong attribution cue. In at least one controlled study, the presence or absence of comment tokens moved classifier decision boundaries by several percentage points — more than abstract-syntax-tree structural metrics did [4]. Generated code frequently carries complete, uniformly-phrased, grammatically-regular comments across most constructs; human commenting tends to be sparser, more uneven, and more idiosyncratic in phrasing. Because this is a surface feature, it is also one of the easiest for an author to alter (Section 05).

4.3 Token-level predictability

Under a reference language model, generated code tends to occupy a higher-probability — that is, lower-perplexity — region than human code, because it was sampled from a distribution similar to the reference model's own. Perplexity is therefore usable as a continuous signal: informally, a measure of how "surprised" a reference model is by each successive token, averaged over the file. The zero-shot curvature methods (Section 3.2) and the perplexity methods (Section 3.3) both exploit this property, the former by examining how perplexity changes under perturbation, the latter by using the aggregate value directly [5][6].

4.4 Structural and complexity features

Features derived from the abstract syntax tree and control-flow graph — nesting-depth distribution, branching factor, and related measures of code shape — capture structure rather than surface text. Their reported discriminative power is mixed: useful in some studies, marginal in others, with at least one study finding that such structural metrics shifted classical-model decision boundaries only slightly compared with surface features [4]. The mixed result is itself informative: it indicates that, for current generating models, the detectable signal lives more in surface style than in deep structure.

4.5 Idiom and imperfection signals

Human-authored code carries traces that machine-generated code often lacks: dead code left behind during development, commented-out experiments, inconsistent shortcuts, idiosyncratic abbreviations, informal TODO and FIXME notes, and occasional misspellings in identifiers or

comments. The relative absence of such "imperfections" is weak evidence toward machine authorship. This category is the least formalised in the literature and the most easily confounded — disciplined human authors and linters remove many of these traces — but it is frequently noted qualitatively.

4.6 Combining signals

No single signal above is individually reliable; each is, in machine-learning terms, a weak classifier. The general and well-established principle is that a set of approximately-orthogonal weak signals can be combined into a stronger classifier than any one alone — the basis of ensemble methods such as random forests and gradient boosting. The detection families of Section 03 differ chiefly in how they perform this combination: engineered-feature classifiers combine explicit signals through a trained model; neural classifiers learn an implicit combination directly from token sequences. The particular choice of signals, their relative weighting, and any per-language calibration are implementation and tuning decisions specific to a given detector; they are not properties of the method class and vary between systems.

05 Provenance Detection: Documented Limitations

The literature is consistent on several limitations that apply across method families. These are reported here because they materially affect how any detection result should be interpreted.

4.1 Adversarial brittleness

Multiple studies report that classifiers remain brittle under adversarial edits: small, meaning-preserving modifications to generated code (reformatting, identifier renaming, comment insertion or removal) can move it across the decision boundary [4]. Because some of the strongest signals are surface features such as comment style (Section 4.2), an author who edits those surface features can substantially degrade detection.

4.2 Distribution shift across models and time

Detectors trained on the output of one generation of language models do not necessarily generalise to newer models. One large-scale study explicitly re-tested its detector against code from later models released after the original data-collection window, treating generalisation as an open empirical question rather than an assumption [7]. As generating models change, a fixed detector's accuracy can be expected to drift.

4.3 The "rare pure case" problem

In practice, code is frequently neither purely human-written nor purely machine-generated but a mixture — human-authored code edited with model assistance, or model-generated code subsequently revised by a human. At least one dataset effort explicitly incorporates "machine-refined" code as a third class distinct from both pure cases, and notes that purely AI-generated code is comparatively rare in real repositories [8]. A binary human-versus-machine framing is therefore a simplification of a spectrum.

4.4 False-positive cost

Because attribution of authorship to a machine can carry consequences for the human associated with the code, the asymmetric cost of false positives is a recurring theme. The literature does not establish a universal threshold; the appropriate operating point on the precision-recall curve is application-specific and is not a property the detection method can supply on its own.

06 Agent Detection: The General Framing

Agent detection addresses a different question from provenance: not who wrote the code, but what the code does when run. Specifically, it concerns whether the code constructs or drives an autonomous AI agent — software that issues model-inference calls and acts on their outputs, potentially in a loop and potentially with access to external tools or system resources.

This problem connects to a well-documented class of security concern. When code executes the output of a language model — passing model-generated strings to a dynamic-execution construct such as `eval` or `exec` — it creates a path by which manipulation of the model's input can lead to arbitrary code execution. This pattern has been catalogued by security researchers and assigned vulnerability identifiers in specific frameworks [9]. Prompt injection, the manipulation of a model through crafted input, is listed as the leading vulnerability class for language-model applications by the OWASP project [10]. Detecting where in a codebase these constructs occur is therefore a recognized static-analysis objective.

07 Agent Detection: Method Classes

6.1 Dependency and import analysis

The most reliable static signal that code instantiates an AI agent is its declared dependencies. Agent frameworks are imported by name. A static scan of import statements, dependency manifests, and lock files identifies whether a project links against known agent or model-orchestration libraries. This is a coarse signal — importing a library does not prove it is used on any given execution path — but it is high-precision for the question "could this code construct an agent at all?" The technique is the same one used in software composition analysis for license and vulnerability auditing, applied to a different target list.

6.2 Dynamic-execution construct detection

A second static signal is the presence of dynamic-execution constructs — `eval`, `exec`, `compile`, `subprocess` invocation, deserialisation of executable content — particularly where the argument to such a construct is data rather than a literal. As noted in Section 05, these constructs are the mechanism by which model output can become executed code [9]. Locating them is a classic static-analysis task: the parser identifies call sites, and data-flow analysis determines whether untrusted or model-derived data can reach them.

> *The detection of a dynamic-execution construct is not by itself a finding of an AI agent. Such constructs have many legitimate, agent-unrelated uses. The construct is a necessary condition for one class of agent behavior, not a sufficient one. Reporting it as agent evidence requires the additional context that model-derived data reaches it.*

6.3 Data-flow and call-graph analysis

A more precise but more expensive class of method builds a call graph and performs data-flow analysis to determine whether a model-inference call feeds, directly or transitively, into an action with external effect — a dynamic-execution construct, a network request, a filesystem write, or a tool invocation. This distinguishes code that merely calls a model and displays the result from code that acts autonomously on model output. The general technique is standard inter-procedural static analysis; recent work has also explored using language models themselves to assist static-analysis tasks [11], though that is a separate research direction from the detection problem itself.

6.4 The limits of static agent detection

Static analysis cannot in general determine what a program does at runtime; this is a consequence of undecidability and applies to agent detection as to any other behavioral property. Dynamically constructed import names, reflection, and configuration-driven dispatch can all hide agent construction from a purely static scan. Conversely, the mere presence of an

agent framework in the dependency tree does not establish that any agent is constructed on a reachable path. Static agent detection therefore produces evidence to be weighed, not a definitive runtime verdict. Dynamic and behavioral monitoring — observing a running system for anomalous inference calls or execution patterns — is the complementary approach where static analysis is insufficient [10].

08 Combining the Two Detections

Because provenance and agent detection answer different questions, a complete report keeps their findings distinct. A file may be flagged as probably machine-generated (a provenance finding) and separately as constructing an agent (a behavior finding); the two flags are computed by different methods and have different confidence characteristics. Conflating them — for instance, treating "imports an agent framework" as evidence that code is AI-generated — introduces a category error, since human authors routinely write agent code and AI generators routinely produce static code.

The two analyses do share infrastructure. Both depend on a parser capable of producing a token stream and, ideally, an abstract syntax tree for each supported language; both benefit from the same file-identification and language-classification front end. Beyond that shared front end, the methods diverge entirely.

09 Evaluation Methodology

The literature converges on a small set of evaluation practices that any reported detection accuracy should specify, and whose absence should be treated as a limitation of the report:

- **Dataset provenance and balance** — the source of the human and machine examples, the generating models used, and the class balance. Detectors are sensitive to all three [8]. Purpose-built datasets vary these factors deliberately: one Python dataset was generated with three different models [12], and a later multi-language successor spanning Python, Java, and Go used six models and three prompts across distinct usage scenarios, explicitly to support cross-model and cross-language evaluation [13].
- **Metric choice** — accuracy alone is misleading on imbalanced data; F1, precision, recall, and area under the ROC curve are the conventional reports [5].
- **Cross-model and cross-language generalisation** — whether the detector was tested on models and languages it was not trained on [7][3].

- **Adversarial evaluation** — whether the detector was tested against meaning-preserving edits intended to evade it [4].

Reproducibility is an acknowledged weakness in parts of this literature: surveys note heavy reliance on a small number of benchmark datasets and uneven release of training code and models [2]. Detection figures reported without the above context are difficult to compare across studies.

10 Summary

Provenance detection and agent detection are distinct problems requiring distinct methods. Provenance detection adapts authorship-attribution techniques — supervised stylometric classification, zero-shot probability-curvature analysis, perplexity scoring, and fine-tuned neural classifiers — to the human-versus-machine question, with reported accuracies ranging from the mid-80% range for interpretable multilingual stylometric models to above 97% for fine-tuned neural classifiers on specific datasets. All provenance methods share documented limitations: adversarial brittleness, distribution shift across model generations, the prevalence of mixed human-machine code, and asymmetric false-positive cost. Agent detection is a static-analysis problem centered on dependency analysis, dynamic-execution-construct identification, and data-flow tracing from model-inference calls to external-effect operations; it is bounded by the general undecidability of runtime behavior and is complemented by dynamic monitoring. A sound analysis reports the two detections separately and specifies its evaluation methodology.

References

1. Caliskan-Islam, A., Harang, R., Liu, A., Narayanan, A., Voss, C., Yamaguchi, F., Greenstadt, R. (2015). *De-anonymizing Programmers via Code Stylometry*. Proceedings of the 24th USENIX Security Symposium, 255–270.
2. Horváth, M., Pietriková, E., Spinellis, D. (2026). *Bridging Behavioral Biometrics and Source Code Stylometry: A Survey of Programmer Attribution*. ACM Computing Surveys (under review); arXiv:2603.11150. Systematic mapping study of 47 programmer-attribution studies (2012–2025), surveying feature categories, learning approaches, datasets, and evaluation practice.
3. Bukhari, S., et al. (2024). *Is This You, LLM? Recognizing AI-written Programs with Multilingual Code Stylometry*. arXiv:2412.14611. Reports a single classifier with 84.1% average F1 across ten languages and

releases the H-AIRosettaMP dataset.

4. Bisztray, T., Cherif, B., Dubniczky, R. A., Gruschka, N., Borsos, B., Ferrag, M. A., Kovács, A., Mavroeidis, V., Tihanyi, N. (2025). *I Know Which LLM Wrote Your Code Last Summer: LLM-generated Code Stylometry for Authorship Attribution.* Proceedings of the 18th ACM Workshop on Artificial Intelligence and Security (AISec), pp. 28–39. arXiv:2506.17323. Reports comment-token presence as a strong attribution cue and structural metrics as weak, and documents adversarial brittleness.
5. Mitchell, E., Lee, Y., Khazatsky, A., Manning, C. D., Finn, C. (2023). *DetectGPT: Zero-Shot Machine-Generated Text Detection using Probability Curvature.* Proceedings of the 40th International Conference on Machine Learning (ICML).
6. Xu, Z. and Sheng, V. S. (2024). *Detecting AI-Generated Code Assignments Using Perplexity of Large Language Models.* Proceedings of the AAAI Conference on Artificial Intelligence.
7. *Who is Using AI to Code? Global Diffusion and Impact of Generative AI.* (2025). arXiv:2506.08945. Re-tests an AI-code detector against code from language models released after its training window.
8. Orel, D., Paul, I., Gurevych, I., Nakov, P. (2025). *Droid: A Resource Suite for AI-Generated Code Detection.* Proceedings of EMNLP 2025, pp. 3126–onwards; arXiv:2507.10583. Introduces the DroidCollection dataset, including human-written, machine-generated, and machine-refined code classes across multiple languages.
9. NVIDIA AI Red Team (2025). *Practical LLM Security Advice from the NVIDIA AI Red Team.* NVIDIA Technical Blog. Documents remote code execution arising from execution of model output via `eval` / `exec` without isolation.
10. OWASP Foundation. *OWASP Top 10 for Large Language Model Applications.* Lists prompt injection (LLM01) as the leading vulnerability class for language-model applications.
11. Hao, Y., et al. (2023). *E&V: Prompting Large Language Models to Perform Static Analysis by Pseudo-code Execution and Verification.* arXiv:2312.08477. Cited as an example of language models assisting static-analysis tasks, a separate direction from detection.
12. Demirok, B. and Kutlu, M. (2024). *AIGCodeSet: A New Annotated Dataset for AI Generated Code Detection.* arXiv:2412.16594. A dataset of 2,828 AI-generated and 4,755 human-written Python programs created with CodeLlama 34B, Codestral 22B, and Gemini 1.5 Flash; reports a Bayesian classifier outperforming other baseline detectors tested.
13. Demirok, B., Kutlu, M., Mergen, S. (2025). *MultiAIGCD: A Comprehensive Dataset for AI Generated Code Detection Covering Multiple Languages, Models, Prompts, and Scenarios.* arXiv:2507.21693. 121,271 AI-generated and 32,148 human-written snippets across Python, Java, and Go, generated by six models under three prompts and three usage scenarios, benchmarked under cross-model and cross-language settings.

Appendix: Links to Academic Papers

Direct links to the cited sources, in reference order. Where a paper sits behind a publisher paywall, the link resolves to the canonical record (DOI or proceedings page) from which open versions can usually be located.

1. Caliskan-Islam et al. (2015) — usenix.org/conference/usenixsecurity15/.../caliskan-islam
2. Horváth, Pietriková, Spinellis (2026), *Bridging Behavioral Biometrics and Source Code Stylometry* — arxiv.org/abs/2603.11150
3. Gurioli, Gabbrielli, Zacchiroli (2024), *Is This You, LLM?* — arxiv.org/abs/2412.14611
4. Bisztray et al. (2025), *I Know Which LLM Wrote Your Code Last Summer* (AISec 2025) — arxiv.org/abs/2506.17323 (DOI: 10.1145/3733799.3762964)
5. Mitchell et al. (2023), *DetectGPT* — arxiv.org/abs/2301.11305
6. Xu & Sheng (2024), *Detecting AI-Generated Code Assignments Using Perplexity* — ojs.aaai.org/index.php/AAAI/article/view/30361
7. *Who is Using AI to Code?* (2025) — arxiv.org/abs/2506.08945
8. Orel, Paul, Gurevych, Nakov (2025), *Droid* — aclanthology.org/2025.emnlp-main.1593/ (arXiv: 2507.10583)
9. NVIDIA AI Red Team (2025) — developer.nvidia.com/blog/practical-llm-security-advice...
10. OWASP, *LLM01: Prompt Injection* — genai.owasp.org/llmrisk/llm01-prompt-injection/
11. Hao et al. (2023), *E&V* — arxiv.org/abs/2312.08477
12. Demirok & Kutlu (2024), *AIGCodeSet* — arxiv.org/abs/2412.16594
13. Demirok, Kutlu, Mergen (2025), *MultiAIGCD* — arxiv.org/abs/2507.21693