VectraOps

Knowledge Library

CHECKLIST

C# Async/Await Deadlock Prevention Checklist

A practical checklist for identifying, validating, and preventing async/await deadlocks in C# before they reach production. Use it to review blocking patterns, context capture, evidence, ownership, and release readiness.

Programming - August 03, 2026

Purpose

Async/await deadlocks are operational failures, not just code-quality issues. They usually appear when synchronous blocking meets asynchronous code, when a captured context cannot resume, or when test environments hide a production-only wait pattern. This checklist helps you verify whether a C# code path is safe to ship, where it can deadlock, and what evidence you need before production use.

Use this as a review worksheet during code review, incident analysis, or pre-release validation. By the end, you should be able to decide whether an async implementation is safe, apply a practical validation workflow, and confirm the specific checks needed to reduce deadlock risk.

How to use this checklist

Review one code path, service, or component at a time. For each phase, mark every item that can be verified, capture the evidence listed, and compare the result to the acceptance criteria before moving on. If a check fails, treat it as a release blocker until the owner documents a safe fix, test evidence, and rollback or mitigation plan.

Use the scoring section at the end to classify readiness. A partial pass is useful only if the failures are understood, bounded, and accepted by the component owner.

Phase 1: Define the async execution path

Purpose: Confirm the code path actually depends on asynchronous execution and identify where deadlock risk can enter the flow.

- ? Confirm the method boundary and the call chain that enter async code, including any sync wrapper, event handler, controller action, worker loop, or startup path.
- ? Review every call site that invokes the async method from synchronous code and identify any blocking wait such as .Result, .Wait(), GetAwaiter().GetResult(), or equivalent wrapper logic.
- ? Validate whether the path can run under a captured synchronization context such as a UI thread, request context, legacy host, or test framework context.
- ? Document whether the code path is latency-sensitive, request-scoped, startup-scoped, or background-only, because the acceptable mitigation differs by execution model.
- ? Assign a single owner for the path and a reviewer who can confirm context behavior, thread usage, and release constraints.

Evidence to capture: call graph, representative stack trace, list of sync-to-async transitions, host or runtime type, and the identified owner.

Acceptance criteria: no unreviewed synchronous blocking in an async call chain; execution context is known; owner and reviewer are assigned.

Review cadence: at design review, again before merge, and after any runtime or hosting model change.

Common mistakes in this phase

- Blocking on task completion in a controller, handler, or UI callback.
- Assuming a background test passes means the same code is safe in a request context.
- Treating helper methods as harmless because they are small, even though they block the chain.

Phase 2: Inspect blocking and context capture risks

Purpose: Find the exact mechanisms that can create deadlock or indefinite waiting.

- ? Confirm that no code in the path blocks on a task when an async alternative is available.
- ? Review every await in the path and identify whether context capture is intentional or whether the code should avoid resuming on the original context.
- ? Validate library code separately from application code, because library helpers should usually avoid assuming a specific context.
- ? Document any use of ConfigureAwait(false) and the reason it is present or omitted, especially in lower-level libraries. For guidance on safe async patterns in network-heavy paths, see C# Async Await Tutorial for Secure Network Programming.
- ? Test whether cancellation and timeout paths return control promptly instead of waiting indefinitely on a contended context.

Evidence to capture: code snippet inventory, await list, synchronization-context assumptions, timeout settings, and cancellation behavior.

Acceptance criteria: blocking calls are eliminated or explicitly justified; context capture is understood; long waits have bounded cancellation or timeout behavior.

Review cadence: every code review, plus after refactoring async boundaries or introducing new framework dependencies.

Common mistakes in this phase

- Using .Result in a method that sometimes runs on a captured context.
- Adding ConfigureAwait(false) inconsistently without understanding downstream continuation requirements.
- Relying on timeouts alone while leaving the underlying blocking pattern intact.

Phase 3: Validate async correctness under realistic conditions

Purpose: Prove the code behaves correctly under contention, scheduling delays, and failure paths.

- ? Test the code path under a host that matches production threading and synchronization behavior, not just a minimal unit-test environment.
- ? Review whether the test suite includes a case where the caller blocks while the callee awaits an incomplete task.
- ? Validate that the code still completes when thread pool pressure, network delay, or slow I/O causes continuations to be scheduled later than expected.
- ? Document any observed wait chains, blocked threads, or thread starvation symptoms during test execution.
- ? Confirm that exception propagation preserves failure visibility rather than masking a deadlock as a timeout, empty result, or partial success.

Evidence to capture: test output, timing logs, thread dumps or debugger traces, failure-mode results, and a reproducible test case.

Acceptance criteria: the code completes successfully under delayed continuation conditions; failures surface clearly; no hidden blocking is observed in the verified path.

Review cadence: before release, after dependency upgrades, and after changes that affect host threading or request processing.

Common mistakes in this phase

- Only testing the happy path where tasks complete immediately.
- Using mocks that return completed tasks and therefore hide blocking behavior.
- Treating a timeout as a fix instead of a symptom.

Phase 4: Review interface and boundary design

Purpose: Prevent deadlocks by designing call boundaries that do not force synchronous consumers into async work.

- ? Confirm the public API exposes an async signature where the operation is inherently asynchronous.
- ? Review whether any synchronous wrapper is necessary and, if so, whether it is isolated, documented, and covered by a specific compatibility justification.
- ? Validate that event handlers, startup hooks, and framework callbacks do not re-enter async work in a way that could deadlock the host.
- ? Document the boundary contract for callers, including whether they must await the task, provide cancellation, or avoid blocking waits.
- ? Test integration points that bridge between synchronous and asynchronous components, especially when the boundary crosses process, thread, or framework layers.

Evidence to capture: API contract notes, wrapper design rationale, integration test results, and caller expectations.

Acceptance criteria: the preferred API is async-first; any synchronous boundary is explicit and narrowly justified; integration tests cover the bridge.

Review cadence: during API design, before public release, and whenever callers report blocked execution or hangs.

Common mistakes in this phase

- Publishing both sync and async versions without a clear rule for which one callers should use.
- Keeping a sync wrapper ?for convenience? even though it introduces deadlock risk.
- Hiding async behavior behind a helper that looks synchronous to callers.

Phase 5: Check runtime safeguards and operational evidence

Purpose: Make sure the deployment environment can reveal, contain, and recover from deadlock-related failures.

- ? Confirm that logging or tracing can identify blocked requests, stalled work items, or repeated waits in the affected component.
- ? Review whether thread starvation, queue buildup, or request latency alarms exist for the service or process.
- ? Validate that cancellation, timeout, and shutdown behavior are documented and exercised during operational testing.
- ? Document the rollback or disablement path if a deadlock pattern appears after deployment.
- ? Assign on-call ownership for the component so a hang can be investigated without delay.

Evidence to capture: alert definitions, operational runbook, shutdown test results, and rollback procedure.

Acceptance criteria: blocked execution is observable; alarms and ownership are in place; recovery actions are documented and feasible.

Review cadence: before production rollout, during scheduled operational review, and after incidents or near misses.

Common mistakes in this phase

- Assuming application logs alone will reveal a deadlock.
- Missing visibility into thread pool exhaustion or request queue buildup.
- Having no operational path to disable a problematic code path.

Phase 6: Production readiness gate

Purpose: Decide whether the component is ready to ship with an acceptable deadlock risk profile.

- ? Confirm that all high-risk blocking calls have been removed, isolated, or formally accepted with a documented justification.

- ? Review unresolved findings and classify them as pass, conditional pass, or fail based on operational impact.
- ? Validate that production monitoring, ownership, and rollback procedures are active before release.
- ? Document the exact release condition that would cause this component to be re-reviewed after deployment.
- ? Test one final end-to-end path in an environment that matches the target host as closely as possible.

Evidence to capture: final review notes, decision record, production test evidence, and the list of accepted risks.

Acceptance criteria: no critical deadlock risk remains unaddressed; monitoring and recovery are ready; final test evidence matches the intended deployment model.

Review cadence: immediately before release and after any post-release signal of blocked execution.

Ready or not? Simple scoring

Use the score below to classify the component.

- 2 points for each phase that passes with no open critical findings.
- 1 point for each phase that passes only with documented mitigations.
- 0 points for each phase that fails or has unknown status.

Score interpretation

- 10?12 points: Ready for production review, assuming no unresolved critical exceptions.
- 7?9 points: Conditional readiness; ship only if the documented mitigations are acceptable to the owner and operations.
- 0?6 points: Not ready; fix the blocking pattern, rerun validation, and repeat the checklist.

Pass/fail criteria at a glance

A component passes only when these conditions are true:

- No unreviewed synchronous blocking exists in an async path.
- Context capture behavior is understood and intentional.
- Validation includes realistic scheduling or latency conditions.
- Boundary design does not force callers into unsafe sync waits.
- Operational visibility and rollback are ready before production use.

A component fails when any of the following are true:

- A blocking wait can occur on a captured context.
- Tests only cover completed-task behavior.
- Ownership, monitoring, or recovery actions are missing.
- The async boundary is hidden behind a synchronous facade without justification.

Follow-up actions after review

If the checklist passes, record the verified evidence in the release notes and keep the monitored path under observation for the first production cycle. If it fails, remove or isolate the blocking call, add a reproducible test that proves the fix, and rerun the relevant phases until the failure mode is no longer present.

If the result is conditional, document the exact limitation, the compensating control, and the deadline for full remediation. That keeps the risk visible and prevents a temporary exception from becoming a permanent deadlock hazard.

Use this guidance together with JWT authentication and authorization to connect the workflow with related operational context already available on the site.

Use this guidance together with measure C# code maturity and secure JSON parsing to connect the workflow with related operational context already available on the site.