CHECKLIST

C# Async/Await Deadlocks Checklist

A practical checklist for preventing, spotting, and diagnosing async/await deadlocks in C# applications, with production-focused guidance for UI, server, and background workloads.

C# Async/Await Deadlocks Checklist

Use this checklist to prevent, detect, and diagnose async/await deadlocks in C# applications.

1) Confirm the symptom is a progress stall

- [ ] The request, job, or UI action starts but never completes.
- [ ] CPU usage stays low while throughput drops.
- [ ] Thread pool or worker availability decreases under load.
- [ ] Higher-level timeouts appear in proxies, clients, schedulers, or load balancers.
- [ ] Logs show the operation started but no completion event follows.
- [ ] The application is unresponsive without crashing.

2) Find the sync-over-async boundary

- [ ] Search for .Result.
- [ ] Search for .Wait().
- [ ] Search for GetAwaiter().GetResult().

- [ ] Inspect wrappers that call async methods from sync methods.
- [ ] Review event handlers, middleware, controllers, and helper utilities.
- [ ] Check whether any code blocks while waiting for an async operation to finish.

3) Check for captured context issues

- [ ] Determine whether the code runs in a UI thread, request context, or custom synchronization context.
- [ ] Verify whether the awaited continuation must resume on the same context.
- [ ] Review code paths that mix UI work and background work.
- [ ] Review legacy ASP.NET or framework-specific request flows.
- [ ] Confirm whether a blocked thread is the same thread the continuation needs.

4) Look for circular waits

- [ ] Check whether Task A waits on Task B while Task B waits on Task A.
- [ ] Review pipelines, queues, callbacks, and orchestration layers.
- [ ] Inspect shared state that may create indirect task dependency cycles.
- [ ] Verify that no helper is waiting on a task it indirectly triggers.

5) Check for locks held across awaits

- [ ] Review lock, SemaphoreSlim, mutexes, and other coordination primitives.
- [ ] Ensure locks are not held longer than necessary.
- [ ] Avoid blocking inside a critical section.
- [ ] Confirm that awaited continuations do not depend on a lock still being held.

6) Inspect async method boundaries

- [ ] Prefer async all the way up the call chain when waiting is expected.
- [ ] Return Task or Task<T> instead of forcing sync return values.
- [ ] Avoid converting async operations into synchronous helpers unless absolutely necessary.
- [ ] Verify that an async method does not internally call blocking APIs.
- [ ] Ensure legacy sync code is not wrapping new async code without a clear boundary.

7) Apply the safest fixes first

- [ ] Replace synchronous waits with await where possible.
- [ ] Keep the execution flow asynchronous end to end.
- [ ] Isolate context-sensitive work from long-running or I/O-bound operations.
- [ ] If appropriate, review whether continuation context capture should be avoided in library code.
- [ ] Validate that the change removes the blocking boundary rather than masking the symptom.

8) Diagnose without making the stall worse

- [ ] Do not add retries to a deadlocked path before identifying the block.
- [ ] Do not add longer timeouts as the primary fix.
- [ ] Avoid adding more logging inside the already blocked code path.
- [ ] Capture traces or logs that show where progress stops.
- [ ] Use correlation IDs to tie the blocked operation to a specific request or transaction.

9) Reproduce in a realistic environment

- [ ] Test under production-like concurrency, not just a single local request.
- [ ] Verify behavior in the same hosting model used in production.

- [ ] Re-test after removing the sync block to confirm progress resumes.
- [ ] Compare low-load and high-load behavior for thread availability and completion rate.
- [ ] Validate that downstream calls still complete when contention increases.

10) Review common high-risk areas

- [ ] Authentication and authorization flows.
- [ ] API gateway or middleware layers.
- [ ] UI event handlers.
- [ ] Background job runners and schedulers.
- [ ] Legacy wrappers around newer async libraries.
- [ ] Audit, logging, or notification paths that run during request handling.

Quick diagnosis workflow

- [ ] Identify where progress stops: UI, request, worker, or job.
- [ ] Search for synchronous waits on tasks.
- [ ] Determine whether a synchronization context is involved.
- [ ] Look for task dependency cycles.
- [ ] Remove the blocking boundary and retest.
- [ ] Reproduce under realistic concurrency.

Operational checklist for production readiness

- [ ] Async methods are propagated through the full call chain.
- [ ] Blocking waits are absent from code paths that depend on async continuations.
- [ ] Logs include enough context to trace stalled requests.
- [ ] Timeout handling is treated as a symptom, not the root fix.

- [ ] Concurrency testing includes contention and load.
- [ ] Code reviews explicitly flag sync-over-async patterns.

Incident triage notes

- [ ] If CPU is low and progress is stalled, suspect a wait-cycle before a crash.
- [ ] If the UI freezes, inspect event handlers and context capture.
- [ ] If a server stalls under load, inspect blocked worker threads and sync wrappers.
- [ ] If one request never completes, trace the last known async boundary.
- [ ] If completion logs never appear, confirm whether the continuation can still run.

Final pass before release

- [ ] No .Result, .Wait(), or GetAwaiter().GetResult() remains in async-sensitive paths.
- [ ] No locks are held across awaited operations.
- [ ] No circular task dependencies exist in orchestration code.
- [ ] The fix was validated in the same environment where the stall occurred.
- [ ] Monitoring and logging can distinguish blocked progress from downstream failure.

Summary

C# async/await deadlocks are usually caused by synchronous blocking, context capture, or circular task dependencies. The most effective prevention strategy is to preserve asynchronous flow end to end and remove the blocking boundary that prevents continuations from running.

When diagnosing incidents, focus first on where progress stops, then verify whether a task is being forced into synchronous waiting. In production systems, that approach is faster and safer than adding retries or longer timeouts.