Engineer

Linter

🖋 a tool that flags likely bugs in code without running it

What it is

A linter is a program that reads source code and reports the parts that look like mistakes, without building the program or running it. Each report names a rule, a file and a line: a variable assigned and never used, a comparison that is always true, a shell variable left unquoted, a format string built from user input.

The name comes from lint, a checker for C programs, and NetBSD still ships one. Its manual gives the job as detecting features of C program files “that are likely to be bugs, to be non-portable, or to be wasteful”, and adds that it does stricter type checking than traditional pre-C90 compilers. It works in two passes: first each file is checked as a separate unit, then the results are cross-checked for inconsistencies between files.

Most languages now have linters of their own, often several:

- **ESLint** for JavaScript.
- **Ruff** for Python.
- **ShellCheck** for shell scripts. Its stated goals run from beginner syntax issues to subtle caveats that may make a working script fail under future circumstances.
- **Stylelint** for CSS, **markdownlint** for Markdown and **yamllint** for YAML.

Every linter is built from rules. Each rule has an identifier, such as F841 in Ruff or SC2059 in ShellCheck, and a severity. A configuration file decides which rules are on and how serious each one is, and many rules come with an automatic fix.

A formatter rewrites layout and has no opinion on meaning; a type checker reasons about what each value is. A linter sits between them and looks at how the code is written: misuse of the language, risky patterns and the conventions a project has agreed on.

Use the concept in a decision

Put “What is a linter?” to work in a real decision. We can help relate the concept to the system you are building or buying. One team taking an idea from a sketch to something your customers can log into, instead of handing it between three suppliers.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Led a GIS map application to 10x revenue growth — 10x revenue growth, led from the map product

Led a GIS map application to 10x revenue growth Get in touch →

Why it matters

A linter reads every branch, including the error handler that runs once a year and that no test reaches. It takes seconds over a whole repository, so it can run on every save and every commit, and a finding costs least at the moment the line is written.

Rules settle arguments once. Whether a bare `except:` is acceptable, or whether every shell variable must be quoted, is decided in the configuration and then applied the same way to every author and every review. Human reviewers keep their attention for the questions a rule cannot express.

The exit code is the interface between a linter and whatever runs it. ShellCheck’s README says the tool “makes canonical use of exit codes”, so a build can call it as one more command. Ruff and ESLint document the same three-way convention: 0 when the check is clean, 1 when it found violations, and 2 when the tool itself failed, for instance on an invalid configuration. A gate can then tell “the code is wrong” apart from “the checker is broken”, and both should stop it.

Severity decides which findings reach that exit code. ESLint exits with success when it finds warnings and no errors, unless `--max-warnings` sets a limit, and lint(1) has `-w` to treat warnings as errors. A project that wants its linters to block mistakes, and not only describe them, writes that choice down where every run uses it:

```
# every finding fails the run, warnings included
lint -w -abcFghprSxz source.c
eslint --max-warnings 0 src/
ruff check src/
shellcheck scripts/*.sh
```

This site takes the strict end of that choice. Its own gate counts every diagnostic as an error and prohibits inline suppressions such as `# noqa`, so a finding is fixed at its cause.

What it looks like in practice

A linter's configuration often begins as whichever rules happened to be quiet on the day someone set it up. One page turns that around: every rule of a Python linter enabled as an error. The first run produced 1,815 findings. They were worked through by category rather than by file, because the categories carry information: unused arguments and shadowed builtins were noise, while the security, mutable-default and exception-handling categories each pointed at real behaviour. The few exemptions carry a written reason at the line, and two of them are backed by a test that asserts the property the disabled rule would have enforced. Strict type checking ran alongside and caught a different kind of defect, about what a value is rather than how it is written. The cost is stated too: someone has to make a judgement 1,815 times.

Another page starts from an awkward discovery. The site's quality gate, thirty-two scripts and 10,242 lines of JavaScript, was the largest body of code in the repository and the only one that no formatter, linter or type checker read. Bringing the gate's own code under a linter started with the formatter, set to two-space indentation for those files so that 38 lines changed instead of 2,173. The linter then produced thirteen real findings, all fixed and none suppressed, and a type checker found two defects the linter had passed. The reason for the effort is stated with it: a defect in a checker shows up as a check that passes when it should not.

The idea also works on prose. A third page describes a linter for the repository's own documentation, with limits that fail rather than advise. Its eight mechanical policies include a per-file line budget that can only shrink, a rule that a plan's status must agree with its own checkboxes, and a check that every path and code symbol a document names exists in the tree. The first run found six plans

marked done with unchecked boxes, fourteen tasks gating every commit that appeared in no document, three broken links, one orphaned document and three dead paths.

Where it goes wrong

A linter seldom fails loudly. The usual fault is a run that exits 0 when it should have stopped something.

Findings that never change the exit code. ESLint's `--max-warnings` defaults to -1, which switches the limit off, and lint(1)'s manual says its cross-file pass always exits successfully. A finding that cannot fail the run is read once and then scrolled past. Decide per rule: an error that fails the run, or off, and set any warnings-as-errors switch in the shared configuration.

A fix that reports success. Ruff exits 0 when every violation it found was fixed automatically. That is convenient at a desk, but a gate running with fixes on repairs only its own working copy, and the commit that arrived still holds every violation. Run the gate without fixes, or add `--exit-non-zero-on-fix` so a fix counts as a failure. Search for `--exit-zero` too: Ruff exits 0 with it even when violations are found, and a flag added for one noisy migration tends to stay.

A pattern that selects zero files. A renamed directory or a moved configuration can leave a linter pointed at an empty set, and checking zero files is a clean result. ESLint treats a quoted glob that matches zero files as an error by default. `--no-error-on-unmatched-pattern` turns that into a quiet pass, and `--pass-on-no-patterns` exits 0 when no file patterns are given. Keep the default, and keep one file with a known violation that the gate must report.

A rule set chosen by what was quiet. A configuration built by switching off whatever fired on day one records the code's habits, not a standard. Start from a deliberate selection, either everything at once or Ruff's suggested small base of `E` and `F` with one group added at a time, and write down why each disabled rule is off. Ruff warns that `ALL` implicitly enables new rules whenever the tool is upgraded, so pin the linter's version and upgrade it as a change of its own.

Suppressions spread one line at a time. Every linter has an escape hatch: `/* LINTED */` in lint(1), `# noqa: F841` in Ruff, `# shellcheck disable=SC2059` in ShellCheck and `// eslint-disable-next-line` in ESLint. Each records what was

silenced, not why. Require a written reason beside every exemption, and for the ones that matter, switch the rule off and add a test for the property it protected. Keeping exceptions in one place helps too: ESLint's `--no-inline-config` makes disable comments ineffective, and ShellCheck reads project-wide `disable` lines from a `.shellcheckrc` file.

Further reading

- lint(1) — NetBSD's manual for the C checker the word comes from: its two passes, `-w` and its suppression comments.
- The Ruff Linter — rule selection, fix safety, suppression comments and exit codes.
- Command Line Interface Reference — ESLint's `--max-warnings`, inline configuration, unmatched patterns and exit codes.
- ShellCheck — the project's README: what it looks for and how it fits into a build.
- Ignore — ShellCheck's ways to ignore a finding per line, per run and per project.



Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is a linter?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/linter/