Technical Debt Checklist

Seven areas, 42 questions and a simple 0–21 score to assess how much technical debt your system carries — in one afternoon, with the people who know it best.

How to use this checklist

1. Set aside an afternoon with two or three people who know the system well: a tech lead, a senior developer and someone who runs it in production.
2. Answer every question with yes or no. All questions are phrased so that "yes" means healthy and "no" points to debt. Count "we don't know" as no — a missing answer is a finding in itself.
3. Score each area from 0 to 3 by the number of "no" answers: none = 0, one or two = 1, three or four = 2, five or six = 3. A 0 means no issue found; a 3 means serious risk.
4. Add up the seven areas for a total between 0 and 21 and read the matching band below. Any single area scoring 3 deserves attention on its own, whatever the total. The bands are a rule of thumb to start a conversation, not an industry standard. If AI assistants contribute to your codebase, answer the AI-generated code questions separately.
5. Repeat the exercise every quarter or two. The trend tells you more than any single score.

Score sheet

Fill in one row per area after you answer its questions.

AREA	SCORE (0–3)	NOTES
1. Code quality		
2. Architecture		
3. Testing		
4. Dependencies and platforms		
5. Security		
6. Delivery and operations		
7. Team, knowledge and process		
Total	/ 21	

AREA 1/7

Code quality

How easy it is to read, change and reason about the code your team touches every week.

QUESTIONS — ANSWER YES OR NO

- ☐ Can a new developer make a small, safe change in the most frequently edited module within their first week?
- ☐ Do you know which files change most often and how complex they are (your hotspots)?
- ☐ Are new static analysis findings (linter, type checker, code quality scanner) fixed before a pull request is merged?
- ☐ Is there one agreed way to handle common concerns such as error handling, logging and data access, instead of several competing patterns?
- ☐ When a business rule changes, do you update it in one place rather than in several copies of the same logic?
- ☐ Are commented-out code, dead code and unused feature flags removed regularly?

Number of "no" answers: ____ → area score (0–3): ____

WARNING SIGNS

- "Only one person dares to touch that module."
- Estimates for small changes in certain parts of the code are routinely inflated "because it is that module".
- Linter or analyser warnings counted in thousands, with nobody looking at them.

EXAMPLE METRICS

- Example: change frequency × complexity per file (hotspots), taken from version control history
- Example: number of files touched by a typical small change
- Example: trend of static analysis findings between releases (the direction matters more than the absolute number)

AREA 2/7

Architecture

Whether the structure of the system still matches what the business needs it to do.

QUESTIONS — ANSWER YES OR NO

- ☐ Is there an up-to-date description or diagram of the main components and how they communicate, and does the team agree it reflects reality?
- ☐ Can you change one business area (for example billing) without coordinated changes in several others?
- ☐ Are module or service boundaries enforced by code structure or tooling, not only by convention?
- ☐ Do you know where the system will hit its limits first if traffic or data volume doubles?
- ☐ Are key architectural decisions recorded together with the reasons behind them (for example as architecture decision records)?
- ☐ Does each core piece of data have one source of truth, rather than copies kept in sync by scheduled jobs or manual fixes?

Number of "no" answers: ____ → area score (0–3): ____

WARNING SIGNS

- Almost every feature needs a "small change" in a shared database schema or a common module that several teams depend on.
- Services that can only be deployed together, in a fixed order.
- Workarounds justified with "the architecture does not allow it".

EXAMPLE METRICS

- Example: share of changes that touch more than one module or service
- Example: number of components that must be released together for a typical change

AREA 3/7

Testing

Whether automated tests give the team enough confidence to change code without fear.

QUESTIONS — ANSWER YES OR NO

- ☐ Do automated tests run on every pull request and block the merge when they fail?
- ☐ Are the most critical business paths (payments, sign-up, core calculations) covered by automated tests the team trusts?
- ☐ Is the test suite fast enough that developers run it before pushing, instead of waiting for CI to tell them?
- ☐ Are flaky tests fixed or quarantined within days, rather than re-run until they pass?
- ☐ Can you reproduce a production bug in a test environment with realistic, anonymised data?
- ☐ When a bug is fixed, is a test added that would have caught it?

Number of "no" answers: ____ → area score (0–3): ____

WARNING SIGNS

- "Just re-run the pipeline, that test is flaky."
- Manual regression testing before every release that takes days.
- A high coverage figure, yet regressions still reach production in areas that are "covered".

EXAMPLE METRICS

- Example: duration of the full test suite and how it changes over time
- Example: share of pipeline runs that fail for reasons unrelated to the change (flakiness)
- Example: production defects in areas with automated tests compared with areas without them

AREA 4/7

Dependencies and platforms

The libraries, frameworks, runtimes and platforms the system relies on — and how far behind they are.

QUESTIONS — ANSWER YES OR NO

- ☐ Do you have a current inventory of direct and transitive dependencies (for example a software bill of materials)?
- ☐ Is every part of your core stack (language runtime, framework, database) on a version that still receives security updates?
- ☐ Are dependency updates handled routinely, for example automated update pull requests reviewed every sprint, rather than as rare big-bang projects?
- ☐ Is it clear who decides about adopting, upgrading or replacing a key library?
- ☐ Are the open-source licences of your dependencies known and checked against the way you distribute the product?
- ☐ Could you build and deploy the system from scratch today without relying on an archived package, an unmaintained private mirror or one person's laptop?

Number of "no" answers: ____ → area score (0–3): ____

WARNING SIGNS

- A major framework upgrade that has been "planned for next quarter" for more than a year.
- Pinned versions with comments such as "do not upgrade — breaks X".
- Components past their end of life still running in production.

EXAMPLE METRICS

- Example: number of dependencies more than one major version behind
- Example: age of the oldest known, unpatched vulnerability in your dependencies
- Example: time from a library's security release to its deployment in production

AREA 5/7

Security

Weaknesses built into the code and the process — the part of technical debt that can turn into an incident overnight.

QUESTIONS — ANSWER YES OR NO

- ☐ Is the repository, including its full history, free of secrets such as API keys, passwords and tokens?
- ☐ Do dependency and static security scans run automatically, with an owner who triages the findings?
- ☐ Are authentication and authorisation handled in one consistent, well-understood place rather than re-implemented endpoint by endpoint?
- ☐ Do you know where personal and sensitive data is stored, and is access to it limited and logged?
- ☐ Is there a tested procedure for rotating credentials and for responding to a security incident?
- ☐ Has someone outside the team reviewed the system's security (code review, penetration test) in the last 12 months, and were the findings closed?

Number of "no" answers: ____ → area score (0–3): ____

WARNING SIGNS

- Shared admin accounts or passwords that "everyone on the team" knows.
- Credentials pasted into chat, tickets or configuration files "just for now".
- Security findings marked "won't fix" with no documented reason.
- Nobody can say quickly which services are able to read customer data.

AREA 6/7

Delivery and operations

How code gets from a developer's machine to production, and how much friction debt adds along the way.

QUESTIONS — ANSWER YES OR NO

- ☐ Is deployment to production automated and repeatable, with no manual steps that only certain people know?
- ☐ Can you release a small change on any working day, not only in a planned release window?
- ☐ Can a bad release be rolled back or switched off with a feature flag within minutes?
- ☐ Are environments (test, staging, production) defined as code and kept consistent with each other?
- ☐ Does monitoring tell you about incidents before your customers do?
- ☐ Is time for paying down technical debt planned explicitly in the roadmap, not squeezed in "when there is time"?

Number of "no" answers: ____ → area score (0–3): ____

WARNING SIGNS

- Releases that need a weekend, a long checklist and one particular person.
- "It worked on staging" incidents caused by differences between environments.
- Lead time for small changes growing quarter after quarter while the team stays the same size.

EXAMPLE METRICS

- Example: lead time for changes, from commit to production
- Example: deployment frequency
- Example: change failure rate and time to restore service

AREA 7/7

Team, knowledge and process

Knowledge, ownership and habits — debt that lives in people and processes, not only in the code.

QUESTIONS — ANSWER YES OR NO

- ☐ Is every critical component understood by at least two people who are still on the team?
- ☐ Is the documentation good enough for a new developer to set up the project and ship a first change without constant help?
- ☐ Is technical debt visible in the backlog as concrete items, each with an estimate of its cost or impact?
- ☐ Do code reviews address design and risk, not only style?
- ☐ Can your engineers explain debt to product owners and management in business terms: delay, risk and cost?
- ☐ If your main vendor or a key engineer left next month, would you know exactly what you are inheriting?

Number of "no" answers: ____ → area score (0–3): ____

WARNING SIGNS

- Onboarding takes months and depends on the availability of one senior person.
- Technical debt is discussed only after an incident or a missed deadline.
- The same people are always on call for the same systems.
- Changes that reach production before anyone other than the author has read them.

EXAMPLE METRICS

- Example: number of components that only one person really understands
- Example: time it takes a new developer to ship their first change to production
- Example: share of sprint capacity spent on unplanned work

NOT PART OF THE 0–21 SCORE

AI-generated code: four extra questions

- ☐ Do you know which parts of the codebase were written largely with AI assistants?
- ☐ Is AI-generated code held to the same review and testing standards as code written by hand, with tests that check real behaviour rather than mirror the implementation?
- ☐ Do you verify dependencies suggested by an assistant before adding them (they exist, they are maintained, their licence fits) and scan generated code for hard-coded secrets?
- ☐ Can the team explain and maintain the AI-generated code without regenerating it from scratch?

Reading your total score

A rule of thumb to start a conversation, not an industry standard.

0–5 Under control

Debt exists, as it does in every system, but it is visible and managed. Keep the routine going: regular dependency updates, debt items in the backlog and a review every quarter.

6–10 Noticeable

Debt is starting to slow delivery in specific places. Pick the one or two areas with the highest scores and plan concrete work there before the problem spreads.

11–15 Significant

Debt is shaping your roadmap: changes take longer than they should and risk is building up. This is the moment for a structured review and a prioritised remediation plan.

16–21 Critical

The system carries risk that can surface as an incident, a failed release or a stalled roadmap. Get an independent assessment before you make major commitments that depend on it.

Next steps

When an internal review is enough

An internal review is usually enough when the total is low or the high scores sit in one or two areas the team already understands, when the people who built the system are still on the team, and when the main goal is to plan improvement work. Turn every "no" into a backlog item with an owner, and come back to the checklist next quarter.

When an external audit makes sense

An external audit makes sense when the stakes or the uncertainty are high: before an investment or acquisition, when you inherit a codebase or a vendor is leaving, when large parts of the code were generated with AI assistants, when many answers are "we don't know", or when the team and management disagree about how serious the debt is. An independent reviewer reads the code without the history and assumptions of the people who wrote it, and turns the findings into priorities that decision makers can act on.

Want an outside view?

Independent code audit and technical due diligence: ardura.consulting/technical-due-diligence/