PYTHON DEVELOPMENT / JAVASCRIPT VALIDATION

How to validate *JavaScript in Python*

Check JavaScript in AI-generated HTML before returning it to users. A comparison of seven Python-compatible options using 607 test samples, with working code and the limits of each approach.

15 NOVEMBER 2026 · 607 TEST SAMPLES · LINUX / PYTHON 3.13

For Python developers validating generated HTML without installing Node.js.

607	7	302 / 305	0
JavaScript test samples	tested parser setups	valid / invalid samples	submitted programs executed

01 / RECOMMENDATION

Recommended for this workload: Acorn with mini-racer

Use `Acorn 8.19.0` through `mini-racer==0.14.1` when you need a strict syntax check in a Python application. mini-racer is a Python package that includes the V8 JavaScript engine. V8 runs the trusted Acorn parser; Acorn reads the generated source as text. The generated program is never executed, and Node.js is not required.

> **Scope of this comparison**
> The input is JavaScript extracted from HTML generated by a large language model (LLM). The application runs in a Python container without Node.js. Python packages containing compiled native code are allowed. The goal is to catch invalid source without unnecessarily rejecting valid modern JavaScript.
>
> A parser reads source and checks language rules. A JavaScript engine runs JavaScript. Here the engine runs only the trusted parser. It never runs the program being validated.

Terms used in the results

Script and Module

Two JavaScript parsing contexts. An ordinary inline `<script>` uses Script; `<script type="module">` uses Module. Imports and top-level `await` depend on this choice. A “parse goal” means this selected context.

ECMAScript / ES2025

ECMAScript is the JavaScript language specification. ES2025 identifies the edition used to label these test samples.

Early error

A language-rule violation detectable before running code, such as declaring `let x` twice in the same scope. A parser may recognize the code’s structure without checking this rule.

Scope and binding

A scope is the region where a declared name is available. A binding connects that name to its declared value or variable.

Strict mode

A JavaScript mode that applies additional restrictions. Modules use it automatically; classic scripts can enable it with `"use strict"`. A “sloppy” script means a classic script outside strict mode. This is separate from a “strict parser,” which reports errors rather than accepting malformed input through error recovery.

Native wheel

A prebuilt Python package containing compiled code for a particular platform. mini-racer includes the V8 engine this way; it does not install Node.

Worker and engine context

A worker is the process that performs validation. An engine context is its JavaScript environment, which can keep the trusted parser loaded between requests.

RSS / MiB

Resident set size (RSS) measures process memory held in RAM. MiB means 1,048,576 bytes. The report uses peak observed worker RSS.

Corpus

The complete set of source samples used for testing. Each sample has a source string, parsing context and expected result.

JSX, AST and regular expressions

JSX is a syntax extension often used with React; browsers do not accept it as ordinary JavaScript without transformation. An abstract syntax tree (AST) represents parsed code. A regular expression, or regex, is a text-matching pattern such as `/a+/`.

DOM and CSP

The Document Object Model (DOM) is the browser's representation of the page. Content Security Policy (CSP) controls which resources and scripts the browser permits.

The test set contains 302 valid and 305 invalid JavaScript samples. Acorn with V8 rejected none of the valid samples and missed one invalid sample. It also accepted the separate valid nesting test at depth 300. These measurements support the recommendation for this use case. They do not establish a production error rate or correctness for every possible program.

Reuse a long-running worker process so you do not initialize the engine for every request. The V8 worker used approximately 103 MiB of memory at its measured peak. If that is too much, consider Acorn with QuickJS-NG, which peaked at approximately 33 MiB. Report its tested nesting limit as an inconclusive check rather than invalid source.

> **Tree-sitter detects fewer invalid programs.** It rejected 3/302 valid samples and missed 259/305 invalid samples. Its speed and error recovery are useful for structural warnings. For final syntax checking, Acorn caught more errors without rejecting additional valid samples in this test set.

> **The modern Esprima fork had the best classification result.** `esprima-fork==5.0.0` recorded no false rejections or missed invalid cases. Its maintenance evidence is less established: 0 repository stars, 129 monthly distribution downloads, and a roughly four-year gap before the latest release. Deeply nested valid input also reached its resource limit. Choosing Acorn accepts one documented missed error in exchange for stronger adoption evidence and tested nesting tolerance.

Choose mini-racer or QuickJS-NG based on memory and input size

Both hosts run the same Acorn parser, so their matching syntax results are not independent confirmation. QuickJS-NG uses less memory in this test and its Python package was released more recently. The V8 route has broader listed platform support and more downloads for its Python package. It also handled deeper input under the tested defaults. mini-racer was last released in February 2026; assess that maintenance gap separately from V8 itself.

Handle uncertain results separately. Request source repair for a syntax error only after selecting the correct Script/Module context and language version. A timeout, unsupported configuration, resource limit or crash is `inconclusive`. Use a warning or review path for uncertain results when false rejection is especially costly. Accepted syntax does not guarantee successful rendering.

Choose a check based on what your application needs

Choose the level of checking your application needs. A basic parser can spot incomplete source. A stricter validator also checks rules such as duplicate declarations. Browser tests can check specific UI behavior after the source is loaded. Each option below states the result you can rely on and the failures it can miss.

01 / LIGHT CHECK

Light check: Tree-sitter for incomplete or malformed code

Recommended: Tree-sitter 0.26.1 with JavaScript grammar 0.25.0.

Inspect `root_node.has_error`, then walk both `ERROR` and `MISSING` nodes. Return `NO_STRUCTURAL_ERROR_FOUND` or `NEEDS_REVIEW`. This catches incomplete expressions and unclosed source while retaining useful error locations.

Use it for: editor feedback, cheap first-pass diagnostics, or workflows where missed invalid code is acceptable. Keep it advisory if false rejection is the highest concern.

Accept the cost: `let x; let x;`, `break;`, `const x;` and JSX can produce clean trees. The benchmark recorded 259 missed invalid cases and three false alarms. That is not a production rate, but it establishes that this is not a complete JavaScript validator.

02 / GOOD DEFAULT

Stricter check: Acorn for syntax and context rules

Recommended: Acorn 8.19.0 through mini-racer 0.14.1. The Python wheel supplies the engine; your container does not need Node.

Use the actual Script/Module goal, an explicit edition, and ordinary strict-parser options. Pass generated source as an argument to the trusted parser. Reuse one context per worker, and keep source size, wall-clock time and memory bounded.

Use it for: a check before delivery for generated source where low false alarms and useful invalid detection both matter. It accepted all 302 valid samples and missed one of 305 invalid samples.

Accept the cost: a native engine, roughly 103 MiB measured worker peak RSS, startup overhead, separate parser/host updates, and a documented static-block early-error miss. An accepted result means syntax accepted, not “the UI works.”

03 / PURE PYTHON

Pure-Python option: the modern Esprima fork

Recommended candidate: esprima-fork 5.0.0, after testing it against representative source from your application. Set `ecmaVersion=2025`, `tolerant=False`, and `jsx=False`.

Use it when: Python portability or native-wheel restrictions outweigh the maintenance

04 / STRONGEST ASSURANCE

Browser tests: check whether the page works

No parser offers a rendering guarantee. Start with the strict syntax check, then use an isolated browser test stage with the same origin, CSP, assets, dependencies and supported browsers as production. Test the expected interactions and collect console, network and DOM failures.

uncertainty. It agreed with all 607 labels in this study, including the Acorn counterexample.

Accept the cost: limited independent adoption and a long historical release gap. Valid nesting hit the resource limit at depths 100 and 300. Return `INCONCLUSIVE`, not invalid syntax, for these diagnostics. Install it alone: all three Esprima distributions use the `esprima` import namespace.

This stage needs infrastructure beyond the Python-only parser container: an independently provisioned browser worker or CI service. It was not implemented or benchmarked in this study.

For higher syntax assurance, a second parser may be useful as a disagreement detector. Do not blindly require both parsers to pass: that can increase false rejection when either parser has a limitation. Send disagreements and infrastructure failures to review.

Describe the test result: "passed these source checks and these browser scenarios under these pinned conditions." Arbitrary JavaScript behavior cannot be universally guaranteed by a finite test suite. Security review and isolation are separate requirements.

Decisions to make before installation

1. Check whether compiled Python packages are allowed. If they are, Acorn's engine hosts are eligible. Otherwise, test the modern fork against your application inputs. Acorn avoids Node.js but still needs a JavaScript engine.
2. Decide whether the result is a warning or a requirement before delivery. Tree-sitter is reasonable for warnings. A delivery requirement needs a context-aware parser that reports syntax errors and separates failed checks.
3. Check the worker memory budget. If approximately 103 MiB is too high, consider Acorn + quickjs-ng, whose worker peaked around 33 MiB. Its default stack reached a limit at depth 300; stack tuning was not tested.
4. Add a separate browser-test stage when you need evidence that the application works. Missing functions, broken DOM selectors and failed imports remain outside the syntax result.
5. Keep uncertain failures separate. Timeouts and unsupported syntax extensions do not necessarily mean broken source. Preserve these distinctions in your API.

Reasons for recommending or rejecting each option

The recommendation considers incorrect rejection of valid code, detection of invalid code, deployment requirements, measured limits and maintenance. The fork had the best classification result in this test set. Acorn is the overall recommendation because of its adoption and tested nesting tolerance, despite its documented missed error. Stars are not added to an accuracy score.

Candidate	Concrete shortcoming	Decision
Acorn + mini-racer	Accepts `class C { static { const f = () => arguments; } };` native footprint and older wrapper release.	Best overall default for a persistent worker, not perfect conformance.
Acorn + quickjs-ng	Same Acorn miss; depth-300 default-stack limit. Smaller and faster startup.	Preferred when memory/startup budget outweighs tested depth tolerance.
esprima-fork	No corpus classification errors, but nesting limits and very small adoption.	Best measured pure-Python alternative; assess maintenance and test your application inputs.
Tree-sitter JavaScript	Clean trees accept duplicate bindings, illegal control flow, JSX and regex errors. Valid non-strict `(yield) => 1` was rejected.	Structural diagnostics, not the authoritative validity gate.
oxc-python	The binding exposes parser errors without Oxc's semantic syntax-checking pass. 153 invalid cases passed.	Assess the checks this Python wrapper actually exposes.
esprima2	10 valid cases rejected; 102 invalid cases passed, including missed declaration and parsing-context errors.	Portable, but weaker validation in this comparison than the stronger candidates.
Original esprima	ES2017-era grammar rejects modern optional chaining and other current constructs; 112 false rejects.	Poor fit for modern LLM output. Popularity does not establish support for newer syntax.

Researched but not in the seven-run benchmark

- pyswc / SWC bindings: wrapper age, platform coverage and limited controls for Script or Module mode prevented a like-for-like deployment recommendation. This does not rank SWC core as a poor parser.
- Babel parser: valuable for JSX/TypeScript/proposals, but extra bundling/options and verification of regular-expression pattern checks make it a different integration. Not measured here.
- Meriyah: regular-expression validation that depends on the engine running the parser and explicit configuration of checks on declarations and scope require separate qualification. Not measured here.
- QuickJS compile-only APIs: the C capability is not automatically exposed by every Python wrapper. Do not call a wrapper's `eval` or `module` execution API and label that syntax-only validation.
- `new Function(source)`: changes the parse goal to a function body, incorrectly accepts a top-level `return` for a classic script, and does not parse modules correctly.

- Regex/bracket counting: cannot resolve comments, literals, template substitutions, lexical rules, automatic semicolon insertion or scope. A quick heuristic is not a language parser.

Return a result that separates bad source from failed validation

CODE EXAMPLE

```
{
  "status": "invalid",
  "stage": "javascript-syntax",
  "scriptIndex": 2,
  "sourceType": "module",
  "parser": "acorn@8.19.0",
  "diagnostic": {
    "message": "Unexpected token (1:8)",
    "line": 1,
    "utf16Column": 8
  }
}
```

This response schema is an example, not a verbatim benchmark record. Preserve the parser's actual error message. Record which script failed and whether locations count lines, UTF-16 code units or another unit. Locating an error in the original HTML also requires a tested mapping from extracted source to document positions.

Suggested caller policy: `valid` proceeds; `invalid` requests a repair based on the error message; `inconclusive` retries under a controlled policy or goes to review. A validator memory failure should not trigger an endless code-regeneration loop. Store the original source and diagnostics to build a production regression set alongside this article's test set.

03 / WALK THROUGH EXAMPLES

Follow a script from HTML extraction to an error message

The result depends on the exact text extracted from HTML and the rules the parser applies. These examples show a missing expression, a duplicate declaration, modern syntax, a Script/Module mismatch and the invalid class definition that Acorn missed.

These walkthroughs display recorded benchmark results. They do not validate new input. The surrounding HTML and step descriptions explain how an application would use the parser. None of the example source is executed.

Animations start only when you press Play. Every stage is also available as a button.

Three examples of syntax and context checks

`const x=;` → grammar failure: a variable initializer needs an expression.

`let x; let x;` → Tree-sitter constructs a clean tree, but Acorn rejects a duplicate lexical binding.

`await 1;` → invalid as a classic script; valid at the top level of an ES2025 module. The HTML `type` attribute determines which parse goal you must use.

A syntax tree can still describe invalid JavaScript

A parser such as Tree-sitter can recover from malformed input and still return a useful syntax tree. Even an error-free tree may omit rules such as "this name is already declared in this scope." An abstract syntax tree (AST) is a structured representation of code. Producing an AST does not by itself establish that the program is valid.

04 / MEASURED RESULTS

Results from 302 valid and 305 invalid samples

A **false rejection**, or false alarm, means valid JavaScript was marked invalid. A **false acceptance**, or missed error, means invalid JavaScript passed the check. The tables keep these counts separate because preventing false alarms is the primary requirement.

Pinned candidate	Valid wrongly rejected / 302	Invalid missed / 305	Inconclusive	Worker parse median µs
Acorn 8.19.0 + mini-racer 0.14.1	0	1	0	479.1
Acorn 8.19.0 + quickjs-ng 0.18.0.1	0	1	0	347.0
esprima-fork 5.0.0	0	0	0	287.7
esprima2 6.0.0	10	102	0	200.3
Tree-sitter 0.26.1 / JS grammar 0.25.0	3	259	0	46.8
oxc-python 0.1.1	0	153	0	7.6
Original esprima 4.0.1	112	43	0	212.6

The test set combines 107 manually designed examples with 500 distinct files from Test262, the official ECMAScript test suite. Expected syntax results for the 500 files come from their test metadata. The programs were parsed without execution. The next table shows this 500-file subset separately.

Standards-only subset	False rejects	False accepts	Inconclusive
acorn-v8	0	0	0
acorn-quickjs	0	0	0
esprima-fork	0	0	0
esprima2	10	88	0
tree-sitter	3	210	0
oxc-python	0	116	0
esprima	94	31	0

The invalid class definition that Acorn accepted

CODE EXAMPLE

```
class C { static { const f = () => arguments; } }
```

JavaScript forbids this `arguments` reference inside a class static block. The arrow function does not create its own `arguments` binding, so the restriction applies inside it too. An ordinary function can use its own binding. Both Acorn hosts accepted the invalid example; the modern fork rejected it. The expected result was checked against the ES2025 ContainsArguments rule. The benchmark does not patch Acorn to hide this miss.

The case explorer shows every source sample and recorded result. The downloads include the complete data for independent analysis.

Three valid samples that Tree-sitter rejected: a hashbang line starting with `#!` and ending at Unicode line separator U+2028; a comment containing a carriage return; and an arrow parameter named `yield` in non-strict JavaScript. The comment permits automatic semicolon insertion, a JavaScript rule that can insert a statement boundary. HTML preprocessing converts carriage returns (CR) to line feeds (LF) before JavaScript parsing. The raw-CR test therefore checks the parser directly; it does not establish that extracted HTML causes the same failure. Leading text or newlines also affect hashbang placement. All three raw-JavaScript labels remain valid, but their relevance to generated HTML differs.

Search the exact source and result for each test

Filter the recorded results by parser and outcome, then open a case to see its exact source, expected label and error message. The explorer contains all 607 test samples, including 500 distinct Test262 files. It does not run a live validator.

607 of 607 samples match. Original recorded results; no live evaluation.

The source viewer is text-only. Expected labels come from the corpus, not from the selected parser. Use the raw downloads below for independent analysis.

06 / TEST METHOD AND LIMITS

Test selection, expected results and execution limits

- Each of the seven fixed-version setups runs in a fresh Linux/Python 3.13 job with the same test samples.
- The Test262 manifest records original paths, the saved repository version, source hashes, parsing context and any added strict-mode directive. It also records excluded tests.
- Only Test262 tests marked as parsing-time `SyntaxError` failures are labeled invalid. Expected runtime failures remain syntax-valid. Tests of module loading/resolution and unsupported proposals are excluded.
- A separate worker process runs each parser. The parent process enforces a five-second limit for each input. It terminates a worker that exceeds the limit and starts another for subsequent inputs. Timeouts and failed checks are inconclusive, not successful error detection.
- Acorn and the modern fork explicitly select ES2025 and Script or Module. The other packages use their shipped grammar and are compared with the same expected results.
- The three Esprima distributions use the same `esprima` import name. Each is installed in a separate environment to avoid mixing package files.
- Separate AI-agent reviews checked sample labels, language rules, maintenance evidence and the final article. The agents used the same model family, so their agreement is not independent-model confirmation.

The same selection rules produce the same 500 Test262 files. The files were not randomly sampled from production traffic, and related tests share language features. This study does not measure a production false-positive rate or run the full Test262 suite. No representative history of actual model-generated output was supplied. Before deployment, add your own accepted and rejected outputs to a regression test set.

The Test262 snapshot is from May 31, 2025, and the selection uses a conservative feature list. It is not a complete inventory of the final ES2025 edition. The 500 files span 195 directories, with more examples from class, generator and destructuring tests; only 24 files use Module mode. A feature tag does not prove that syntax appears in the parsed source. For example, a test of the RegExp constructor may contain no regular-expression literal. The separate modern-syntax examples cover additional cases, but neither subset is

exhaustive. An AI agent from the same model family, working in a separate context, checked all 500 labels, source transformations and hashes against the saved upstream archive.

The selection manifest records each original Test262 path and source hash. Test262 notices and its license are preserved. Expected results for the 500 Test262 files were derived from official metadata. No expected result was selected by asking the preferred parser.

07 / MEMORY AND PERFORMANCE

Compare startup time, memory use and input limits

Candidate	Adapter init ms	Worker peak RSS MiB	Depth 100	Depth 300	112,779-byte valid program
acorn-v8	318.0	102.8	valid	valid	valid; 15.6 ms
acorn-quickjs	23.8	32.7	valid	resource_limit	valid; 149.9 ms
esprima-fork	8.4	33.3	resource_limit	resource_limit	valid; 185.6 ms
esprima2	32.6	37.0	validator_failure	validator_failure	valid; 92.7 ms
tree-sitter	1.1	32.7	valid	valid	valid; 16.3 ms
oxc-python	0.8	32.8	valid	valid	valid; 3.3 ms
esprima	292.8	132.0	validator_failure	validator_failure	valid; 109.9 ms

The median parse time includes the parser call and conversion of its result through the Python binding. It excludes communication between the parent process and its worker. Initialization includes imports and loading the trusted parser, but excludes launching Python. Peak RSS measures the worker's highest observed resident memory use across the run. It includes imports and earlier tests, so it is not memory allocated by one sample or a configured memory limit. Each size test ran once. Jobs used separate hosted runners, so hardware variation limits timing comparisons.

The valid nesting tests at depths 32, 100 and 300 are separate from the 607 classification samples. So are the flat programs of 929, 10,279 and 112,779 bytes. The raw results also retain repeated end-to-end timings measured after initialization. Concurrency scaling, long-running leaks, browser compatibility and hardened sandboxing were not certified. Set operating-system or container memory limits for production workers.

The nesting tests used each host's default stack settings. QuickJS-NG allows stack-size configuration, so its depth-300 limit does not establish an absolute maximum. Stack budgets were not tuned across hosts. These results compare the tested defaults.

Deployment role of each library

- **Tree-sitter:** fast, mature parsing ecosystem; helpful recovery and source ranges. It includes JSX and omits many early-error rules. A clean tree is not a browser-validity guarantee.
- **Original Esprima:** stale modern-syntax coverage; false rejections make it a poor fit here.
- **Modern Esprima fork:** strongest tested pure-Python strict option; smaller adoption and depth constraints.
- **esprima2:** portable but misses scope/context errors found by the stronger validators.
- **Oxc Python wrapper:** fast native parser; the inspected v0.1.1 binding omits Oxc's semantic syntax-checking pass. Its results do not evaluate the full Oxc toolchain.
- **Acorn:** modern grammar, source-goal and early-error checking; needs a vendored trusted bundle plus a Python engine wheel. Embedded engine execution is only for the parser.

08 / MAINTENANCE AND ADOPTION

Check maintenance of both the parser and its Python package

Component	Stars snapshot	Monthly downloads	Release/maintenance observation
Tree-sitter core / Python binding	27,465 / 1,522	48,642,917 (binding)	Binding released Nov 9, 2026
JavaScript grammar	498	16,220,142	Released Sep 1, 2025; repository newer than shipped wheel
Acorn	11,476	1,297,676,177 (npm)	8.19.0 Oct 5, 2026; several releases during 2026
esprima-fork	0	129	5.0.0 Oct 13, 2026; previous release 2022

Component	Stars snapshot	Monthly downloads	Release/maintenance observation
esprima2	73	21,741	6.0.0 Apr 22, 2026
Original Python Esprima	265	1,514,617	Last package release Aug 2018
Oxc core / Python wrapper	23,454 / 0	49 (wrapper)	Wrapper Nov 2025; core cadence does not update wrapper
quickjs-ng Python wrapper	10	553,556	Nov 5, 2026; recurring recent releases
mini-racer Python wrapper	219	2,293,645	Feb 1, 2026; roughly nine months without a package release

These figures were retrieved on November 15, 2026. Repository and registry APIs supply the version and project data. PyPI Stats is a third-party download service. The npm count covers October 16 through November 14. PyPI Stats labels its count `last_month` without specifying identical window boundaries. Downloads include automation, installations pulled in by other packages and repeat installations. They do not count unique users, deployments or adoption of the latest release. Stars indicate attention; they do not establish validation accuracy or security.

QuickJS-NG's engine has 4,152 stars; its Python wrapper has 10. Treat them as separate projects. V8's upstream reputation does not establish the freshness of mini-racer's bundled engine. Available wheel filenames were checked, but benchmarks ran only on Linux x86-64. Windows Application Control blocked some native imports on the development laptop. The Linux CI jobs did not have that restriction.

All exact endpoints, timestamps, release samples, wheel coverage, licenses and raw download responses: maintenance evidence report. No vulnerability audit, maintainer response-time study or supply-chain certification is claimed.

How test results and maintenance affect the recommendation

1. Select the correct parsing context and distinguish syntax errors from failed checks.
2. Prioritize avoiding incorrect rejection of valid code.
3. Compare the invalid cases detected, then weigh detection differences against deployment limits and maintenance evidence.
4. State any accepted detection gap explicitly, as this recommendation does for Acorn's static-block case.

Acorn is the overall recommendation because it combines useful error detection with established parser adoption. The costs are the native engine dependency, startup overhead, maintenance of the Python wrapper and one known missed error. Use the modern fork when an implementation entirely in Python is required. Use Tree-sitter when an advisory check for malformed structure is sufficient.

Install mini-racer and call Acorn from Python

CODE EXAMPLE

```
python -m pip install mini-racer==0.14.1
```

Download Acorn 8.19.0's prebuilt JavaScript file and store it as `acorn.js` beside the Python module. Retain its MIT license. The helper verifies the file's expected SHA-256 hash before loading it. The application container needs no npm or Node installation. The tested API requires Python ≥3.10 and a compatible prebuilt mini-racer package.

CODE EXAMPLE

```
"""Node-free strict syntax check: only trusted Acorn code executes."""
import hashlib
from pathlib import Path

from py_mini_racer import MiniRacer

ACORN_SHA256 = "b95392d79457d0357555c5ed3e56302d074b92dff119bcf3b44179500b10e9df"
WRAPPER = """
function checkJavaScript(source, mode) {
  try {
    acorn.parse(source, {ecmaVersion: 2025, sourceType: mode});
    return {status: "valid"};
  } catch (error) {
    if (error instanceof SyntaxError &&
        !error.message.startsWith("Not enough stack space")) {
      return {status: "invalid", message: error.message,
              line: error.loc.line, utf16Column: error.loc.column};
    }
    return {status: "inconclusive", reason: "parser_resource_or_internal_error"};
  }
}
"""


class JavaScriptValidator:
    """Use one context per worker; close it when that worker is retired."""
    def __init__(self, acorn_path: str):
        bundle = Path(acorn_path).read_bytes()
        if hashlib.sha256(bundle).hexdigest() != ACORN_SHA256:
            raise ValueError("Unexpected Acorn bundle: expected pinned 8.19.0")
        self.context = MiniRacer()
        try:
            self.context.eval(bundle.decode("utf-8"))
            self.context.eval(WRAPPER)
        except Exception:
            self.context.close()
```

```
            raise

    def check(self, source: str, mode: str = "script") -> dict:
        if mode not in ("script", "module"):
            raise ValueError("Use the actual script or module parse goal")
        if len(source.encode("utf-8")) > 256_000:
            return {"status": "inconclusive", "reason": "input_size_limit"}
        try:
            # Pass source as an argument; never interpolate it into eval().
            return self.context.call("checkJavaScript", source, mode, timeout=2000)
        except Exception as exc:
            return {"status": "inconclusive", "reason": type(exc).__name__}

    def close(self):
        self.context.close()
```

```
 CODE EXAMPLE
validator = JavaScriptValidator("acorn.js")
try:
    result = validator.check(extracted_script, mode="module")  # actual HTML type
finally:
    validator.close()
```

Keep one engine context per worker and reuse it across requests. The example's 256,000-byte input limit and two-second engine timeout are application policies, not JavaScript rules. The benchmark also used an external worker deadline, which this compact helper does not provide. For a service, add process-level time and memory limits. Infrastructure exceptions return `inconclusive`. Error lines start at 1, and columns start at 0 and count UTF-16 code units. Convert those units before combining them with Python character indexes or UTF-8 byte offsets.

The integration tests cover modern syntax, duplicate declarations, Script/Module differences for `await`, size limits, deep input, JSX rejection and the documented missed error. A test parses an assignment and confirms that the variable was not created, checking that submitted statements did not run. Acorn can construct RegExp objects from the source while building its AST, but does not run matches. Parsing still consumes CPU and memory in the native engine.

Extract the JavaScript that the browser will receive

Preserve script text when extracting it with BeautifulSoup. Classify each element as a classic script, module or data block, then parse each JavaScript block separately. A script with `src` remains unchecked unless its external source is supplied. Validate JSON and JSON-LD with a strict JSON parser. Import maps and speculation rules also need checks for their required structure.

A literal `</script>` can end an HTML script element even inside a JavaScript string. Inside a comment, it can leave truncated source that still parses successfully. HTML tokenizer states, `noscript`, templates, namespaces and event handlers also affect extraction. Do not decode HTML entities in script bodies. Event-handler attributes are a different context, where a statement such as `return false` can be valid.

Expanded boundary checks and exact observed outcomes: HTML/JSON regression results. There are 76 passing assertions and 4 context observations, measured separately on Windows/Python 3.14.8 with BeautifulSoup 4.15.0 and html5lib 1.1. Their scope and limitations are recorded; no full browser rendering test is claimed.

10 / REPORT AND TEST DOWNLOADS

Download the report, test samples and raw results

Download the PDF for the recommendation and integration examples. Use the JSON files to inspect exact samples and results, or the technical appendix to reproduce the experiment.

Rerun the tests when upgrading a dependency

Pin both the parser file and the Python engine package. Keep the known Acorn failure in your regression tests. Rerun the published samples and representative generated output whenever either dependency changes.

- Parallel Linux workflow and successful runs
- 76 passing HTML/JSON assertions and four context observations
- Test262 license · Pinned Acorn source and license

> **Before deployment:** run the downloaded samples in your own environment, then add examples from your generator. Keep syntax errors separate from failed validation. Use browser tests to check DOM behavior, imports and application logic that a parser cannot evaluate.

This finite, versioned study does not establish full conformance, audit security or estimate production error rates. No submitted JavaScript was executed in the benchmark.