CHECKLIST

JavaScript Checklist for Production Readiness

A practical JavaScript checklist for technical teams to validate code quality, security, testing, deployment, and runtime readiness before production use.

Programming - July 03, 2026

Purpose

JavaScript issues often surface late: a dependency update changes runtime behavior, a browser-only API reaches a Node service, a build step strips needed code, or an insecure pattern slips through review. In production, those mistakes turn into outages, security exposure, and hard-to-diagnose regressions.

This JavaScript checklist helps you verify whether a codebase is ready for production use. It is designed for technical professionals who need a repeatable review method for applications, services, libraries, and tooling written in JavaScript. After using it, you should be able to decide whether the implementation is safe to release, apply a practical validation workflow, and know exactly what to verify before deployment.

How to use this checklist

Use this checklist during pull request review, release readiness review, or pre-deployment validation. Work through the phases in order. For each check, capture evidence, assign an owner, and record the review cadence so the checklist stays operational rather than theoretical.

Treat any failed item as a release blocker unless you can document a compensating control and a temporary risk acceptance. For components with stricter release controls, you can pair this checklist with a broader operational readiness review such as the NET Checklist: Operational Readiness Review for Production Use when JavaScript is part of a larger service delivery pipeline.

Readiness scoring

Score each phase as follows:

- 2 points ? All checks in the phase pass and evidence is complete.
- 1 point ? Most checks pass, but one non-critical item needs follow-up.
- 0 points ? One or more checks fail, or evidence is missing.

Interpretation:

- 10?12 points: Ready for production with normal change control.
- 7?9 points: Conditionally ready; approve only with documented follow-up actions and owner dates.
- 0?6 points: Not ready; do not release until failures are resolved.

1) Scope and runtime assumptions

A JavaScript review starts with the environment. Many failures come from unclear runtime assumptions: browser versus Node.js, ESM versus CommonJS, transpiled versus native syntax, and platform-specific APIs that only exist in certain versions or settings.

Purpose: Confirm the code targets the intended runtime and that the execution model matches production.

- ? Confirm the code path is explicitly tied to the correct runtime: browser, Node.js, edge runtime, serverless function, or test-only utility.
- ? Review module format assumptions and verify whether the package expects ESM, CommonJS, or dual packaging.

- ? Validate that all runtime-specific APIs are supported in the deployed environment and version range.
- ? Document any required flags, polyfills, transpilation targets, or language features that affect behavior.
- ? Assign a named owner for runtime compatibility decisions and version upgrades.
- ? Review the change cadence for runtime versions, browser support policy, and dependency update windows.

Evidence to capture: package metadata, runtime matrix, supported platform list, build configuration, feature flag list, and compatibility notes.

Acceptance criteria: the code runs in the declared target environment without hidden assumptions, and compatibility requirements are documented before merge.

Owner: application engineer or platform engineer.

Review cadence: every release and after any runtime upgrade.

Common mistakes:

- Mixing browser-only globals into server code.
- Relying on syntax unsupported by the actual production target.
- Assuming local development behavior matches production runtime flags.

2) Source quality and maintainability

Production JavaScript should be easy to reason about under pressure. If the control flow is unclear, future changes become riskier and incident response slows down. This phase checks whether the implementation is readable, testable, and internally consistent.

Purpose: Confirm that the source code is maintainable and has clear logic boundaries.

- ? Review function size and complexity and confirm the code is split into testable units.
- ? Validate variable names, parameter names, and return values for clarity and consistency.
- ? Confirm error handling paths are explicit and do not suppress failures without justification.
- ? Document any intentional tradeoffs such as compatibility shims, temporary duplication, or guarded exceptions.

- ? Assign a reviewer to verify that dead code, debugging statements, and commented-out logic are removed.
- ? Test the primary execution paths and failure paths with realistic input values.

Evidence to capture: static analysis output, code review comments, test results, and a short design note for any non-obvious implementation choice.

Acceptance criteria: the code is understandable without tribal knowledge, failure behavior is explicit, and no unresolved clean-up items remain.

Owner: code author with peer reviewer approval.

Review cadence: each pull request and before release candidates.

Common mistakes:

- Leaving broad catch blocks that hide root causes.
- Using nested callbacks or deeply chained promises where a simpler structure would be safer.
- Allowing implicit coercion to create hard-to-read behavior.

3) Dependency and supply-chain control

JavaScript projects often depend on large trees of third-party packages. That makes dependency governance a core release concern, not an afterthought. You need to verify what is installed, why it is installed, and whether it is still acceptable to ship.

Purpose: Confirm that dependencies are intentional, pinned appropriately, and reviewed for risk.

- ? Review direct and transitive dependencies and confirm each package has a clear business or technical need.
- ? Validate lockfile consistency and verify the committed lockfile matches the intended dependency graph.
- ? Confirm version ranges are deliberate and do not permit uncontrolled upgrades in production builds.
- ? Document any known package exceptions, forks, or vendored code that affect supportability.
- ? Assign ownership for dependency review, including security alerts and patch cadence.

- ? Test install and build behavior from a clean environment to confirm reproducible results.

Evidence to capture: lockfile, dependency inventory, package audit results, clean install logs, and approval notes for exceptions.

Acceptance criteria: the dependency set is reproducible, necessary, and reviewed; no unapproved packages are introduced.

Owner: application engineer plus security or supply-chain reviewer.

Review cadence: every dependency change and on a fixed periodic review cycle.

Common mistakes:

- Accepting broad semver ranges without understanding upgrade impact.
- Shipping packages that are no longer maintained without mitigation.
- Reviewing only direct dependencies and ignoring transitive risk.

4) Security controls and input handling

JavaScript commonly processes external input from forms, APIs, message queues, cookies, headers, or local storage. If you do not verify input boundaries, the code may expose injection, script execution, token leakage, or unsafe deserialization patterns.

Purpose: Confirm the code handles untrusted data safely and enforces least privilege.

- ? Review all input sources and confirm validation happens before the data reaches business logic.
- ? Validate that output encoding or escaping is applied in the correct context, especially for HTML, attributes, URLs, and logs.
- ? Confirm secrets, tokens, and session values are never written to client-visible storage unless there is a documented design requirement.
- ? Document any use of dynamic code execution, template injection risk, or runtime evaluation and prove why it is necessary.
- ? Assign a security reviewer to confirm authorization checks are server-side and not only enforced in the client.
- ? Test negative cases for malformed input, oversized payloads, missing fields, and tampered requests.

Evidence to capture: validation rules, security review notes, negative test results, and examples of encoded output or rejected payloads.

Acceptance criteria: untrusted input is validated and constrained, sensitive data is protected, and risky execution paths are justified and reviewed.

Owner: security engineer or application engineer with security sign-off.

Review cadence: every security-sensitive change and after any input surface expansion.

Common mistakes:

- Trusting client-side validation as a security boundary.
- Logging secrets or session identifiers during debug sessions.
- Using dynamic code paths when static alternatives exist.

5) Error handling, logging, and observability

A production JavaScript system must fail loudly enough for operators to detect issues, but safely enough to avoid leaking sensitive context. This phase checks that you can diagnose incidents without exposing internal data to users or logs.

Purpose: Confirm errors are actionable, logs are safe, and operational signals exist.

- ? Review error messages and confirm they are precise for operators but do not expose secrets or internal stack details to end users.
- ? Validate that logs include enough context for troubleshooting, such as request IDs, component names, or correlation values.
- ? Confirm log levels are appropriate and that debug logging is disabled or controlled in production.
- ? Document alert conditions for recurring errors, failed requests, and degraded dependencies.
- ? Assign ownership for log retention, access control, and on-call response.
- ? Test failure scenarios to confirm errors are captured, reported, and observable in the expected location.

Evidence to capture: sample logs, alert definitions, incident runbook references, and output from controlled failure tests.

Acceptance criteria: operators can detect and diagnose failures, users do not see sensitive internals, and monitoring data is actionable.

Owner: operations engineer or SRE with application owner support.

Review cadence: every release that changes error handling, logging, or alerting.

Common mistakes:

- Returning raw stack traces to clients.
- Logging too little context to identify the failing request.
- Generating noisy alerts without an owner or runbook.

6) Testing and validation coverage

Tests should prove that important behavior works, not just that the code executes. JavaScript codebases often pass unit tests while failing on integration boundaries, asynchronous timing, browser compatibility, or serialization issues. If the project involves data workflows, align this phase with a structured validation approach such as the Learning Checklist for AI and Machine Learning Projects when JavaScript is part of an AI-adjacent delivery chain that depends on evidence, ownership, and acceptance criteria.

Purpose: Confirm the code is validated at the right levels and against realistic failure modes.

- ? Review unit tests and confirm they cover core logic, edge cases, and error branches.
- ? Validate integration tests for boundaries such as APIs, databases, queues, authentication, and external services.
- ? Confirm asynchronous behavior is tested for timing, retries, cancellation, and race conditions.
- ? Document any browser, device, or environment coverage requirements that affect release confidence.
- ? Assign a person responsible for maintaining failing test triage and test data quality.
- ? Test the build in the same mode used for production release, including minification or bundling if applicable.

Evidence to capture: test reports, coverage summary, integration test results, and environment matrix results.

Acceptance criteria: essential behavior is covered at the right depth, and production build output is validated rather than assumed.

Owner: engineer responsible for the feature or module.

Review cadence: every change set and before release cutover.

Common mistakes:

- Treating high coverage as proof of business correctness.
- Ignoring asynchronous race conditions in event-driven code.
- Testing only source files and not the bundled output.

7) Build, packaging, and deployment readiness

JavaScript production failures often come from build-time differences: environment variables, tree-shaking, asset paths, bundler behavior, or module resolution. This phase checks that the artifact you ship is the artifact you validated.

Purpose: Confirm the build and deployment process produce a deterministic, deployable artifact.

- ? Review build scripts and confirm the production build uses the intended environment and configuration.
- ? Validate that required environment variables are documented, present, and safely injected at deploy time.
- ? Confirm the packaged artifact contains the expected files and excludes development-only content.
- ? Document rollback prerequisites, version identifiers, and deployment order for dependent services.
- ? Assign release ownership and approval authority for the deployment window.
- ? Test installation or deployment from a clean pipeline run to verify repeatability.

Evidence to capture: build logs, artifact manifest, deployment manifest, environment variable inventory, and rollback plan.

Acceptance criteria: production artifacts are reproducible, deployable, and traceable to a specific source revision.

Owner: build/release engineer or platform engineer.

Review cadence: every release and after build pipeline changes.

Common mistakes:

- Building with one set of environment variables and deploying with another.
- Shipping unreviewed generated assets.
- Assuming rollback is possible without verifying artifact retention.

8) Runtime performance and resource safety

JavaScript applications can create production issues through memory growth, long event-loop blocking, excessive client-side work, or unbounded retries. The goal here is not micro-optimization; it is verifying that the code behaves predictably under expected load.

Purpose: Confirm the implementation uses acceptable CPU, memory, and network resources.

- ? Review any synchronous loops, large object copies, or expensive transformations that could block runtime execution.
- ? Validate request handling paths for unbounded retries, recursion, or accumulation of state across calls.
- ? Confirm browser-side code does not create unnecessary re-renders, layout thrash, or large payload downloads.
- ? Document expected throughput, latency sensitivity, and resource limits for the feature or service.
- ? Assign a reviewer to confirm performance-sensitive changes are rechecked after code modifications.
- ? Test with representative load or data volume to observe memory use, response time, and failure behavior.

Evidence to capture: profiling output, load test observations, memory snapshots, and any measured limits or thresholds.

Acceptance criteria: the code stays within expected resource bounds and has no obvious unbounded behavior under normal use.

Owner: feature owner with performance or platform support.

Review cadence: before release and after any change affecting hot paths.

Common mistakes:

- Optimizing for small test cases while ignoring production data size.
- Introducing repeated parsing or serialization in hot paths.
- Leaving retry loops without backoff or upper bounds.

9) Operational ownership and maintenance

A JavaScript release is only production-ready when someone can support it after deployment. Ownership includes patching dependencies, responding to incidents, and keeping the code aligned with platform changes over time.

Purpose: Confirm the code has a support model, update plan, and review schedule.

- ? Assign an operational owner for the service, library, or application module.
- ? Review patching responsibility for runtime, dependencies, and build tooling.
- ? Document support boundaries, including what is in scope for the owning team and what requires escalation.
- ? Validate that key runbooks exist for deployment failure, runtime error, and dependency rollback.
- ? Confirm review cadence for security updates, runtime version changes, and dependency drift.
- ? Test at least one recovery path, such as rollback or config reversion, before production approval.

Evidence to capture: ownership record, runbooks, patch schedule, escalation contact list, and recovery test results.

Acceptance criteria: the component has an accountable owner, a maintenance plan, and a proven recovery path.

Owner: team lead or service owner.

Review cadence: monthly or aligned to the organization?s change schedule.

Common mistakes:

- Releasing code with no named operational owner.

- Ignoring dependency drift until an incident forces a rushed upgrade.
- Keeping runbooks outdated after deployment changes.

Pass/fail decision rules

Use simple decision rules so the review produces a clear outcome.

- Pass: all critical checks pass, evidence is attached, and no unresolved security, build, or runtime issues remain.
- Conditional pass: only non-critical items remain, each has an owner, and follow-up dates are recorded before release.
- Fail: any issue can cause data exposure, service outage, unsupported runtime behavior, or unreproducible deployment.

A failed review should not be negotiated away without a documented risk decision. If you need a more formal release gate, combine this checklist with release evidence from your pipeline and with a project-specific implementation checklist, such as the Learning Implementation Roadmap Checklist when the JavaScript work is part of a broader implementation plan.

Final review packet

Before you approve production use, make sure the review packet contains the minimum evidence needed to defend the decision later:

- ? Confirm the checklist score and pass/fail outcome are recorded.
- ? Review the list of open exceptions and verify each has an owner and due date.
- ? Validate that artifacts, build logs, test results, and rollback information are attached or linked.
- ? Document any residual risk and the person who approved it.
- ? Assign follow-up actions for unresolved items and schedule the re-review date.

If that packet is complete and the failed items are resolved or formally accepted, the JavaScript change is ready to move with controlled risk. If not, the safest decision is to delay release until the checklist is complete and the evidence is strong enough to support production use.

Use this guidance together with python checklist and ASP checklist to connect the workflow with related operational context already available on the site.

Related guides in this cluster

- How to get started with JavaScript
- JavaScript Reporting Template FAQ: Build a Reliable Output Pattern

Related guides in this cluster

- JavaScript FAQ: How to Handle Async Errors with Promises

Related guides in this cluster

- JavaScript Prototype Pollution: Detect and Prevent Exploits
- JavaScript Secure Coding: Prevent Prototype Pollution Attacks

Related guides in this cluster

- Secure JavaScript Input Validation with Regular Expressions
- JavaScript Prototype Pollution Prevention and Detection Techniques

Related guides in this cluster

- JavaScript Promise Handling Patterns for Reliable Async Code

Related guides in this cluster

- Secure JavaScript Error Handling with Async Try-Catch Patterns

Related guides in this cluster

- JavaScript Regex Validation for Secure Input Handling

Related guides in this cluster

- How to Validate JavaScript Input with Regular Expressions

Related guides in this cluster

- Secure JavaScript Object Validation with Zod Schemas

Related guides in this cluster

- JavaScript Async/Await Error Handling for Secure APIs

Related guides in this cluster

- Securing JavaScript APIs with JWT Authentication and Role Checks

Related guides in this cluster

- JavaScript Secure JSON Parsing to Prevent Prototype Pollution