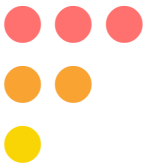
50 Technical mistakes that blow game development budgets and how to avoid them

Published: 7 Jul 2026
Last updated: 8 Jul 2026



Herman Tulleken
https://www.linkedin.com/in/hermantulleken/
herman.tulleken@gamelogic.co.za

Omar Rojo
https://www.linkedin.com/in/omarrojo/
omar.rojo@gamelogic.co.za

gamelogic

Contents

Introduction

This document catalogs the technical risks most likely to affect delivery, stability, and player experience. Each issue is paired with why it matters and proposed mitigations.

This document focuses on technical issues. Also left out are a few classes of important problems that relate to performance, memory, multiplayer, live-ops, and content management.

Crashes, freezes, and data loss

Issues that directly cause crashes, freezes, or destroyed player data — the fastest way to generate 1-star reviews and refund requests.

Crashes caused by null pointer exceptions

WHY IT'S SERIOUS

Crashes ruin the player's experience, and can also lead to data loss (which is very demotivating to players if it includes loss of progress).

Null pointer exceptions can manifest in a vast number of other ways besides crashes. A project with many null pointer exceptions is an overall stability risk.

PROPOSED MITIGATIONS

Use layers with null checks enforced at the boundaries; so that objects inside the boundary are guaranteed non-null and need no defensive checks.[1]

Don't misapply defensive programming: always flag errors.[2]

Don't use null-related operators (`?.`, `??`) and null pattern-matching on types derived from `UnityEngine.Object` or interfaces.[3]

Save data fails to persist

WHY IT'S SERIOUS

Data loss is one of the most damaging failure modes for player trust and reviews.

PROPOSED MITIGATIONS

Follow save-data best practices, including centralized API, versioning, read and write validation, continuous saving.[4] [5]

[1] Design by Contract (Bertrand Meyer, 1997)
[2] Fail Fast (Jim Shore, 2004)
[3] Custom == operator, should we keep it? (Lucas Meyer, 2014)
[4] Microsoft GDK. Game Saves walkthroughs and samples
[5] Saving the Day: Save Systems in Games (David Sirlin, 2008)

Freezes caused by poorly designed coroutines or asynchronous code

WHY IT'S SERIOUS

Freezes are highly visible, hard to reproduce, and directly drive negative reviews.

PROPOSED MITIGATIONS

Use a single strategy for asynchronous execution.

Follow best practices: never block the main thread, support cancellation, use timeouts, and handle exceptions in fire-and-forget operations.[6]

Design APIs that wrap complex asynchronous code that can be re-used across the project.

Server-side data updates are implemented in multiple places instead of centrally

WHY IT'S SERIOUS

This can lead to race conditions, multiple invocations, and other issues that can lead to data corruption that is hard to trace back to a root cause.

PROPOSED MITIGATIONS

Implement a robust API for doing server updates centrally.

Shield against multiple invocations, queue, and batch calls, carefully handle errors.

Other user-facing quality issues

Defects that are visible to players in the shipped product and are the kind reviewers and players call out directly.

Input on different platforms feels different

WHY IT'S SERIOUS

Changes to input can drastically affect gameplay, especially action genres. It can make the game feel broken or sluggish.

PROPOSED MITIGATIONS

Use an appropriate abstraction for input that can be used across platforms.

Use analytics or another reporting tool to explore differences in outcomes.

Config files break in hard to detect ways (syntax or formatting)

WHY IT'S SERIOUS

Produces recurring, low-grade but visible content bugs across builds.

PROPOSED MITIGATIONS

Run a sanity check against a checklist before builds.

Automate that check where possible.

Make sure (de)serialization is done with appropriate culture context.[7]

Buggy UI, especially multi-resolution support

WHY IT'S SERIOUS

UI defects are immediately visible to every player and are a common target of critical reviews.

PROPOSED MITIGATIONS

Document UI structure and layout rules.

Design reusable components and layout algorithms.

Build a reference demo as the project or house standard.

Document UI screens and flow, and test it formally.

[6] Async Guidance (David Fowler, 2018)
[7] The Turkish İ Problem and Why You Should Care (Phil Haack, 2012)

Some fonts don't support all languages the game ships in

WHY IT'S SERIOUS

Results in visibly broken or missing text for entire player segments.

PROPOSED MITIGATIONS

Build a test scene where language can be switched, with enough sample text in the game font to quickly surface font-coverage gaps.[8]

Text length and legibility vary by language and don't fit the UI

WHY IT'S SERIOUS

Causes visibly truncated or overflowing text in shipped localized builds.

PROPOSED MITIGATIONS

Build a tool to simplify and speed up checking this.

Rendering issues

WHY IT'S SERIOUS

Visual defects are highly noticeable and commonly screenshotted in negative reviews. Platform-specific issues may be especially problematic: more difficult to detect, and more difficult to solve without breaking rendering on other platforms or making the design overly complex.

PROPOSED MITIGATIONS

Regularly test on target platforms.

Document common visual issues.

Build a scene to preview all project materials under correct lighting.

Use standard tools, such as Unity's frame debugger, and RenderDoc, to debug visual issues.

Silent failures

Practices that suppress the visibility of errors, so problems accumulate invisibly and surface later, closer to release, when they're most expensive to fix.

Exceptions and error conditions are swallowed in code

WHY IT'S SERIOUS

This hides errors — underlying bugs still exist but produces no diagnostic signal. This often leads to manifestations far removed from the cause of error.

PROPOSED MITIGATIONS

Adopt fail-fast practice: flag all errors.

Implement a global hook for logging unhandled exceptions to find them.

Errors and warnings, including shader errors, are left to accumulate

WHY IT'S SERIOUS

Unresolved errors and warnings create background noise that hides genuinely new problems.

PROPOSED MITIGATIONS

Adopt a firm rule: always fix errors and warnings immediately when they appear.[9]

Async code does not surface error conditions.

WHY IT'S SERIOUS

Failures happen without a visible trace, so they are often only found through player reports post-release.

PROPOSED MITIGATIONS

Observe every async operation's faults (no un-awaited tasks, no empty catches), so failures surface instead of disappearing silently. [6]

[8] What is pseudo-localization? A practical guide for localization testing (Kinga Pomykała, 2025)
[9] The Pragmatic Programmer, Software Entropy (Andrew Hunt & David Thomas, 1999)

Logs become too long or too messy to be useful.

WHY IT'S SERIOUS

It makes it harder to spot real errors and warnings, delaying detection of genuine regressions.

PROPOSED MITIGATIONS

Specify explicitly what gets logged and what not.

Review logging regularly and remove stale entries.

Design entries to be minimal, filterable, categorized, and toggleable.[10]

Systemic issues

Root-cause structural problems that don't produce one bug — they produce a continuous stream of them across a project's lifetime.

State management is complex and global

WHY IT'S SERIOUS

Global, complex state is a classic multiplier of hard-to-trace bugs, since any part of the code can mutate it.

PROPOSED MITIGATIONS

Assign a single owner to each piece of state and expose read-only views to other systems.
Replace global access with explicit references passed at construction.
Funnel mutations through a small, traceable API rather than direct field writes.

Class hierarchies are poorly designed — most often because of bad inheritance

WHY IT'S SERIOUS

This can lead to a variety of problems, such as changes made in base classes not propagating to derived classes, memory leaks because of faulty lifetime management, and data loss because of serialization issues.
In Unity, `MonoBehaviour` lifetime methods (`Awake`, `OnEnable`, etc.) are especially problematic.

PROPOSED MITIGATIONS

Be deliberate about what can be extended and what not (default to sealed classes, and keep virtual methods minimal). [11]
Avoid designs that require upcalls, especially for lifetime management.

Event-based systems have difficult-to-trace bugs

WHY IT'S SERIOUS

Indirection between cause and effect increases diagnosis time and risk of misdiagnosis. It can also be complex to reason about, especially in complex interactions such as the negotiations in multiplayer connection logic.

PROPOSED MITIGATIONS

Keep a single dispatch path per event so call site is searchable. Log publish/handle pairs to reconstruct cause-and-effect.

Include caller info with all events, and include this in error reports and logs.

Use explicit subscription over implicit wiring, and use the IDE's tools to trace handlers.

[10] Monitoring Distributed Systems (Rob Ewaschuk, 2016)
[11] Effective Java (3rd ed.), Item 19 (Joshua Bloch, 2017)

Implicit execution order causes unexpected behavior

WHY IT'S SERIOUS

Order-dependent bugs are notoriously intermittent and hard to reproduce reliably.

PROPOSED MITIGATIONS

Avoid designs that require events to happen in a certain order where possible. Make dependencies explicit, for example, by using managers to initialize objects in the correct order.

UI bugs stemming from race conditions, especially during initialization

WHY IT'S SERIOUS

Race conditions in UI can lead to whack-a-mole situations, where a fix breaks the UI in other places.

PROPOSED MITIGATIONS

Design UI components to be self-contained views, launched from managers that populate their data directly.

Small changes often break the UI, especially across different resolutions

WHY IT'S SERIOUS

Fragile UI architecture means routine changes carry outsized regression risk.

PROPOSED MITIGATIONS

Keep hierarchies shallow, design robust general-purpose containers, and be explicit about extension and composition rules.

Write detailed specifications for layout managers. Most issues come from missing cases in the implementation.

Build a demo as a project or house standard.

Arbitrarily extending external code

WHY IT'S SERIOUS

When external code is poorly designed or implemented, extending it without deliberate containment will compound complexity across the project.
Many problems also arise from misusing external code because it is not well known or understood.

PROPOSED MITIGATIONS

Wrap external code behind a narrow interface (an anti-corruption layer), so its problems can't affect the rest of the project. [12] [13] [14]

Understand the original design intent before extending it.

Extend without adding unnecessary complexity.

Avoid accidentally duplicating existing abstractions.

Changes in one place cause bugs elsewhere

WHY IT'S SERIOUS

Indicates low modularity — the cost and risk of any single change is unpredictable.

PROPOSED MITIGATIONS

Identify the inherent design problems leading to this situation, and see if it can be fixed.

If this is not possible, then:

Use IDE reference-search tools before making changes.

Run a "where would this break?" thought experiment and test accordingly.

Document best practices and test strategies when working with affected code.

[12] Anti-Corruption Layer pattern
[13] Domain-Driven Design: Tackling Complexity in the Heart of Software (Eric Evans, 2003)
[14] Working Effectively with Legacy Code (Michael Feathers, 2004)

Bugs caused by incorrect asset import settings

WHY IT'S SERIOUS

A recurring, systemic source of defects rather than isolated mistakes, repeated on every new asset.

PROPOSED MITIGATIONS

Automate checking import settings for new assets.

Internal re-usable code has weak top-level documentation and few working examples

WHY IT'S SERIOUS

This makes code less likely to be re-used, so it will be duplicated.
If it *is* used, it is more likely to be used incorrectly.

PROPOSED MITIGATIONS

Prioritize writing good top-level documentation for reusable code.

Build and maintain demos to show how features should be used.

Assign clear ownership to keep it current as features are added.

Schedule and process risk

Issues that directly threaten timelines — either by causing late discovery of problems or by consuming time that should go to development.

Critical issues surface only just before release

WHY IT'S SERIOUS

Explicitly the highest-risk schedule failure mode: little to no time remains to fix issues found this late.

PROPOSED MITIGATIONS

Allow more schedule buffer or adjust timelines.

Feature-freeze well ahead of release.

Schedule at least a week before major releases for bug-fixing and optimization.

Be extra rigorous with last-minute changes: don't skip procedures.

Merges are treacherous when branches live too long before merging

WHY IT'S SERIOUS

Besides taking an unexpected amount of time and upsetting the schedule, complex merges can lead to many bugs that further delay the project.

PROPOSED MITIGATIONS

Integrate often: keep branches short-lived so merges never accumulate risk.[15] [16] [17]

Features that work in-editor often don't work on-device

WHY IT'S SERIOUS

Device-only failures are discovered later than editor-only failures, compressing the time available to fix them.

PROPOSED MITIGATIONS

Always test changes on real devices.

Builds fail regularly and unpredictably for no apparent reason

WHY IT'S SERIOUS

Build issues waste developer time, and make it more difficult to distinguish real regressions from build-failure noise. What may be tolerable initially can become serious as the deadline approaches.

PROPOSED MITIGATIONS

Investigate all failures, and treat regular failures as high priority. Try to find the root cause even if it is not causing serious problems now.

[15] Continuous Integration (Martin Fowler, 2024)
[16] Trunk-based development (Paul Hammant, 2017)
[17] Accelerate (Nicole Forsgren, Jez Humble, Gene Kim, 2018)

Build times are overly long

WHY IT'S SERIOUS

Slow iteration loops directly reduce the amount of testing and fixing that fits into a schedule. It also harms individual processes (for example, it is hard for developers to maintain concentration if build times are long, making it more likely to introduce errors).

PROPOSED MITIGATIONS

Invest in fast build machines.

Find ways to speed up building, such as caching assets and trimming unnecessary steps.

Track build times so sudden regressions can be spotted. [17][17]17 above

Features are implemented without a specification

WHY IT'S SERIOUS

Without specifications, it's harder to implement all cases the system needs to cater for. It's harder to design test cases and therefore know whether the system is working as expected.

This is especially problematic for issues that relate to certification where the required behavior can be complex.

If systems need to interface (such as client and server), developing without specification for the interface will almost surely lead to incompatible systems with unclear responsibilities.

PROPOSED MITIGATIONS

Always program to a written specification. Write one if it doesn't exist. [18]

Especially: always specify interfaces between systems.

Link specifications directly in tickets so testers can test against them.

Projects run on old software face unplanned forced update risk

WHY IT'S SERIOUS

An unplanned forced upgrade can land at the worst possible time in a schedule, especially if there is no buffer to absorb the time spent.
Software also has complex dependencies: upgrading one system can trigger an upgrade avalanche (forcing you to update multiple systems). It can also be tricky to find the right combination of systems (engine, platform SDK, rendering pipeline, third party tools) in some cases.

PROPOSED MITIGATIONS

Carefully check support windows for all the software you use, and proactively upgrade. Don't rely on timely completion of the project to stay within this window.

Avoid using untrusted third-party software for core components, such as multiplayer or content management, where future support failures are more likely.

Debug tools are often built late in a project

WHY IT'S SERIOUS

Issues that need those tools to diagnose are consequently also found and fixed late.

PROPOSED MITIGATIONS

Identify the debug tools a project will need as features are designed, and build them before introducing the feature.

Performance and memory are not tracked over the development of the project

WHY IT'S SERIOUS

Without a tracking, regressions aren't caught until they're already severe — often close to release.

PROPOSED MITIGATIONS

Do performance and memory profiling regularly and document the results, ideally as part of formal process.

Automate measurements that can be.

[18] Specification by Example (Gojko Adzic, 2011)

Device debugging is painful

WHY IT'S SERIOUS

Often, a debugger can't be attached. Some tools introduce lag that can make it more difficult to play. Slower, less reliable diagnosis directly extends bug-fix turnaround time.

PROPOSED MITIGATIONS

Use state-of-the art tools and practices for platform debugging instead of arbitrary workarounds.

Prioritize learning the tools and strategies early.

Introduce cheats to make it easier to play in the presence of lag.

Test coverage gaps

Structural weaknesses in how testing is done, which let real defects slip through to release undetected.

Recurring bugs aren't caught early and get re-diagnosed and re-fixed multiple times

WHY IT'S SERIOUS

Indicates no working regression safety net; the same defect can resurface post-release.

PROPOSED MITIGATIONS

Add recurring bugs to regression checklist and keep it current.

Use bug tracker as a searchable database.

It's difficult to track all bugs when many fixes land at the same time

WHY IT'S SERIOUS

This increases the chance a fix is never actually verified before the build ships. It often happens close to delivery, if many bugs are discovered and then fixed (often using external QA for the first time).

PROPOSED MITIGATIONS

Give testers a running, updated list of fixed issues instead of relying on individual tickets, ideally automatically retrieved from bug tracking system.

Some features lack enough data to test properly

WHY IT'S SERIOUS

For example, when a leaderboard does not have enough entries, many issues may be hidden, only to be discovered publicly after launch.

PROPOSED MITIGATIONS

Generate synthetic test data, ideally one order of magnitude bigger than expected. [19]

Add unit tests around these systems.

Minimal automated testing

WHY IT'S SERIOUS

Heavy reliance on manual testing alone means coverage is inherently incomplete under deadline pressure.

PROPOSED MITIGATIONS

Separate pure logic and plain old C# classes from the rest, and design unit tests for those. Consider what low hanging fruit can be automated — automatic walking through UI to take screenshots.

The game has aspects with too many permutations to test exhaustively

WHY IT'S SERIOUS

Combinatorial explosion means some tested paths are effectively guaranteed to ship untested.

PROPOSED MITIGATIONS

Investigate whether automated testing can be applied to cover these cases.

Use randomization for testing (e.g. character attributes), even if the game itself does not randomize permutations.

[19] Handling Overload (Alejandro Forero Cuervo, 2016)

The game has bugs with low reproduction rates

WHY IT'S SERIOUS

It is very hard to verify that such bugs have been fixed after an attempt, so there is a risk to ship the broken feature.

PROPOSED MITIGATIONS

Make reproduction steps for sporadic bugs extra detailed and clear.

Low reproduction rates are often caused by missing steps rather than true randomness.

Repetitive testing causes tester fatigue

WHY IT'S SERIOUS

It increases the chance of missing bugs, especially under deadline pressure.

PROPOSED MITIGATIONS

Implement high-value cheats.

Make it easy to jump straight to deep content.

Swap testers around the project frequently.

The game can only be launched from specific scenes

WHY IT'S SERIOUS

This prevents developers and testers from testing specific sections efficiently, reducing effective coverage per hour spent.

PROPOSED MITIGATIONS

Make it possible to launch the game from any level or scene.

Implement developer shortcuts to skip or clear levels.

External dependency risk

Risk introduced by code, infrastructure, or decisions outside the studio's direct control, which the team must absorb.

Teams working together commit over each other's scenes and prefabs

WHY IT'S SERIOUS

Overwrites inject bugs and destroys work already done. For developers, it is very demoralizing redoing work, especially when difficult or unpleasant.

PROPOSED MITIGATIONS

Agree on a workflow at project start covering system ownership, merge-conflict handling, and communication of tasks taken on.

Ideally use a shared tracking system.

Not everything merged into the main branch is reviewed

WHY IT'S SERIOUS

Unreviewed merges are a direct path for defects to enter the shared codebase unchecked.

PROPOSED MITIGATIONS

Always review code merged into main.

External code is inefficient, undocumented, hard to test, brittle, or contains stale code

WHY IT'S SERIOUS

This is especially a problem for hired hands, where the bad code comes from a client. The studio inherits a standing source of bugs.

PROPOSED MITIGATIONS

Identify problem areas when code is ingested. [14]

Pin risky code with characterization tests, then wrap it behind a seam to contain it.

Teams working together duplicate systems already implemented

WHY IT'S SERIOUS

This duplicates effort and systems that may need parallel maintenance.

PROPOSED MITIGATIONS

Decide on responsibilities upfront and review often. [20]

Collect requirements and design interfaces jointly.

Use a shared tracking system.

A team may not have access to all the systems when working with another team

WHY IT'S SERIOUS

For example, a team may not have access to a backend if it is owned by another team. This blocks direct investigation of backend-driven bugs, forcing slower, indirect diagnosis.

PROPOSED MITIGATIONS

Request access for any system needed as early as possible.

If not granted, plan the project workflow around that constraint from day one, deciding who does what and when.

Issues related to porting

Issues most likely to trigger a failed platform certification pass or force a re-submission cycle, which directly delays shipment.

Certification violations are often caught late

WHY IT'S SERIOUS

Certification errors are more likely to be caught by external QA, and often in systems that come into being late (such as supporting user profiles or save data). Certification failures found late force emergency fixes and re-submission right before a deadline, causing real pressure.

PROPOSED MITIGATIONS

Make a person in the team responsible for checking platform requirements.

Review them early in the project to avoid designs that make their implementation difficult.

Console button and UI terminology and imagery are often wrong, especially once translated

WHY IT'S SERIOUS

Incorrect platform terminology and button prompts are a classic, well-documented cause of certification rejection.

PROPOSED MITIGATIONS

Require testers to test this and start early in the project.

Give translators an approved list of platform-specific terms.

Add it to the cross-platform checklist.

Making platform interruptions (user switching, incoming calls, audio/haptics) robust is difficult
Making controller and network disconnect handling robust is difficult

WHY IT'S SERIOUS

There are many scenarios to cover, with complex rules. Some situations may be difficult to set up or simulate.

This is a standard requirement on consoles; incomplete coverage is a common cause of failed submissions.

PROPOSED MITIGATIONS

Maintain an extensive documented test list covering all cases.

When scheduling, be aware that the happy path is often the tip of the iceberg – covering exceptional cases is the bulk of the work.

[20] Team Topologies (Matthew Skelton and Manuel Pais, 2019)