CHECKLIST

C# Async/Await Checklist for Production-Ready Services

Use this practical C# async/await checklist to verify deadlock safety, exception handling, cancellation, timeouts, testing, and production readiness before deployment.

Programming - August 03, 2026

Purpose

The practical problem with C# async/await is not writing asynchronous code once; it is proving that the code is safe to run under load, does not deadlock, handles failures predictably, and behaves correctly when cancellation or timeouts occur. In real services, small mistakes such as blocking on a Task, swallowing exceptions, or omitting cancellation tokens can turn a routine implementation into a reliability issue.

Use this checklist to decide whether an async/await implementation is fit for production. By the end, you should be able to confirm whether the approach applies, validate the implementation path, collect evidence for review, and verify what must be true before release.

How to use this checklist

Work through the phases in order. For each item, capture evidence, assign an owner, and record whether the check passed or failed. A check is only complete when you can point to concrete proof such as code review notes, test output, logging, or a runtime validation result.

If you are already troubleshooting a service that hangs or fails intermittently, this checklist also helps you isolate the most common async/await failure patterns. In particular, if you suspect a sync-over-async problem, compare your findings with C# Async/Await Deadlocks: Detect and Prevent Them. For error-path behavior, use the exception-handling patterns in C# Async/Await Exception Handling Patterns for Reliable Services.

Scope and readiness decision

Before reviewing implementation details, confirm that async/await is actually the right model for the workload. This phase prevents teams from treating async as a default instead of a design choice.

Checklist items

- ? Confirm the code path performs I/O-bound or latency-sensitive work that benefits from asynchronous execution.
- ? Review whether the caller chain can remain asynchronous end to end without forced blocking.
- ? Validate that library code does not assume a UI thread, request thread, or any specific synchronization context.
- ? Document where synchronous execution is still intentional and why it is safe.
- ? Assign an owner for the async contract at each boundary: caller, service, library, and integration layer.

Evidence to capture

- Architecture notes that identify the async boundary.
- Code review comments showing that blocking calls were examined.
- A call-flow diagram or dependency trace for the main request path.

Acceptance criteria

- No required path blocks on Task.Result, Task.Wait(), or equivalent synchronous waits.
- Any synchronous segment is explicitly justified and isolated.
- The async contract is clear across layers.

Owner

Application architect or senior developer responsible for the feature.

Review cadence

At design review, then again during implementation review for any refactor that changes call flow.

Common mistakes

- Treating async as a cosmetic keyword instead of an architectural decision.
- Introducing async in one layer while leaving the caller synchronous.
- Assuming a pattern that works in one hosting model is safe in another.

Implementation safety checks

This phase verifies that the code is written to avoid deadlocks, hidden blocking, and context capture problems. It is the core of the checklist because most production issues appear here.

Checklist items

- ? Confirm every awaited operation that can suspend the current flow is marked with await rather than being manually blocked.
- ? Review all calls for sync-over-async patterns such as .Result, .Wait(), or blocking wrappers around asynchronous APIs.
- ? Validate that continuations do not rely on a captured context unless that behavior is intentionally required.
- ? Test whether any code path can resume on a context that is already occupied or unavailable.
- ? Review helper methods for accidental task re-wrapping or nested task usage that obscures failures.
- ? Document any place where context flow is intentionally preserved and why.
- ? Validate library-facing methods with the correct async signature so callers can propagate cancellation and failure.

Evidence to capture

- Static analysis findings or code review annotations.
- A list of methods audited for sync blocking.
- Notes on any context-sensitive code paths.

Acceptance criteria

- No synchronous waiting remains in the request or service path unless explicitly approved.
- Any context dependency is intentional, documented, and testable.
- Async methods expose predictable return types and do not hide task completion state.

Owner

Primary developer, with review by a peer familiar with async runtime behavior.

Review cadence

Per pull request, and again after any change that adds cross-thread or cross-layer execution.

Common mistakes

- Calling async APIs from synchronous constructors or property getters.
- Using async void outside event handlers.
- Assuming that ConfigureAwait is a universal fix without first understanding the execution context.

Exception, cancellation, and timeout behavior

Async code is not production-ready unless failure paths are defined as clearly as success paths. This phase checks what happens when operations fail, time out, or are cancelled.

Checklist items

- ? Confirm every external I/O call has a defined exception-handling path.
- ? Review whether exceptions are logged, translated, or rethrown consistently at service boundaries.

- ? Validate that cancellation tokens are passed through all supported layers and are honored by downstream calls.
- ? Test that cancellation is treated as a controlled outcome rather than an error condition in logs where appropriate.
- ? Confirm that timeout behavior is explicit and distinguishable from cancellation and general faults.
- ? Document which exceptions are expected, which are retryable, and which require immediate failure.
- ? Verify that exception handling does not lose stack traces or suppress root causes.

Evidence to capture

- Unit or integration test output for fault, cancellation, and timeout cases.
- Logging samples that show the intended error classification.
- Service contract notes showing retry and fail-fast decisions.

Acceptance criteria

- The service reports failures with enough detail for operators to act.
- Cancellation stops work cleanly and does not leak resources.
- Timeouts are consistent and measurable.

Owner

Service owner or reliability engineer.

Review cadence

During implementation, after any retry policy change, and before production release.

Common mistakes

- Catching exceptions broadly and returning a success-shaped response.
- Conflating cancellation with failure in alerts or dashboards.
- Leaving timeout behavior to defaults that vary by caller or transport.

Concurrency and resource management checks

Async code often fails under pressure because it waits on shared resources, opens too many concurrent operations, or leaks handles when a task exits early. This phase checks operational robustness rather than syntax.

Checklist items

- ? Validate that concurrent operations are bounded where the dependency or system capacity requires it.
- ? Review whether shared resources such as sockets, streams, database connections, or file handles are disposed deterministically.
- ? Confirm that long-running tasks do not monopolize thread pool threads through hidden blocking work.
- ? Test whether backpressure or throttling is needed when fan-out increases.
- ? Document any SemaphoreSlim, queue, or rate-limit pattern used to control concurrency.
- ? Verify that disposable async resources use the correct asynchronous cleanup path when available.

Evidence to capture

- Load test observations, even if limited to a small-scale stress run.

- Resource usage samples showing no runaway growth.
- Code review notes for disposal and throttling logic.

Acceptance criteria

- Resource use remains stable under expected concurrency.
- The code does not leak handles or leave pending work unmanaged.
- Any concurrency limits are intentional and documented.

Owner

Developers responsible for the affected subsystem, with operations input for capacity-sensitive paths.

Review cadence

During performance review, after dependency changes, and after incident follow-up.

Common mistakes

- Spawning unbounded tasks for each request.
- Disposing resources only on the success path.
- Using async code while still performing heavy CPU work on request threads.

Test and validation checks

A checklist is incomplete unless it includes proof from tests. Validate both happy paths and failure modes so the implementation is not only correct in theory.

Checklist items

- ? Confirm unit tests cover success, exception, cancellation, and timeout behavior for each critical async method.
- ? Review integration tests that exercise the full async path through the real hosting and transport stack.
- ? Validate that tests fail when synchronous blocking is reintroduced into the path.
- ? Test that deadlock-prone or context-sensitive code paths behave correctly in the intended runtime.
- ? Document the minimum set of scenarios that must pass before merge.
- ? Verify that test names and assertions clearly describe the expected operational outcome.

Evidence to capture

- Test results with pass/fail status.
- Reproduction steps for any defect found during testing.
- Coverage notes for failure paths and boundary conditions.

Acceptance criteria

- Critical async paths are exercised under realistic execution conditions.
- Failure-path tests confirm that errors are visible and controlled.
- The test suite would detect a regression that reintroduces blocking or hides exceptions.

Owner

QA engineer, developer, or both depending on the team model.

Review cadence

Before merge, after any async refactor, and after dependency upgrades that change runtime behavior.

Common mistakes

- Testing only the successful request path.
- Using mocks so heavily that the async behavior of the real stack is never exercised.
- Treating a passing unit test as proof that production concurrency is safe.

Production readiness checks

This phase turns implementation confidence into release confidence. It confirms that the service is observable, supportable, and safe to operate once deployed.

Checklist items

- ? Confirm logging includes enough context to trace failures without overwhelming normal operation.
- ? Review metrics for latency, error rate, cancellation count, and timeout count on async paths.
- ? Validate that alerts distinguish persistent faults from expected transient cancellations.
- ? Document rollback criteria if the async change increases latency, failure rate, or resource consumption.
- ? Assign a runbook owner who can investigate hangs, retries, and unexplained task failures.

- ? Verify that production settings such as framework version, hosting model, and dependency versions are recorded because async behavior can vary by runtime and configuration.

Evidence to capture

- Monitoring dashboard references or metric names.
- Runbook links or operational notes.
- Release approval record with rollback thresholds.

Acceptance criteria

- Operators can tell whether the service is waiting, failing, cancelling, or timing out.
- A rollback path exists if async behavior degrades service health.
- Runtime and configuration dependencies are documented before release.

Owner

Operations lead, service owner, and release manager.

Review cadence

Every release, then during post-deployment review if new async behavior changes runtime characteristics.

Common mistakes

- Shipping async changes without new observability for timeouts and cancellations.
- Assuming all failures are visible in application logs.

- Not recording version-sensitive behavior that may differ across environments.

Readiness scoring

Use a simple maturity score to decide whether the implementation is ready, needs remediation, or is only suitable for limited use. Score each phase from 0 to 2.

- 0 = Not verified
- 1 = Partially verified
- 2 = Fully verified

Scoring method

- Scope and readiness decision: 2 points max
- Implementation safety checks: 2 points max
- Exception, cancellation, and timeout behavior: 2 points max
- Concurrency and resource management checks: 2 points max
- Test and validation checks: 2 points max
- Production readiness checks: 2 points max

Interpretation

- 10 to 12 points: Ready for production review, assuming no critical failed checks remain.
- 7 to 9 points: Use with caution; complete the missing verification items before release.
- 0 to 6 points: Not ready; do not deploy until the failed phases are remediated.

Pass/fail rule

A single failed critical check in deadlock safety, exception handling, cancellation, or production observability is a fail regardless of the total score.

Final review criteria

Use this short gate before sign-off:

- ? Confirm no critical async path blocks synchronously.
- ? Validate that exceptions are observable and classified correctly.
- ? Review cancellation and timeout behavior for correctness and consistency.
- ? Test resource cleanup under success, failure, and cancellation.
- ? Document the owner, rollback path, and runtime assumptions.

If any item here cannot be proven with evidence, the implementation is not production-ready yet. The safest sign-off is one where the checklist, the tests, and the operational notes all tell the same story.

Use this guidance together with secure JSON parsing and prototype pollution to connect the workflow with related operational context already available on the site.

> Part of the Programming: C# Insights content cluster.