JHAVTECH STUDIOS

The 48-Hour Code Audit:

What We Look For

The exact process, the five critical assessment areas,the most common issues we find — and what they cost if you ignore them.

This document is a complete,transparentbreakdownofhowJhavtechStudiosconductsa software code audit. We'republishingitbecausewebelievefounderswhounderstandwhata proper auditlooks like willmakebetterdecisions—andwillimmediatelyrecognisewhentheir current codebase has neverbeenassessedtothisstandard.

48h	30+	A–F	$0
in your inbox	& rescued	score delivered	no obligation

Free. No sales call. NDA signed before you share a single file.

WHY MOST CODE AUDITS MISS THE THINGS THAT MATTER

Ask ten developers to audit a codebase and you'll get ten different reports – most of them **incomplete.**

The most common audit approach is a surface-level review: does the code run, does it follow style guidelines, are there obvious bugs. This takes a few hours, produces a confident-sounding document, and misses the issues that will actually cost you money – the architecture that collapses at scale, the security gaps that expose user data, the dependency chain that breaks when a library is deprecated. The founders who come to us for a rescue have almost always received at least one of these shallow audits. The code was told to be 'fine.' It wasn't. A proper audit is not a code review. It is a systematic, multi-layer assessment of every dimension that determines whether a software product can survive real users, real load, and real business growth. This document explains exactly how we do ours – and what we find when we do.

> "We've taken over codebases that passed internal reviews, third-party audits, and developer self-assessments. Every single one had at least three issues that those reviews missed. The most expensive one had been 'audited' two weeks before a Series A close."
>
> **– Jhavtech Studios Senior Engineering Team**

SECTION 01

What Happens in a 48-Hour Code Audit

The exact sequence—from your first message to your written report.

1

Hour 0-2

Intake & NDA

You submit the audit request. Before anything else, we countersign a legally binding NDA — protecting your code, your architecture decisions, and your product concept from the first touchpoint. We then collect repository access, environment details, and any context about the known issues or concerns.

2

Hour 2-8

Architecture & Structure Review

A senior Jhavtech engineer maps the system architecture: how the layers communicate, where the data flows, what the database design looks like, and whether the core structure can support the business it's supposed to serve. This is where we find the issues that no surface review catches.

3

Hour 8-15

Security & Dependency Scan

We run both automated tooling and manual analysis across authentication systems, API exposure, data encryption, access control, and the full dependency chain. Every third-party library is checked for version, maintenance status, and known CVEs. This phase frequently produces the most alarming findings.

4

Hour 16-28

Performance, Load & Infrastructure Review

We assess data base query efficiency, caching strategy, API response design, and infrastructure configuration. We model what happens at 10x the expected load. We check deployment pipelines, environment parity, and backup / disaster recovery posture. Most codebases fail at least one of these checks.

5

Hour 28–40

CodeQuality & Maintainability Assessment

We assess the code for test ability, documentation, error handling, logging, and the presence of test coverage. A codebase that can't be maintained by someone who didn't write it is a liability — and it dramatically increases the cost of every future feature.

6

Hour 40–48

ReportWriting & Recommendations

We compile the findings into a structured, plain-English written report. Every issue is categorised by severity, given a plain-English explanation of the business risk, and accompanied by a recommended resolution path. The report is ready within 2 business days of receiving access.

SECTION 02

The 5 Critical Areas We Assess

What we check, what we commonly find, and what happens when it goes wrong.

AREA 01

Architecture & Scalability

What we check:

– Database design and schema efficiency
– Service layer separation and coupling
– API design and versioning strategy
– Stateless vs stateful architecture decisions
– Horizontal vs vertical scaling readiness

Common mistakes we find:

– Monolithic logic crammed into a single service with no separation
– Database schemas designed for demo data, not production volume
– No consideration for multi-region or high-availability deployment

Real-world consequence: A startup comes to us after 8 months of development. Their architecture works perfectly – until they run a marketing campaign and 3,000 users try to log in simultaneously. The system collapses. The fix requires rebuilding the session management layer from scratch: 6 weeks of engineering time, a delayed launch, and a $40,000 bill.

AREA 02

Code Quality & Maintainability

What we check:

– Test coverage percentage and test design
– Error handling and graceful failure patterns
– Logging and observability instrumentation
– Documentation of complex logic and decisions
– Consistency of coding standards across the team

Common mistakes we find:

– Zero test coverage – no way to verify nothing breaks when changes are made
– No error handling – failures surface as cryptic 500 errors in production
– No logging – when something breaks, there's no record of what happened or why

Real-world consequence: A company inherits a codebase from a departing contractor. There is no documentation, no tests, and no logging. Adding a single new feature requires a senior developer to spend two weeks reading code before writing a line. Every change carries a 30% chance of breaking something unrelated.

AREA 03

Security Vulnerabilities

What we check:

– Authentication and authorisation implementation
– API key and secret management (no hardcoded credentials) – Data encryption at rest and in transit
– SQL injection and XSS vulnerability exposure
– Rate limiting and brute-force protections

Common mistakes we find:

– API keys committed directly into the repository (visible to anyone with repo access)
– No rate limiting on login endpoints – open to credential stuffing attacks
– User data stored unencrypted – a breach exposes everything, readable

Real-world consequence: We audited an app before its Series A close. The developer had committed the production database credentials directly into the GitHub repository – which had been public for three weeks. Anyone who had searched GitHub in that period could have had full database access. The fix took 4 hours. The exposure risk was existential.

AREA 04

Performance & Load Readiness

What we check:

– Database query efficiency and index design
– API response time under simulated load
– Caching strategy and implementation
– Asset delivery and frontend performance
– Connection pooling and resource management

Common mistakes we find:

– N+1 database queries – the app makes 500 database calls to render a single page
– No caching – every request hits the database at full cost
– Unoptimised images and assets – 8-second load times on mobile networks

Real-world consequence: An e-commerce app launches to a modest press mention. 800 concurrent users visit in 20 minutes. The app slows to unusable within 4 minutes. The engineering team scrambles to add caching and optimise queries under live pressure – the worst possible conditions. Three days of crisis engineering follow what should have been a successful launch.

AREA 05

Deployment & Infrastructure

What we check:

– Environment parity (dev / staging / production)
– CI/CD pipeline design and reliability
– Secrets management and environment variable handling – Backup strategy and disaster recovery posture
– Infrastructure-as-code and reproducibility

Common mistakes we find:

– Production differs from staging – things work in testing but fail in the real environment
– No automated backups – a database corruption event means total data loss
– Manual deployments with no rollback capability – fixing a broken release requires 4+ hours

Real-world consequence: A SaaS product deploys a minor update manually, without a rollback mechanism. The update contains a bug that corrupts user preference data. The rollback takes 6 hours of engineering time. 200 users are notified their data was lost. Three enterprise accounts churn the following week.

SECTION 03

The Most Common Issues We Find

These appear in the majority of code bases we audit. If your project is over 6 months old, assume at least four of these are present until proven otherwise.

01

Built for MVP, not for the business it became

The database schema, architecture, and data model were designed for a 50-user demo. The business now has 5,000 users and the infrastructure is straining under load it was never designed for.

Business impact:

Partial or full architecture rebuild. Typically 4–16 weeks of engineering time.

02

No error handling — failures are invisible until users report them

The application has no structured error handling. When something fails, it fails silently or with a generic 500 error. There is no logging to trace what went wrong or when.

Business impact:

Every production bug investigation starts from zero. Mean time to resolution is 3–5x longer than necessary.

03

Secrets in the codebase

API keys, database passwords, or third-party credentials are hardcoded directly into the source code. If the repository has ever been public – or if any team member leaves – the credentials are compromised.

Business impact:

Potential full data breach. Immediate remediation required: all credentials must be rotated and vault-managed.

04

No test coverage

The application has no automated tests. Every change to the codebase is a gamble: there is no way to verify that an update hasn't broken existing functionality without manually testing every user flow.

Business impact:

Engineering velocity slows dramatically as the codebase grows. Feature delivery time doubles or triples.

05

N+1 database query problems

Theapplication makes a separate database call for each item in a list – so displaying 100 records triggers 100 database queries. At scale, this creates catastrophic performance degradation.

Business impact:

Pages that render in 0.3 seconds under test conditions take 8–12 seconds under real user load.

06

No rate limiting on authentication endpoints

Login, passwordreset, andAPI authenticationendpoints have no rate limiting. They are fully open to credential stuffing, brute-force attacks, and API abuse.

Business impact:

A targeted attack can compromise user accounts in hours. OWASP lists this as a top-10 web vulnerability.

07

Deprecated or abandoned dependencies

The application relies on third-party libraries that have not been maintained in 12–36 months. These libraries may have known security vulnerabilities and will receive no future patches.

Business impact:

Security exposure grows over time. Eventually the library breaks and the application breaks with it.

08

No backup or disaster recovery strategy

The production database has no automated backup schedule. There is no tested restore process. A corruption event, ransomware attack, or accidental deletion means permanent data loss.

Business impact:

Total data loss is possible from a single incident. For a SaaS product, this is an existential event.

09

Staging and production are not equivalent

The staging environment differs from production in infrastructure, configuration, or data. Testing in staging gives false confidence — issues only surface after deployment to production users.

Business impact:

Launch failures that could have been caught in testing instead hit real users with full impact.

10

No documentation of architectural decisions

Critical decisions — why a particular database was chosen, why a specific API pattern was used, how the authentication flow was designed — exist only in the memory of the developer who built it.

Business impact:

When that developer leaves, every future engineer must reverse-engineer intent from code. Onboarding a new developer takes 4–8 weeks instead of 1–2.

SECTION 04

What You Get in the Report

A written document that tells you exactly where you stand – before you spend another dollar.

Technical Health Score – A to F

Every codebase receives a single overall grade across five dimensions. The grade is honest. We do not soften findings to make them easier to hear.

A
Excellent. Launch-ready with minor optimisations only.

B
Good. A few issues worth addressing before scaling.

C
Moderate risk. Several issues that will cost more if left unresolved.

D
High risk. Significant structural or security problems requiring immediate attention.

F
Critical. Do not launch. This codebase presents serious risk to your business.

Top 5 Critical Issues (ranked by severity)

Not a list of every problem — a prioritised view of the five issues that most urgently need addressing. Each issue includes a plain-English explanation, the business risk it creates, and the estimated resolution effort.

Security Assessment

A dedicated security findings section covering every vulnerability identified, rated by severity (critical / high / medium / low) with remediation guidance for each.

Performance & Scalability Assessment

A specific projection: what happens to this application at 2x, 5x, and 10x current load. Where are the likely failure points, and what architectural changes prevent them.

Dependency Risk Report

A complete list of third-party libraries, their maintenance status, known vulnerabilities, and a risk rating for each. Abandoned dependencies are flagged with suggested alternatives.

Go / No-Go Launch Recommendation

A clear, unambiguous answer. Is this codebase ready to ship to real users? If not, what specifically must be resolved first, and in what order.

Rescue Roadmap (if applicable)

For codebases with significant issues, we provide a prioritised, sequenced plan: what to fix first, what can wait, and a realistic timeline to launch readiness.

SECTION 05

Fix vs Ignore: What It Really Costs

The financial case for fixing issues before launch is almost always overwhelming. Here is why – with real numbers.

Every issue identified in a code audit has two costs: the cost to fix it now, and the cost to fix it after it has caused a problem. The ratio is never favourable.

FIX IT NOW	IGNORE IT
Security vulnerability fixed pre-launch 4–8 hours of engineering time. **Cost: $800–$2,000.**	**Security breach post-launch** User data exposed. Legal liability. Regulatory notification obligations. Reputational damage. Emergency engineering. Customer churn. **Cost: $50,000–$500,000+**
Architecture rebuilt before scale 3–6 weeks of planned engineering. **Cost: $15,000–$40,000.**	**Architecture fails at scale** Unplanned downtime. Emergency re-architecture under live load. Lost users during outage. Investor confidence impact. **Cost: $60,000–$200,000+** and 3–6 months of delay.
Dependency audit and update 1–2 weeks of engineering. **Cost: $4,000–$10,000**	**Deprecated dependency causes production failure** Application breaks. Feature delivery halts until replaced. If a CVE is involved, security exposure during repair window. **Cost: $20,000–$80,000+**

FIX IT NOW

Test coverage added during build

20–30% additional development time.
Cost: Factored into original build.

IGNORE IT

No test coverage when team scales or changes

Every new feature risks breaking existing functionality. Senior developer time consumed by manual regression testing. Engineering velocity degrades by 40–60% as codebase grows. Cost: Ongoing, compounding.

In 12 years of building and rescuing software products, we have never seen a case where ignoring a code audit finding was cheaper than fixing it. The only variable is when you pay — and whether you pay in money alone, or in money plus reputation plus time.

YOU'VE SEEN HOW WE WORK.

Now let us apply it to your codebase.

The 48-hour code audit is free. It's written. It's honest. And it's the fastest way to know whether the product you've invested in is ready for what comes next — or whether it has a problem you don't know about yet.

What happens when you request the free audit:

1. You fill in the request form
90 seconds. Tell us your platform, your concern, and what you want to know.

2. We sign your NDA
Before you share a single file. Legally binding. Countersigned within 24 hours.

3. You share codebase access
Repository access and any relevant context about your application.

4. We run the full audit
A senior Jhavtech engineer runs every assessment described in this document.

5. You receive your written report
Within 48 business hours. Every finding explained in plain English.

6. You decide what happens next
No pressure. No follow-up call required. The report is yours to keep.

Written guarantee: if we identify issues and you commission Jhavtech to fix them, we fix the agreed scope or the remaining balance is cancelled. **This is in the contract — not the pitch**

Get Your Free 48-Hour Code Audit

jhavtech.com.au/software-project-rescue-service

Free. Written report. NDA before access. No obligation to proceed.
Response within 4 business hours during AEST business hours

Jhavtech Studios | Melbourne, Australia | jhavtech.com.au | (03) 9344 1619
Australia's Software Rescue & Build Studio | 13+ years | 100+ apps | Zero offshore