How to Detect AI Coding Agents in Open Source Repositories

Maria Cattini | 06/07/2026 | AI

The first mistake is looking for the bot account.

It is visible, convenient and easy to count.

It is also incomplete.

AI coding agents do not leave one clean trace in open-source repositories. Sometimes they appear as pull request authors. Sometimes they are visible in commit messages. Sometimes they leave configuration files. Sometimes they are only suggested by co-authorship patterns, tool-specific files, generated instructions, task logs or repeated changes that do not look like ordinary human maintenance.

For OSINT work, that changes the question.

Do not ask only:

```
Was this commit made by an AI agent?
```

Ask:

```
Which public signals support agent involvement, and how strong is each signal?
```

That distinction matters because AI coding agents are now part of the open-source supply chain. They write code, open pull requests, modify dependencies, generate tests, touch configuration files and sometimes operate inside developer environments with access to credentials, local files and project context. Treating them as invisible productivity tools is no longer enough.

They are becoming observable actors.

But they are not always observable in the same way.

Why one signal is not enough

A June 2026 paper by Arsham Khosravani and Audris Mockus, "Detecting AI Coding Agents in Open Source", gives a useful warning for investigators: no single detection method captures the full picture.

The authors studied more than 180 million Git repositories through World of Code and combined several signals: configuration-file scanning, commit-message analysis, author-identity matching and

bot-signature lookup. Their result is not just a measurement of adoption. It is a methodological lesson.

Bot-account lookup alone misses too much.

In one snapshot, their multi-method approach identified 850,157 Claude Code commits. A bot-account lookup recovered only 28,154 of them, or 3.3 percent. Across snapshots from December 2024 to April 2026, the paper reports more than 320,000 commit-attributed agent commits per month.

The exact numbers will change over time.

The lesson is more durable:

```
If you only count the obvious bots, you are not measuring agent activity.
You are measuring the part of agent activity that chose to identify itself.
```

For open-source OSINT, this means the workflow must be multi-signal from the start.

The four public traces to check

An AI coding agent can surface through several different layers. None of them is perfect. Each should be treated as one piece of an evidence chain.

1. Bot or agent identity

This is the most direct signal.

Look for:

- known bot accounts;
- agent-branded users;
- pull requests opened by automation;
- commits authored by accounts that identify as a coding assistant;
- signed-off messages that mention a tool or model;
- platform labels that mark a contribution as automated.

This signal is strong when present, but weak as a coverage method.

Many agent-assisted changes are committed by the human developer. Some tools run locally. Some agents prepare changes that a human reviews and commits manually. Some projects squash commits and erase the original authorship pattern. Some teams deliberately avoid exposing agent identity in public history.

The absence of a bot account is not evidence of absence.

2. Commit-message and co-author patterns

Commit messages can reveal agent involvement even when the author is human.

Look for:

- "Generated with" patterns;
- "Co-authored-by" lines linked to known agents;
- repeated phrasing across unrelated repositories;
- overly structured messages that match tool templates;

• commit bodies that include task summaries, tests run or tool output;
• references to local agent sessions, prompts or commands.

This layer is useful because it often survives in Git history.

It is also easy to overread.

Human developers can copy templates. Teams can standardize messages. Some tools produce human-like commit text. Some maintainers edit agent output before committing. A commit message should support a hypothesis, not conclude it.

3. Configuration files and tool artifacts

AI coding agents often need local or repository-level configuration.

Depending on the tool and workflow, public repositories may contain:

• agent configuration files;
• instruction files;
• task definitions;
• prompt templates;
• local workflow examples;
• CI instructions for agent-generated pull requests;
• documentation telling contributors how to use a coding assistant;
• hidden or tool-specific folders that support agent execution.

This is one of the most important OSINT layers because it can show adoption even when no commit is explicitly attributed to an agent.

The Khosravani and Mockus paper highlights this problem: some adoption is visible only through configuration-file signals. That matters because local, in-editor agents can produce code that later appears under an ordinary human identity.

Configuration evidence is not proof that a specific commit was agent-authored.

It is evidence that the project had an agent-capable workflow.

4. Pull request behavior

Pull requests can reveal another kind of agent activity:

• automated branch names;
• repetitive PR descriptions;
• generated test plans;
• unusual diff-to-description consistency;
• high commit counts for small changes;
• routine maintenance work produced at unusual scale;
• comments that expose review loops between humans and agents.

The problem is that PR data and commit data do not always describe the same population.

The 2026 census paper compares its commit-detected agents with an independent pull-request census and finds that the two channels capture different kinds of agent use. PR-deployed cloud agents surface differently from local in-editor agents. If an investigator studies only pull requests, local agent use can disappear. If an investigator studies only commits, agent-generated review workflows can be missed.

For OSINT, that means the repository should be treated as a system, not a single page.

A practical OSINT workflow

Use this workflow when you need to assess whether an open-source project shows public signs of AI coding agent use.

Step 1: Define the unit of analysis

Do not begin with the whole ecosystem.

Choose one of these:

- a repository;
- a project organization;
- a time window;
- a specific release;
- a pull request set;
- a package version;
- a suspected incident timeline.

Then define the question:

```
Are we trying to detect any agent adoption?
Are we trying to attribute a specific change?
Are we trying to assess supply-chain risk?
Are we trying to understand contributor behavior?
```

The evidence standard changes with the question.

Detecting adoption requires weaker evidence than attributing a specific security-relevant commit.

Step 2: Build a timeline

Start with public Git data:

- first appearance of agent-related configuration;
- first agent-labeled PR;
- first commit-message signal;
- first bot account contribution;
- release dates before and after adoption;
- security-relevant changes near those dates.

The goal is not to prove everything at once.

The goal is to find when agent traces begin.

This matters because a configuration file added after a suspicious commit cannot explain that earlier commit. A bot account that appears only in pull requests may not explain local commits. A tool instruction file may show adoption, but not authorship of a particular patch.

Timeline first.

Interpretation second.

Step 3: Separate adoption from authorship

This is the most common analytical error.

A repository can adopt an AI coding agent without every later commit being agent-authored. A commit can be agent-assisted without being fully generated by an agent. A pull request can be opened by an agent and then heavily rewritten by a human. A human can use an agent privately and leave no public attribution.

Use three labels:

```
Agent-
capable workflow: public files or documentation show the project can use an agent.
Agent-assisted contribution: public traces suggest an agent participated.
Agent-attributed contribution: public metadata directly identifies an agent or tool.
```

Do not collapse them into one category.

Step 4: Check the repository for agent artifacts

Look for files and documentation that change the development workflow.

Examples:

- contributor instructions for AI tools;
- task or prompt files;
- agent-specific configuration;
- generated test plans;
- tool-specific session logs accidentally committed;
- references to local coding agents in documentation;
- CI or automation rules for agent-created branches or PRs.

Do not publish sensitive material if you find it.

If the repository accidentally exposes prompts, logs, secrets or private context, document the existence, hash, path and retrieval time for your own evidence log, but avoid amplifying private data. When appropriate, report the exposure responsibly.

The OSINT value is in the signal, not in exposing more than necessary.

Step 5: Compare commit metadata with PR metadata

For each suspicious or relevant change, collect:

- commit hash;
- author and committer;
- timestamp;
- branch;
- PR number;
- labels;
- review comments;
- commit message;
- files changed;
- tests mentioned;
- release notes;
- any linked issue.

Then ask:

```
Does the commit signal match the PR signal?
Does the PR signal match the repository configuration?
Does the timeline make sense?
Is there an independent sign of agent use?
```

One signal can mislead.

Convergence is stronger.

Step 6: Review the type of work

Agent traces are not only about identity. They are also about task type.

Several recent studies suggest that different agent workflows surface differently. PR-deployed agents often appear in visible feature work. Local in-editor agents may surface more in maintenance or routine changes. SWE-chat, a 2026 dataset of real coding-agent sessions, also shows that agent use in the wild is messy: developers push back, interrupt, correct outputs and often commit only part of what the agent produced.

That matters for analysis.

Do not assume that an agent trace means the entire change was machine-generated. Do not assume that clean code means human authorship. Do not assume that routine maintenance is low-risk. Dependency updates, configuration changes, test rewrites and small refactors can all affect security.

Classify the work:

- feature;
- bug fix;
- dependency update;
- test generation;
- refactor;
- documentation;
- CI or build change;
- security patch;
- configuration change.

Then evaluate risk by what changed, not only by who or what may have helped write it.

Evidence table

Use a table like this before writing a conclusion.

Signal	What it can support	What it cannot prove alone	Confidence
Bot account opened PR	Direct automation or agent workflow	That every line was agent-written	Medium to high
Commit message mentions tool	Agent assistance or attribution	That the tool produced the final code	Medium
Agent config file exists	Project has or had agent-capable workflow	That a specific commit was agent-authored	Low to medium

Signal	What it can support	What it cannot prove alone	Confidence
Tool-specific instruction file	Local agent setup or contributor workflow	Active use without timeline support	Low to medium
Repeated generated PR descriptions	Automated workflow pattern	Maliciousness or unsafe code	Medium
Human author plus agent co-author	Human-agent collaboration	Percentage of agent-written code	Medium
Sudden agent traces before release	Possible adoption near release	Causation for later bugs or vulnerabilities	Low without more evidence

This table prevents the most damaging shortcut: turning a weak trace into a strong claim.

Why this matters for supply-chain security

Agent adoption is not automatically a security problem.

But it changes the attack surface.

A coding agent can read documentation, run setup scripts, edit files, call package managers, follow repository instructions, generate code, write tests and sometimes execute commands. If it acts inside a developer environment, it may be close to credentials, local secrets, private repositories, browser sessions or deployment tools.

That is why public traces of agent adoption matter for supply-chain analysis.

They help answer questions such as:

- Did the project recently introduce an agent-assisted workflow?
- Are security-sensitive changes being generated or reviewed differently?
- Are agent instructions stored in the repository?
- Do generated pull requests receive human review?
- Are dependency or CI changes being made by automation?
- Are releases close to unexplained changes in contributor patterns?

This does not mean every AI-assisted project is unsafe.

It means agent activity should become part of the repository risk model.

Common false positives

Be careful with these:

A generic commit message is not proof

Phrases like "update tests" or "fix linting" can be human or machine-written. Template-like text is a weak signal unless it matches a known tool pattern or appears with other evidence.

A config file is not authorship

An agent configuration file shows capability or adoption. It does not prove that a specific line of code was generated by an agent.

A bot account is not the whole story

Visible bots are only the easiest part to count. Local agents, human-committed agent output and edited agent suggestions may not use bot accounts.

A security issue is not automatically caused by AI

If a vulnerability appears after agent adoption, that is a timeline signal. It is not causation. You still need code review, diff analysis, issue history and preferably independent confirmation.

A clean repository is not a safe repository

Recent agent-security research and incident reporting show that dangerous behavior can be triggered indirectly through documentation, setup commands, dependency scripts or runtime instructions. The repository may look ordinary at first glance.

For OSINT, that means static inspection should be combined with workflow analysis.

A defensive checklist

Before publishing an assessment, answer these questions:

1. What exact repository, organization, time window or release is being analyzed?
2. Which agent traces were observed?
3. Are the traces identity-based, message-based, configuration-based or PR-based?
4. Do at least two independent signals converge?
5. Does the timeline support the conclusion?
6. Is the claim about adoption, assistance or authorship?
7. Are security-relevant files involved?
8. Are dependency, CI, credential, permission or release workflows affected?
9. What could be a false positive?
10. What remains unknown?

The final line should be conservative.

Not:

```
This project is AI-generated.
```

Better:

```
This repository shows public signs of AI-agent adoption through configuration files a
nd commit-message patterns. The available evidence supports agent assistance in the w
orkflow, but not full attribution of the reviewed change.
```

That is less dramatic.

It is also more accurate.

The new OSINT problem

AI coding agents are not only tools inside private developer workflows. They are beginning to leave public traces across open-source infrastructure.

Those traces are uneven.

Some are obvious. Some are indirect. Some are deliberately hidden behind human commits. Some are visible only when repository files, pull requests, commit messages and timelines are analyzed

together.

That is the new OSINT problem.

The investigator's task is not to guess whether a machine wrote the code.

The task is to build an evidence chain around agent involvement:

```
identity + metadata + configuration + timeline + code change + review trail + limits
```

One signal is a clue.

A converging set of signals is a finding.

And in the open-source supply chain, that distinction is now part of the security work.
The first mistake is looking for the bot account.

It is visible, convenient and easy to count.

It is also incomplete.

AI coding agents do not leave one clean trace in open-source repositories. Sometimes they appear as pull request authors. Sometimes they are visible in commit messages. Sometimes they leave configuration files. Sometimes they are only suggested by co-authorship patterns, tool-specific files, generated instructions, task logs or repeated changes that do not look like ordinary human maintenance.

For OSINT work, that changes the question.

Do not ask only:

```
Was this commit made by an AI agent?
```

Ask:

```
Which public signals support agent involvement, and how strong is each signal?
```

That distinction matters because AI coding agents are now part of the open-source supply chain. They write code, open pull requests, modify dependencies, generate tests, touch configuration files and sometimes operate inside developer environments with access to credentials, local files and project context. Treating them as invisible productivity tools is no longer enough.

They are becoming observable actors.

But they are not always observable in the same way.

Why one signal is not enough

A June 2026 paper by Arsham Khosravani and Audris Mockus, "Detecting AI Coding Agents in Open Source", gives a useful warning for investigators: no single detection method captures the full picture.

The authors studied more than 180 million Git repositories through World of Code and combined

several signals: configuration-file scanning, commit-message analysis, author-identity matching and bot-signature lookup. Their result is not just a measurement of adoption. It is a methodological lesson.

Bot-account lookup alone misses too much.

In one snapshot, their multi-method approach identified 850,157 Claude Code commits. A bot-account lookup recovered only 28,154 of them, or 3.3 percent. Across snapshots from December 2024 to April 2026, the paper reports more than 320,000 commit-attributed agent commits per month.

The exact numbers will change over time.

The lesson is more durable:

```
If you only count the obvious bots, you are not measuring agent activity.
You are measuring the part of agent activity that chose to identify itself.
```

For open-source OSINT, this means the workflow must be multi-signal from the start.

The four public traces to check

An AI coding agent can surface through several different layers. None of them is perfect. Each should be treated as one piece of an evidence chain.

1. Bot or agent identity

This is the most direct signal.

Look for:

• known bot accounts;
• agent-branded users;
• pull requests opened by automation;
• commits authored by accounts that identify as a coding assistant;
• signed-off messages that mention a tool or model;
• platform labels that mark a contribution as automated.

This signal is strong when present, but weak as a coverage method.

Many agent-assisted changes are committed by the human developer. Some tools run locally. Some agents prepare changes that a human reviews and commits manually. Some projects squash commits and erase the original authorship pattern. Some teams deliberately avoid exposing agent identity in public history.

The absence of a bot account is not evidence of absence.

2. Commit-message and co-author patterns

Commit messages can reveal agent involvement even when the author is human.

Look for:

• "Generated with" patterns;
• "Co-authored-by" lines linked to known agents;
• repeated phrasing across unrelated repositories;

• overly structured messages that match tool templates;
• commit bodies that include task summaries, tests run or tool output;
• references to local agent sessions, prompts or commands.

This layer is useful because it often survives in Git history.

It is also easy to overread.

Human developers can copy templates. Teams can standardize messages. Some tools produce human-like commit text. Some maintainers edit agent output before committing. A commit message should support a hypothesis, not conclude it.

3. Configuration files and tool artifacts

AI coding agents often need local or repository-level configuration.

Depending on the tool and workflow, public repositories may contain:

• agent configuration files;
• instruction files;
• task definitions;
• prompt templates;
• local workflow examples;
• CI instructions for agent-generated pull requests;
• documentation telling contributors how to use a coding assistant;
• hidden or tool-specific folders that support agent execution.

This is one of the most important OSINT layers because it can show adoption even when no commit is explicitly attributed to an agent.

The Khosravani and Mockus paper highlights this problem: some adoption is visible only through configuration-file signals. That matters because local, in-editor agents can produce code that later appears under an ordinary human identity.

Configuration evidence is not proof that a specific commit was agent-authored.

It is evidence that the project had an agent-capable workflow.

4. Pull request behavior

Pull requests can reveal another kind of agent activity:

• automated branch names;
• repetitive PR descriptions;
• generated test plans;
• unusual diff-to-description consistency;
• high commit counts for small changes;
• routine maintenance work produced at unusual scale;
• comments that expose review loops between humans and agents.

The problem is that PR data and commit data do not always describe the same population.

The 2026 census paper compares its commit-detected agents with an independent pull-request census and finds that the two channels capture different kinds of agent use. PR-deployed cloud agents surface differently from local in-editor agents. If an investigator studies only pull requests, local agent use can disappear. If an investigator studies only commits, agent-generated review

workflows can be missed.

For OSINT, that means the repository should be treated as a system, not a single page.

A practical OSINT workflow

Use this workflow when you need to assess whether an open-source project shows public signs of AI coding agent use.

Step 1: Define the unit of analysis

Do not begin with the whole ecosystem.

Choose one of these:

- a repository;
- a project organization;
- a time window;
- a specific release;
- a pull request set;
- a package version;
- a suspected incident timeline.

Then define the question:

```
Are we trying to detect any agent adoption?
Are we trying to attribute a specific change?
Are we trying to assess supply-chain risk?
Are we trying to understand contributor behavior?
```

The evidence standard changes with the question.

Detecting adoption requires weaker evidence than attributing a specific security-relevant commit.

Step 2: Build a timeline

Start with public Git data:

- first appearance of agent-related configuration;
- first agent-labeled PR;
- first commit-message signal;
- first bot account contribution;
- release dates before and after adoption;
- security-relevant changes near those dates.

The goal is not to prove everything at once.

The goal is to find when agent traces begin.

This matters because a configuration file added after a suspicious commit cannot explain that earlier commit. A bot account that appears only in pull requests may not explain local commits. A tool instruction file may show adoption, but not authorship of a particular patch.

Timeline first.

Interpretation second.

Step 3: Separate adoption from authorship

This is the most common analytical error.

A repository can adopt an AI coding agent without every later commit being agent-authored. A commit can be agent-assisted without being fully generated by an agent. A pull request can be opened by an agent and then heavily rewritten by a human. A human can use an agent privately and leave no public attribution.

Use three labels:

```
Agent-
capable workflow: public files or documentation show the project can use an agent.
Agent-assisted contribution: public traces suggest an agent participated.
Agent-attributed contribution: public metadata directly identifies an agent or tool.
```

Do not collapse them into one category.

Step 4: Check the repository for agent artifacts

Look for files and documentation that change the development workflow.

Examples:

• contributor instructions for AI tools;
• task or prompt files;
• agent-specific configuration;
• generated test plans;
• tool-specific session logs accidentally committed;
• references to local coding agents in documentation;
• CI or automation rules for agent-created branches or PRs.

Do not publish sensitive material if you find it.

If the repository accidentally exposes prompts, logs, secrets or private context, document the existence, hash, path and retrieval time for your own evidence log, but avoid amplifying private data. When appropriate, report the exposure responsibly.

The OSINT value is in the signal, not in exposing more than necessary.

Step 5: Compare commit metadata with PR metadata

For each suspicious or relevant change, collect:

• commit hash;
• author and committer;
• timestamp;
• branch;
• PR number;
• labels;
• review comments;
• commit message;
• files changed;
• tests mentioned;

• release notes;
• any linked issue.

Then ask:

```
Does the commit signal match the PR signal?
Does the PR signal match the repository configuration?
Does the timeline make sense?
Is there an independent sign of agent use?
```

One signal can mislead.

Convergence is stronger.

Step 6: Review the type of work

Agent traces are not only about identity. They are also about task type.

Several recent studies suggest that different agent workflows surface differently. PR-deployed agents often appear in visible feature work. Local in-editor agents may surface more in maintenance or routine changes. SWE-chat, a 2026 dataset of real coding-agent sessions, also shows that agent use in the wild is messy: developers push back, interrupt, correct outputs and often commit only part of what the agent produced.

That matters for analysis.

Do not assume that an agent trace means the entire change was machine-generated. Do not assume that clean code means human authorship. Do not assume that routine maintenance is low-risk. Dependency updates, configuration changes, test rewrites and small refactors can all affect security.

Classify the work:

• feature;
• bug fix;
• dependency update;
• test generation;
• refactor;
• documentation;
• CI or build change;
• security patch;
• configuration change.

Then evaluate risk by what changed, not only by who or what may have helped write it.

Evidence table

Use a table like this before writing a conclusion.

Signal	What it can support	What it cannot prove alone	Confidence
Bot account opened PR	Direct automation or agent workflow	That every line was agent-written	Medium to high
Commit message mentions tool	Agent assistance or attribution	That the tool produced the final code	Medium

Signal	What it can support	What it cannot prove alone	Confidence
Agent config file exists	Project has or had agent-capable workflow	That a specific commit was agent-authored	Low to medium
Tool-specific instruction file	Local agent setup or contributor workflow	Active use without timeline support	Low to medium
Repeated generated PR descriptions	Automated workflow pattern	Maliciousness or unsafe code	Medium
Human author plus agent co-author	Human-agent collaboration	Percentage of agent-written code	Medium
Sudden agent traces before release	Possible adoption near release	Causation for later bugs or vulnerabilities	Low without more evidence

This table prevents the most damaging shortcut: turning a weak trace into a strong claim.

Why this matters for supply-chain security

Agent adoption is not automatically a security problem.

But it changes the attack surface.

A coding agent can read documentation, run setup scripts, edit files, call package managers, follow repository instructions, generate code, write tests and sometimes execute commands. If it acts inside a developer environment, it may be close to credentials, local secrets, private repositories, browser sessions or deployment tools.

That is why public traces of agent adoption matter for supply-chain analysis.

They help answer questions such as:

- Did the project recently introduce an agent-assisted workflow?
- Are security-sensitive changes being generated or reviewed differently?
- Are agent instructions stored in the repository?
- Do generated pull requests receive human review?
- Are dependency or CI changes being made by automation?
- Are releases close to unexplained changes in contributor patterns?

This does not mean every AI-assisted project is unsafe.

It means agent activity should become part of the repository risk model.

Common false positives

Be careful with these:

A generic commit message is not proof

Phrases like "update tests" or "fix linting" can be human or machine-written. Template-like text is a weak signal unless it matches a known tool pattern or appears with other evidence.

A config file is not authorship

An agent configuration file shows capability or adoption. It does not prove that a specific line of code was generated by an agent.

A bot account is not the whole story

Visible bots are only the easiest part to count. Local agents, human-committed agent output and edited agent suggestions may not use bot accounts.

A security issue is not automatically caused by AI

If a vulnerability appears after agent adoption, that is a timeline signal. It is not causation. You still need code review, diff analysis, issue history and preferably independent confirmation.

A clean repository is not a safe repository

Recent agent-security research and incident reporting show that dangerous behavior can be triggered indirectly through documentation, setup commands, dependency scripts or runtime instructions. The repository may look ordinary at first glance.

For OSINT, that means static inspection should be combined with workflow analysis.

A defensive checklist

Before publishing an assessment, answer these questions:

1. What exact repository, organization, time window or release is being analyzed?
2. Which agent traces were observed?
3. Are the traces identity-based, message-based, configuration-based or PR-based?
4. Do at least two independent signals converge?
5. Does the timeline support the conclusion?
6. Is the claim about adoption, assistance or authorship?
7. Are security-relevant files involved?
8. Are dependency, CI, credential, permission or release workflows affected?
9. What could be a false positive?
10. What remains unknown?

The final line should be conservative.

Not:

```
This project is AI-generated.
```

Better:

```
This repository shows public signs of AI-agent adoption through configuration files a
nd commit-message patterns. The available evidence supports agent assistance in the w
orkflow, but not full attribution of the reviewed change.
```

That is less dramatic.

It is also more accurate.

The new OSINT problem

AI coding agents are not only tools inside private developer workflows. They are beginning to leave public traces across open-source infrastructure.

Those traces are uneven.

Some are obvious. Some are indirect. Some are deliberately hidden behind human commits. Some are visible only when repository files, pull requests, commit messages and timelines are analyzed together.

That is the new OSINT problem.

The investigator's task is not to guess whether a machine wrote the code.

The task is to build an evidence chain around agent involvement:

```
identity + metadata + configuration + timeline + code change + review trail + limits
```

One signal is a clue.

A converging set of signals is a finding.

And in the open-source supply chain, that distinction is now part of the security work.