ELLM

How to Understand the Verification Gates

This guide explains the verification process used by eLLM Code, detailing each step from running tests to browser checks, and how to interpret the results. It also covers troubleshooting tips and settings adjustments for effective use.

AUTHOR
Paul Flanders

PUBLISHED
10 Sep 2026

DOCUMENT
InfoHub — Knowledge, news, and updates from EssingtonITS

How to Understand the Verification Gates

Paul Flanders · 10 Sep 2026

Estimated time: 5 minutes
Difficulty: Intermediate

Why you'd use this

A coding assistant that says "done" is only useful if "done" is checked. eLLM Code runs a series of deterministic gates before it reports success: it checks imports resolve, runs the project's own tests or build, formats and lints, loads web apps in a headless browser, and then has an independent review pass judge whether the request itself was met. This guide explains what each gate does, how to read the evidence in the chat, and which settings turn them on and off.

Before you start

Permissions required:

- Ability to run the project's toolchain locally.

You'll need:

- A project with a recognisable test or build command (npm, pytest, go, cargo, dotnet, maven, gradle and around a dozen more are detected), or a `package.json` script named `test`.
- For browser verification: Chrome or Chromium installed, or the `ELLM_CHROMIUM` environment variable pointing at a browser binary.

Steps

1. Open ⚙ in the chat panel and check the gate toggles: **Test / build gate** and **Format / lint gate** are on by default. In VS Code settings, `ellmCode.autoVerify`, `ellmCode.requestJudge`, `ellmCode.lintOnWrite` and `ellmCode.summarizeRuns` are also on by default.
2. Make a code change through the chat with **Auto-run commands** on. After the change lands, an **Auto-verify** line shows the detected test or build command running.
3. Read the outcome line. "Tests: passed" with the runner's totals is strong evidence. "exit-only" means the command exited 0 but printed no totals, which is weaker and is reported as such. A red result is fed back and the assistant fixes it, up to three rounds.
4. In a Plan Mode build, watch the gates run when the last task is ticked: a static import and export check, the test or build gate, the formatter and bug-focused linter, then for a web app a browser load that captures console errors.
5. After a green gate, a **Judge** line shows an independent pass comparing the original request with the changed files. Gaps are fed back for one corrective round.
6. Read the closing recap. It comes from the record of what actually happened, listing files applied, commands run and the verification status, not from the model's own claims.

What you should see

Evidence is tied to the files it covers. If a later change touches source, an earlier green result is marked stale and the tests run again. A command that rewrites source files, such as a formatter or code generator, is detected and also invalidates earlier evidence. A generated test runner that swallows failures is flagged the moment it lands.

Troubleshooting

- **"skipped, no test/build command detected":** add a `test` script to `package.json` or the equivalent for your ecosystem. If the assistant ran a test command itself during the request, that command is reused as the gate.
- **"Preview unavailable" or "no headless browser":** install Chromium or set `ELLM_CHROMIUM`. The build completes but is labelled unverified in the browser.
- **The gate times out:** gates are capped at three minutes. Timeouts are reported as unresolved, never as passes. Split slow suites or raise `ellmCode.commandTimeout`.
- **Verification gave up with errors:** after two errored preview attempts the build finishes with the errors handed to the Bug-fixing agent. Read its report before trusting the result.

Related guides

- How to Preview a Web App and Capture Runtime Errors
- How to Run Commands from the Chat
- How to Run a Specialist Agent