



AST-grep for Faster, Safer Code Changes by AI Agents and Human Engineers

Executive summary

AST-grep is a structural search, linting, and rewriting tool that matches code by syntax-tree structure rather than raw text, aiming to make large-scale, mechanical code changes both precise and fast. It is designed as a “syntax-aware grep/sed”, supports ad-hoc one-liners (`ast-grep run`) as well as project-wide scanning (`ast-grep scan`), and provides testing utilities (`ast-grep test`) plus editor integration via an LSP and a VS Code extension.

For AI coding agents, the main performance gain from AST-grep is not that it “writes code”, but that it makes **finding**, **constraining**, **verifying**, and **mechanically rewriting** code cheaper and more reliable—reducing iteration loops, lowering token usage for retrieval, and improving correctness of repetitive edits. AST-grep’s official guidance explicitly discusses “using ast-grep with AI tools” via prompting, feeding full documentation (`llms.txt`), and tool-augmented setups (notably an experimental MCP server) that enable iterative rule development and AST inspection.

For human engineers, AST-grep slots naturally into linting, refactoring (codemods), and review automation: rules-as-config in a repo, CI enforcement with SARIF emission, interactive “confirm/apply” rewrites, and editor diagnostics/code actions driven by the same rule set.

Performance evaluation is best treated as a measurement problem across multiple dimensions—accuracy/correctness, false positive/negative rates, runtime/latency, and developer productivity/throughput. AST-grep’s own rule-testing framing maps directly to detection theory (reported/noisy/missing/validated), and industry-standard productivity frameworks (SPACE) and delivery metrics (DORA) provide complementary lenses for quantifying impact in real teams.

Unspecified in the request (and therefore not assumed): target organisation size, codebase size, languages beyond those requested, baseline toolchain (linters/formatters/SAST), CI/CD platform, and any hard constraints on latency or spend.

What AST-grep is and what it can do

AST-grep (CLI name often shown as `ast-grep` and also referenced as `sg`) is a fast, polyglot tool for structural search, linting, and rewriting “at large scale”, implemented in Rust and designed to utilise CPU cores for parallel scanning.

Its core operating model is:

1. Parse source files into syntax trees using Tree-sitter parsers.

2. Match candidate nodes using AST-grep's pattern/rule system.
3. Optionally rewrite matches (from simple template replacements to more advanced “rewriter” workflows) and patch the original code.

Tree-sitter itself is an incremental parsing library that builds concrete syntax trees and can update them efficiently as code changes—one reason it underpins editor tooling and “structure-aware” search.

AST-grep exposes this model through several interfaces:

- **Ad-hoc command-line querying and rewriting:** `ast-grep run --pattern <PATTERN> ...` with optional `--rewrite` and interactive editing.
- **Project scanning as a linter/refactoring engine:** `ast-grep scan` consumes `sgconfig.yml` and YAML rule files, supports multi-threading, interactive sessions, and “apply all” rewriting.
- **Rule development and regression testing:** `ast-grep test` supports “valid/invalid” cases and reports outcomes aligned with false positives/false negatives concepts.
- **Editor integration:** an LSP plus an official VS Code extension that provides structural search/replace UI, diagnostics, and code actions when an `sgconfig.yml` project is configured.
- **Programmatic APIs/bindings:** a Node.js binding (`@ast-grep/napi`) is highlighted in official docs, plus API usage guidance focused on reducing FFI overhead and maximising parallelism.

Pattern syntax and rule expressiveness

AST-grep patterns are valid code snippets in the target language, parsed into ASTs and matched structurally (including nested matches). This forces a key discipline: patterns must be syntactically parseable.

It supports: - **Meta-variables** (`$NAME`) as AST-node wildcards. - **Multi-node meta-variables** (`$$$`) to match “zero or more” nodes (arguments, parameters, statements), optionally named (e.g. `$$$ARGS`). - Non-capturing meta-variables (leading underscore) as a stated micro-optimisation technique.

Rules are YAML objects with a clear separation between **finding** (`rule`, `constraints`, etc.) and **patching** (`fix`, `transform`, `rewriters`), and AST-grep documents both the schema and how constraints are applied (match first, then filter).

How AST-grep works at a systems level

The “bird’s-eye view” documentation describes AST-grep as accepting queries in multiple formats (pattern, YAML rule, and API calls), parsing with Tree-sitter, then performing matching and optional rewrite/lint/analyse operations, with attention to multi-core performance.

```

flowchart TD
    A[Source files] --> B[Tree-sitter parse]
    B --> C[Syntax trees]
    D[Query input:\npattern / YAML rule / API] --> E[Matcher]
    C --> E
    E --> F{Mode}
    F -->|Search| G[Locations + captures]
  
```

```
F -->|Lint| H[Diagnostics + severity]
F -->|Rewrite| I[Generate fix from captures]
I --> J[Patch text back into files]
```

This operational decomposition is useful when assessing “performance improvements”: AST-grep can speed up *search* and *mechanical rewrite* steps, but it does not, by itself, establish semantic correctness (types/dataflow/control-flow) unless you optionally pair it with other analyses.

How AST-grep differs from regex and other AST-based tools

Structural matching vs regex matching

Regex and plain-text tools match byte sequences; structural tools match syntax constructs. The practical difference shows up immediately in AST-grep’s examples: a pattern like `console.log($GREETING)` matches code but not the same tokens inside comments or strings, because those are different syntactic nodes.

This property is directly relevant to both humans and AI agents: structural matching reduces accidental edits and spurious matches (a core driver of false positives) compared to regex-heavy workflows.

Comparison table

The table below focuses on *capability shape* rather than popularity. Where features are optional/paid/experimental, that is noted at a high level.

Tool / approach	Primary strength	Typical weak spot	Best fit vs AST-grep
Regex / grep-family	Fast text scanning; ubiquitous	No syntax awareness; fragile for refactors	Use when you truly want text-only matches (comments, docs, logs) rather than code structure.
Tree-sitter queries	Precise node-type S-expression matching and captures	Query language is lower-level; not a rewrite/lint “product” by default	Use when you already have Tree-sitter ASTs and want bespoke extraction/highlighting rules.
	Pattern-based static analysis; includes dataflow/taint modes and other advanced analyses	Heavier analysis can cost more time; designed strongly around security workflows	Use when you need semantic-ish analyses (taint/dataflow) or security rule ecosystems; AST-grep is lighter-weight and primarily syntactic.

Tool / approach	Primary strength	Typical weak spot	Best fit vs AST-grep
Comby	Lightweight “template” structural search across many languages/data formats; can mix regex in holes	Not language-parse-based in the same way; limitations around indentation-sensitive languages are noted in AST-grep’s comparison	Use when you must operate across non-code formats or arbitrary syntaxes; prefer AST-grep when you want real parser-backed matches.
Coccinelle (SmPL)	Semantic patching for C; strong for collateral evolution and large refactors in C	Language scope is C-centric; learning SmPL	Use for large-scale C evolutions where SmPL’s patch paradigm fits; AST-grep targets many languages via Tree-sitter.
OpenRewrite	Automated refactoring “recipes”, especially for ecosystem migrations; deep build-tool integration	Heavier and ecosystem-specific (often JVM-centric), recipe authoring overhead	Use for framework migrations with published recipes and build integration; AST-grep for fast syntactic enforcement and multi-language coding patterns.
jscodeshift	JavaScript/TypeScript codemod toolkit; AST-to-AST transform runner over files	Requires writing imperative transforms; narrower language focus	Use when you need custom scripted JS/TS AST transforms; AST-grep is often faster to author for pattern-like changes.

Syntactic tools vs semantic tools

AST-grep’s own comparison explicitly positions it as syntactic: it does **not** provide deep semantic equivalence, type information, control-flow, or dataflow/taint analysis (at least “at the moment” of that documentation).

Research on syntactic code search frames tools like Semgrep and Comby as “lightweight syntactic analysis” that leverage tree structure for expressiveness beyond regex, while still differing from frameworks that require explicit AST manipulations or deeper program analysis.

This distinction matters for performance claims: - AST-grep can **reduce the cost and risk of syntactic transformations** (mechanical refactors, style constraints, API renames), especially where type information is not required. - It cannot, alone, prove semantic correctness; you typically pair it with compilation/tests, and optionally other analysers (type checkers, SAST) when semantics matter.

Integration patterns for AI code-generation agents

AST-grep’s official “Using ast-grep with AI tools” guidance describes three broad ways to make LLMs use it reliably: (1) prompting conventions, (2) providing full documentation context, and (3) tool-augmented

“agentic” rule development using an MCP server. These map cleanly onto prompt-time, post-processing, tool augmentation, and feedback loops.

Prompt-time integration

Prompt-time integration means you instruct the agent that structural questions should be answered via AST-grep rather than text search. The AST-grep docs even include a concrete example of a system-level prompt (in an `AGENTS.md`-style workflow) that prioritises `ast-grep --lang ... -p ...` for syntax-aware searches.

Key prompt-time practices (derived from the official guidance and typical agent reliability pitfalls):

- Provide the agent with the **exact CLI shape** you expect it to use (language flag, quoting rules, output mode).
- Ask for **structural evidence**: “show the pattern and the matched locations” before asking the agent to generate a patch. AST-grep supports JSON output for composability.
- Avoid relying on the model “knowing” AST-grep: the docs explicitly warn that if the model is not familiar, it may not use the tool effectively.

Supplying documentation to reduce hallucinated tool use

AST-grep provides an `llms.txt` / `llms-full.txt` bundle explicitly intended to be fed into an LLM’s context window to reduce incorrect rule generation, positioning it as a mitigation for tool hallucination and misuse.

This is particularly relevant when the agent is expected to author YAML rules or interpret output reliably—key steps in automating transformations.

Tool-augmented agents via MCP

The experimental AST-grep MCP server is a first-class “tool augmentation” mechanism: it exposes tools like dumping syntax trees, testing code against YAML rules, and searching codebases by pattern or rule. Its README claims this enables assistants to operate on code with AST pattern matching “rather than simple text-based search”, and it explicitly describes iterative rule development and debugging.

A subtle but material performance lever for AI agents is **token efficiency**: the MCP server documents a text output mode for searches that is “~75% fewer tokens” than JSON (at the expense of metadata), which can reduce latency and cost in tool-using agent loops.

Post-processing and guardrail integration

Post-processing means: let the agent generate a patch, then run AST-grep (and tests) as an automated validator and mechanical fixer. You can structure this as:

- **Reject**: if AST-grep finds disallowed patterns, the agent must revise.
- **Auto-fix**: if rules include `fix`, apply them.

- **Normalise:** use AST-grep rewrite to enforce stylistic invariants that your formatter/linter cannot easily express.

AST-grep's scan mode supports interactive review (`--interactive`) and also fully automatic application of rewrites (`--update-all`), enabling both human-in-the-loop and fully automated post-processing in agent pipelines.

Feedback-loop design

Robust agent integration benefits from *two* feedback loops:

1. **Syntactic loop** (AST-grep): iteratively refine patterns/rules until matches align with intent.
2. **Semantic loop** (tests/build): verify behaviour with compilation and test suites.

AST-grep supports explicit rule testing with valid/invalid cases and reports outcomes in a way that aligns with standard false positive/false negative concepts ("noisy" vs "missing"). This is directly usable as a feedback signal to an agent developing or refining rules.

```
flowchart LR
    U[User intent / ticket] --> A[LLM agent drafts change]
    A --> S[AST-grep search\n(structural retrieval)]
    S --> A
    A --> P[Patch proposal]
    P --> G[AST-grep scan\n(rule guardrails)]
    G -->|violations| A
    G -->|clean or auto-fixed| T[Build + tests]
    T -->|fail| A
    T -->|pass| M[Merge / deliver]
```

Concrete AI-agent examples and commands

Example: TypeScript safe optional chaining rewrite

This is a canonical AST-grep CLI rewrite example from the AST-grep repository: rewrite `a && a()` to `a?.()` in TypeScript.

```
ast-grep -p '$A && $A()' -l ts -r '$A?.()'
```

Agent usage pattern: - Agent identifies a codebase area where guard expressions precede calls. - Agent runs the command (possibly scoped to a folder). - Agent inspects diffs (or uses `--interactive` in `scan`) and then runs tests.

Example: Lint rule to flag `await` inside `Promise.all(...)`

AST-grep's rule guide provides a minimal TypeScript example using relational matching (`has` + `stopBy`) to find `Promise.all` calls that contain `await`, which is described as suboptimal parallelisation.

```
id: no-await-in-promise-all
language: TypeScript
rule:
  pattern: Promise.all($A)
  has:
    pattern: await $_
    stopBy: end
  message: "Avoid awaiting inside Promise.all; it serialises work."
  severity: warning
```

Run it against a file (as documented):

```
ast-grep scan --rule no-await-in-promise-all.yml test.ts
```

This is a strong “AI guardrail” pattern: an agent can be allowed to generate async code freely, but the pipeline blocks (or auto-fixes) anti-patterns that your team has decided to ban.

Example: Python structural rewrite for `== None` to `is None`

AST-grep supports Python (`--lang python` / alias `py`) as a built-in language.

```
ast-grep run --lang python -p '$X == None' -r '$X is None' .
```

This is intentionally “mechanical”: it improves stylistic correctness (and often lint compliance) without asking an agent to reason about every occurrence. Pair it with unit tests to guard semantics.

Integration patterns for human engineers and IDE-centric workflows

Local workflows: linting and refactoring in the editor

AST-grep provides an official VS Code extension that bridges: - a UI for the CLI, and - a client for AST-grep's LSP.

The extension supports: - structural search (pattern or YAML), - structural replace with preview/commit, - diagnostics and code actions, which require `sgconfig.yml` in the workspace root.

The editor integration page also documents that the LSP publishes diagnostics and supports fix code actions, and that it requires `sgconfig.yml` in the project root (or a configured path).

This enables a common workflow for senior engineers: - Prototype patterns in the playground or locally. - Promote them into versioned YAML rules. - Surface violations directly in the IDE, not just in CI, shortening the feedback cycle.

CLI workflows for humans

AST-grep supports: - **Interactive mode** (confirm/skip/edit) for rewrite sessions. - **JSON mode** for piping into other tooling. - Thread control (`--threads`) for performance tuning.

An engineer-friendly pattern is to begin interactively (to validate the rule) and then switch to automatic application once confidence is high:

```
# Review changes one by one
ast-grep scan --interactive

# Apply all rewrites once the rule is trusted
ast-grep scan --update-all
```

Team workflows: code review automation and CI/CD enforcement

AST-grep provides an official Action to run linting in a workflow. The repo documents inputs like `version`, `config`, and `paths`, and shows basic/advanced workflow snippets.

For CI code scanning interoperability, AST-grep scan supports output formatting including SARIF.

A common engineering-manager pattern is: - Run AST-grep scan in CI for pull requests. - Fail builds only on high-severity rules (policy-based). - Optionally upload SARIF results to the platform's code scanning UI for triage and auditability.

Real-world adoption evidence exists in public repos: a turborepo pull request explicitly discusses using the official ast-grep Action (instead of installing from npm), crediting a suggestion from .

Project configuration as a shared contract

AST-grep uses `sgconfig.yml` to define rule directories and test configurations; it is explicitly described as analogous to `tsconfig.json` or ESLint config, and is distinct from the rule YAML files themselves.

A minimal configuration shape:

```
ruleDirs:
  - rules
```



```
testConfigs:
  - testDir: rule-tests
```

This makes AST-grep a good “shared contract” tool: the same rule set can back IDE diagnostics, local scans, and CI enforcement.

```
timeline
  title Adoption timeline for a team (illustrative)
  week 1 : Pilot 3-5 rules on high-signal patterns
           : Add sgconfig.yml + rule repo structure
  week 2 : Enable IDE diagnostics for volunteers
           : Add CI scan in non-blocking mode
  week 3 : Add rule tests (valid/invalid cases)
           : Turn on auto-fixes for safe rewrites
  week 4 : Enforce blocking severity for critical rules
           : Integrate SARIF reporting for triage
```

Measuring impact on correctness, speed, and productivity

Metric categories to track

AST-grep is best evaluated as a “change accelerator with guardrails”. The following metrics are commonly meaningful for senior engineering stakeholders, and most can be captured automatically.

Category	Metric	How to measure	Why it matters for AST-grep
Match quality	False positives / false negatives	Use AST-grep rule tests (valid/invalid suites) and interpret outcomes as noisy/missing vs reported/validated	Directly measures whether rules encode intent; AST-grep explicitly frames testing in detection-theory terms.
Codegen correctness	Functional correctness (pass@k, tests pass)	Standard execution-based benchmarks like HumanEval / MBPP; for repo tasks, use SWE-bench-style “issue → patch → tests”	Establishes that structural rewrites didn’t degrade behaviour and that agent loops converge to working code.
Build health	Compilation / typecheck pass rate	CI signals before/after AST-grep enforcement	Detects “syntactically valid but semantically broken” rewrites; complements AST-grep’s syntactic focus.
Runtime performance	Scan time, rewrite time, CPU utilisation	Track duration of <code>ast-grep scan</code> in CI; use <code>--threads</code> and compare	AST-grep supports multi-threading controls and is positioned as multi-core performant.

Category	Metric	How to measure	Why it matters for AST-grep
Agent performance	Iterations to converge; tool-call token usage	Count agent loops; measure tokens returned by structural search (e.g., MCP text vs JSON output)	AST-grep can reduce retrieval noise and token volume in tool-augmented loops.
Developer productivity	Satisfaction/flow and efficiency metrics (SPACE)	Use SPACE dimensions and combine quantitative + qualitative measures	Avoids “lines of code” fallacies; supports rigorous team-level evaluation.
Delivery performance	DORA metrics (change fail rate, lead time, etc.)	Use DORA definitions and track via CI/CD and incident data	Captures whether faster refactors and reduced review burden translate into improved delivery.

Benchmarks and experimental designs

Because AST-grep is a *tooling primitive*, public benchmarks rarely isolate its effect directly. A rigorous evaluation therefore usually uses one of three designs:

Controlled offline evaluation with standard code benchmarks

For AI code generation: - **HumanEval** provides execution-based functional correctness for synthesising Python functions from docstrings, and the Codex paper uses pass@k on HumanEval as a standard metric.

- **MBPP** is a benchmark of ~1,000 “basic” Python tasks used to evaluate program synthesis from natural language.

- **SWE-bench** evaluates real-world GitHub issues where the model must generate a patch that resolves the issue; SWE-bench Verified is a curated subset intended to improve evaluation reliability.

Experimental method: - Baseline agent solves tasks normally. - Treatment agent uses AST-grep as a retrieval + post-processing + lint gate. - Compare pass rates, iteration counts, and latency/cost distributions.

Statistically: - Prefer paired comparisons per task (same task, two conditions). - Report confidence intervals over success rates; use matched-pair tests where appropriate (e.g., McNemar’s test for paired binary outcomes) and nonparametric tests for latency distributions if heavy-tailed. (Statistical method choice is context-dependent and remains unspecified here.)

In-repo workload evaluation

For human teams, synthetic benchmarks often miss real constraints (review norms, build times, monorepo tooling). A more predictive design:

- Select 5–10 recurring change classes (API rename, logging policy enforcement, forbidden patterns, migration scaffolding).
- Implement the change once with classic tools (manual/regex/codemods) and once with AST-grep rules plus IDE+CI integration.
- Track:

- time-to-land (lead time),
- number of review rounds,
- number of defects found post-merge,
- rule test precision/recall,
- and developer-reported friction/satisfaction (SPACE).

This aligns with the SPACE paper's premise that productivity is multidimensional and should not be reduced to a single activity metric.

Static-rule quality evaluation via detection theory

AST-grep's rule testing explicitly frames four outcomes (validated, noisy, missing, reported). That enables a classic detector-quality view:

- Precision = reported / (reported + noisy)
- Recall = reported / (reported + missing)

You can treat each rule independently and enforce minimum thresholds before enabling the rule as blocking in CI.

What “performance improvement” realistically looks like

Based on AST-grep's documented strengths and constraints, the most plausible measurable improvements are:

- **Fewer erroneous bulk edits than regex-based refactors**, because matches are syntax-scoped and do not hit strings/comments in the same way.
- **Faster large-scale mechanical changes**, because AST-grep is designed for multi-core scanning and provides automation paths (`--update-all`, CI action) rather than manual editing.
- **Lower agent iteration count and token usage** in tool-augmented workflows, if AST-grep is used to retrieve precisely-scoped code snippets and to apply consistent rewrites.
- **Improved maintainability via enforced invariants**, when teams encode architectural rules and code-smell detection into versioned rule sets surfaced directly in IDEs and CI.

These are hypotheses that should be validated with the experimental designs above; the exact magnitude is unspecified and will be highly codebase-dependent.

Best practices, limitations, security considerations, and adoption roadmap

Best practices for robust rule authoring

High-signal patterns from AST-grep's own documentation and schema design:

- Start with **patterns** for quick exploration, then graduate to **rules** for relational constraints and context (e.g., `has`, `stopBy`) when you need precision.

- Treat YAML rules as “programs”: add **tests** (valid/invalid cases) before enabling them broadly, using AST-grep’s testing tooling and the `testConfigs` configuration in `sgconfig.yml`.
- Use `constraints` to filter meta-variable matches after the main rule matches; AST-grep documents that constraints are applied after matching and are per single meta-variable.
- For refactors, prefer **interactive mode** initially, then move to `--update-all` only after you have evidence the rule is stable (tests + sample diffs).

Limitations and failure modes

AST-grep’s most important limitations (explicitly documented):

- It is primarily syntactic and does not provide type information, control/data flow analysis, taint analysis, or other deeper semantic capabilities (in the comparison document’s wording).
- Patterns must be valid parseable code; incomplete fragments may require more context or different rule styles.
- Overly broad patterns can still match unintended constructs (structural false positives), which is why rule testing and constrained rules matter.

For teams relying heavily on semantic guarantees, consider pairing AST-grep with: - type checking, - test suites, - or complementary analysers (e.g., Semgrep taint mode where appropriate).

Security and privacy considerations

Tool-using agents and prompt injection

When AST-grep is used inside AI agents (especially tool-augmented agents that ingest untrusted code/comments/issues), you inherit the broader security risks of tool-using LLM systems:

- OWASP explicitly lists prompt injection as a top risk category for LLM applications.
- Research on prompt injection against LLM agents highlights that integration with external tools and sources creates a path for attackers to influence tool selection and actions.

Practical mitigations (high level, since environment constraints are unspecified): - Run AST-grep locally/on trusted infrastructure where possible, minimising code exfiltration. - Scope tool permissions (read-only search vs write-enabled rewrite) and require human approval for broad rewrites. - Treat tool outputs and retrieved code as untrusted inputs in the agent’s instruction hierarchy, consistent with LLM security best practice.

MCP-specific considerations

The MCP specification positions MCP as a standard protocol for connecting LLM apps to external tools/resources and defines server “tools” as invokable capabilities.

If you expose AST-grep via MCP: - constrain accessible filesystem roots, - prevent arbitrary command execution beyond the intended AST-grep surface, - and use MCP transport/authentication guidance where applicable (details depend on whether you use STDIO vs HTTP and are unspecified here).

CI reporting and data handling

If you emit SARIF and upload it to code scanning systems: - SARIF is a standard format for static analysis tool results (OASIS spec). - Platforms like GitHub document SARIF ingestion and code scanning support.

Treat SARIF artefacts as potentially sensitive (paths, snippets, rule metadata), and apply normal CI artefact retention policies accordingly (policy details are unspecified).

Migration and adoption roadmap for teams

AST-grep is MIT-licensed and distributed via multiple package managers (npm, pip, Homebrew, Cargo, etc.), which lowers direct licensing cost but does not eliminate adoption cost (rule authoring, training, CI time).

Because no specific constraints were provided, the roadmap below is written as a generic template.

Phase-based adoption

- **Foundation**

- Install AST-grep via your standard developer bootstrap.
- Add `sgconfig.yml`, define `ruleDirs`, and establish a `rules/` repo layout.

- **Pilot**

- Implement 3–10 high-signal rules (security policy, reliability patterns, banned APIs, architectural boundaries).
- Write rule tests (valid/invalid) and require passing tests before merging rule changes.

- **Integrate with daily workflow**

- Enable IDE diagnostics/code actions for volunteers (VS Code extension + LSP).
- Run CI scans in warning mode first (non-blocking), then graduate to blocking for critical rules.

- **Scale and govern**

- Add SARIF outputs for auditability where useful.
- Version-pin CI actions for stability (mirroring the turborepo discussion that emphasises using the official action and pinning).

Cost model (illustrative; constraints unspecified)

- Direct software cost: typically low (open-source MIT licence).
- Main costs:
 - engineer time to develop/maintain rules + tests,
 - CI runtime (CPU minutes) for scans,
 - and change management (training, documentation, rule governance).

AST-grep can reduce costs elsewhere (manual refactor time, review cycles), but those savings are empirical and should be measured with SPACE/DORA-style outcomes.

Recommended adoption checklist

- Establish ownership: rule maintainers and review policy.
 - Create `sgconfig.yml` and standard directories (`rules/` , `rule-tests/`).
 - For each rule:
 - document intent (what it catches and why),
 - add valid/invalid tests,
 - track precision/recall from test outcomes,
 - and decide whether it is advisory vs blocking.
 - Integrate IDE support (VS Code extension and/or LSP clients) and confirm diagnostics work in representative repos.
 - Add CI scanning:
 - start non-blocking,
 - then block on error severity,
 - optionally emit SARIF for code scanning UI integration.
 - Define a safe rewrite policy:
 - prefer `--interactive` initially,
 - allow `--update-all` only for rules with strong test coverage and low risk.
 - If using with AI agents:
 - provide `llms.txt` /documentation context,
 - restrict write access and filesystem scope,
 - and treat prompt injection as a first-class risk.
-