GUIDE · LYRIKS

The legacy takeover checklist

How to take over code nobody understands anymore, and change it without breaking it.



lyriks.io

Contents

Who this guide is for

1. Before you touch anything

2. Recover the rules and the behavior

3. Recover the technical side, and link it to the rules

4. Find where the sources disagree

5. Secure the first change

6. Decisions to make with the business

7. A 30/60/90-day plan

8. Common mistakes

9. How Lyriks helps

Who this guide is for

You just inherited a system. Maybe you joined as CTO, maybe a reorganization handed your team a platform built by people who have since left, maybe an acquisition came with a codebase attached. The software runs. Customers depend on it. And the honest answer to most questions about it is "we think so."

This guide is a working checklist. Print it, tick the boxes, hand pages to your leads. It assumes nothing about your stack or your tools. The goal is modest and concrete: within roughly ninety days, know one or two parts of the system precisely, have the business agree on what they should do, and ship a first change you can stand behind.

> **Why this is now a board topic.** In a study of 1,000 senior executives at Global 2000 companies, Cognizant expects the share of IT budgets spent on modernizing legacy systems to rise from 26% to 43% by 2030, and 85% of those executives worry about whether their systems can integrate AI. (Cognizant, October 2025, cognizant.com.)

One idea runs through every section. The code tells you what the system does today. It does not tell you what it was supposed to do. Your job in a takeover is to recover both, put them side by side, and let the right people decide when they differ.

---

1. Before you touch anything

The first two weeks are for inventory, not for fixes. Every quick patch you ship before you know the terrain adds one more change nobody can explain later. Resist it, unless production is on fire.

Code and environments

- [ ] List every repository, including the forgotten ones: scripts, infrastructure, mobile apps, internal tools, the "temporary" service from four years ago.
- [ ] Confirm you can build each one from a clean machine. Write down every step that was not documented.
- [ ] Map the environments (production, staging, test, local) and note how they differ. Config drift between staging and production is a classic source of surprises.
- [ ] Find out how deployments actually happen. Pipeline, manual script, someone's laptop.
- [ ] Check who holds access to production, cloud accounts, DNS, certificates and third-party dashboards. Move ownership to named people in your team.
- [ ] Locate the backups and test one restore. A backup nobody has restored is a hope, not a backup.
- [ ] Collect the scheduled jobs: cron entries, batch runs, queue consumers. They often carry business rules nobody remembers.

Written sources

- [ ] Gather the ticket tracker, wiki pages, design files, architecture diagrams, contracts with integrators and any audit reports.
- [ ] Note the last update date on each document. Treat anything older than the last major release as a lead, not a fact.
- [ ] Export the incident history for the past year. Where incidents cluster, knowledge is thin.

People

- [ ] Build a "who knows what" map: for each area of the system, the person who knows it best, inside or outside the company.
- [ ] Reach the people who left while the door is still open. One hour with a former developer can save weeks.
- [ ] Identify the business owners: the people who can say what a rule should be, not just what it does.
- [ ] Ask support and operations which parts of the system they work around every week.

- [ ] Record every interview in a shared place. Knowledge in your head is the same problem you inherited.

> **Freeze rule.** During the inventory, agree with stakeholders that only urgent fixes ship, and each one is written down with its reason. You buy two calm weeks and a clean baseline.

---

2. Recover the rules and the behavior

Start with behavior, not code structure. A business stakeholder cannot validate a class diagram, but they can validate "a customer is refunded within 30 days." This is the start of the specification you never received, rebuilt from the evidence. The technical side, in section 3, belongs to the same spec: one model of the system, not two documents.

Do not attempt the whole system. Pick the one or two modules that matter most (more on that choice in section 5) and go deep there.

- [ ] **Features.** What the module offers, in words a user would recognize.
- [ ] **Journeys.** The paths people take through it, from first screen to outcome, including the unhappy paths.
- [ ] **Actions.** What a user or another system can trigger, with its inputs.
- [ ] **Business rules.** Thresholds, deadlines, eligibility conditions, calculations, exceptions. Write each rule as one sentence a non-engineer can read.
- [ ] **Screens.** Each screen, what it shows, what it lets you do, and which roles see it.
- [ ] **Data.** The business objects and their attributes, as the business names them: order, contract, refund, customer.
- [ ] **Invariants.** What must always be true. "An order is never both shipped and cancelled." These are the guardrails your future changes must respect.
- [ ] **Testable requirements.** For each rule, a concrete case you could check: given this input, expect that result.

Where do the rules hide? In conditionals, obviously. Also in database constraints, stored procedures, validation on forms, configuration files, feature flags, batch jobs and the occasional hardcoded date. Look in all of them.

> **Write rules as sentences, not code.** "Orders over a set amount require a second approval" is a rule. A pasted `if` block is not. The test is simple: could the person who owns this rule read it and say yes or no?

3. Recover the technical side, and link it to the rules

The same spec also covers the technical side: the architecture, data and integrations that carry the behavior you just listed. You need it to estimate, to plan changes and to know what a modification will disturb.

- [ ] **Architecture.** The main components and how they talk to each other. One page, kept current, beats a beautiful diagram from 2019.
- [ ] **Modules.** The boundaries that exist in the code, and the ones that only exist on paper.
- [ ] **Dependencies.** Internal dependencies between modules, plus external libraries with their versions. Flag anything unmaintained or with known vulnerabilities.
- [ ] **Data model.** Tables or collections, keys, relations, and the columns whose meaning nobody is sure about.
- [ ] **APIs.** Every endpoint the system exposes and every one it calls, with who consumes them.
- [ ] **Flows.** How data moves for the critical journeys: from screen to service to database to downstream system.
- [ ] **Integrations.** Payment providers, ERPs, identity, email, file transfers. Note the contract, the owner on the other side and what happens when it is down.
- [ ] **Technologies.** Languages, frameworks, runtimes and databases, with end-of-support dates.

Link every rule to its code

This is the step most takeovers skip, and the one that pays off later. For each business rule, note where it lives in the code. For each critical file or table, note which rules depend on it.

RULE (WHAT)	IMPLEMENTATION (HOW)
A customer is refunded within 30 days.	`refund.py`, the Payment API, the `refunds` table
Your rule here	File, service, table
Your rule here	File, service, table

With this mapping, a change to `refund.py` tells you which rules are at stake, and a change in refund policy tells you where to act. Without it, every estimate includes a large, invisible margin for fear.

4. Find where the sources disagree

In an inherited system, the code is rarely the only witness. There are tickets, wiki pages, design mockups, commit messages, and what people remember. They will not all agree. Finding the disagreements is not a side effect of the takeover. It is one of its main deliverables.

> **Illustrative example.** On the refund deadline, a Jira ticket says 14 days, the Confluence page says 30 days, and the code applies 21 days. Three sources, three answers. Support quotes one number, finance plans with another, and customers get the third.

Notice what the code gives you in that example: the truth about current behavior, and nothing more. Whether 21 days is right is a business question. Maybe the code is wrong. Maybe the wiki is out of date. Maybe a past decision was never written down anywhere.

- [ ] For each rule in your pilot module, compare what the code does with what the tickets, wiki and design files say.
- [ ] Log every mismatch as a named gap: the rule, each source and its value, the date of each source.
- [ ] Assign each gap to the business owner who can settle it. No owner means you found another problem worth flagging.
- [ ] Separate gaps by impact: money, legal or compliance, customer experience, internal only.
- [ ] Do not fix gaps silently in the code. A quiet fix recreates exactly the knowledge loss you are trying to repair.
- [ ] Keep the log alive. Each settled gap becomes a confirmed rule with a date and a decision maker.

A list of precise gaps changes the mood of a team. The vague dread of "we don't know what this does" becomes twelve questions with owners. Twelve questions can be worked through.

---

5. Secure the first change

The point of all this work is to change the system again. Do it on a narrow front first.

Pick one or two modules

Good candidates have at least two of these traits:

- [ ] A change is already on the roadmap for it.
- [ ] Incidents keep coming back to it.
- [ ] Other parts of the system depend on its output.
- [ ] People get noticeably careful when it comes up in meetings.
- [ ] It is small enough to understand in a few weeks.

A narrow scope gets you a result in weeks rather than quarters, gives the business something concrete to validate, and produces real evidence for deciding whether to go wider.

Put safety nets on the boundaries

You will not get full test coverage on legacy code, and you do not need it. You need tests where the module meets the rest of the world.

- [ ] Write characterization tests on the module's inputs and outputs: given today's input, assert today's output, even if today's output looks odd.
- [ ] Cover each confirmed business rule with at least one test derived from its testable requirement.
- [ ] Cover each invariant. These tests protect you from the regressions that hurt most.
- [ ] Test the integrations at their contract: request shape, response shape, error handling.
- [ ] Run the tests in the pipeline on every change, not just before releases.
- [ ] Add monitoring on the module's key outputs in production, so you see drift before customers do.

Write the rules down before you change them

- [ ] Before the change, list the rules it touches, the ones it must leave alone, and the invariants that must still hold.
- [ ] Have the business owner approve the target rules, not the code diff.
- [ ] After the change, check each rule again: still respected, changed as intended, or broken.
- [ ] Watch for what was added without being asked for. Unrequested behavior is a risk too.
- [ ] Update the rule log and the rule-to-code mapping in the same pull request as the code.

> **If AI writes part of the code.** Give the assistant the approved rules and their sources, not just the ticket. Then check what comes back against those rules, one by one. Reviewing the diff line by line tells you the code is plausible. Checking it against the rules tells you it does what was decided.

---

6. Decisions to make with the business

Several questions in a takeover are not technical, even if engineers are the first to notice them. Put them in front of the right people early, in writing.

DECISION	WHO DECIDES	WHAT YOU BRING
Which value is right when sources disagree	Rule owner (product, finance, legal, operations)	Each source, its value and date, current behavior and its impact
Which modules to stabilize first	Sponsor with product	Incident history, roadmap pressure, dependencies
Whether to keep, refactor or migrate a module	Sponsor with CTO	Rebuilt spec, technical health, end-of-support dates
What can be retired	Product with support	Usage data, features nobody can explain
Acceptable risk for the first change	Sponsor	Rules touched, tests in place, rollback plan
Who owns each rule from now on	Management	The gap log and the "who knows what" map

- [ ] Schedule a short, recurring review where rule owners settle open gaps.
- [ ] Record each decision with its date and its author, next to the rule it concerns.
- [ ] Agree on what "done" means for the pilot before it starts.
- [ ] Agree on how bugs found during the takeover are prioritized against roadmap work.

7. A 30/60/90-day plan

Adjust the rhythm to your size and your system. The order matters more than the dates.

PERIOD	FOCUS	OUTCOMES YOU CAN SHOW
Days 1 to 30 Inventory and first understanding	Repositories, environments, access, backups. "Who knows what" map. Interviews. Choice of one or two pilot modules. First pass on what the pilot does.	Every repo builds from a clean machine. Access is in named hands. Pilot chosen and agreed. Draft list of features and rules for the pilot.
Days 31 to 60 Spec, docs and gaps	Business rules, journeys, data and invariants for the pilot. Architecture, data model, APIs and integrations. Rule-to-code mapping. Comparison with tickets and wiki.	Rebuilt spec reviewed by rule owners. Technical documentation for the pilot. Gap log with owners. Characterization tests on module boundaries.
Days 61 to 90 First controlled change	Settle priority gaps. Plan the first change against the rules. Implement, test, check rule by rule. Retrospective.	First change shipped with rules checked before and after. Decision log. Honest estimate for extending the approach to the next module.

At day 90 you will not know the whole system. You will know one part of it precisely, with agreed rules tied to the code and a short list of open questions. That is enough to change it with confidence, and enough to plan the next part.

8. Common mistakes

Trying to understand everything at once

The scope grows, nobody can validate anything, and the effort stalls before its first useful result. One module done properly beats ten done halfway.

Treating the code as the source of truth

The code is the source of the implementation. It shows what happens, including the bugs. What should happen is what the business intended, and that has to be recovered and confirmed.

Generating a mountain of documentation

Pointing an AI assistant at the repository and asking for documentation produces dozens of pages quickly. The system is now documented, and still out of your hands. Those pages link to nothing, nobody can verify them, and they age with the next commit. Aim for documentation that is structured, checkable and tied to the code.

Skipping the business review

A spec only engineers have read is still a guess. The person who says "that exception was removed two years ago" is worth more than another week of analysis.

Waiting for perfect sources

The tickets will never be complete and the wiki will never be consistent. Start with what exists. The inconsistencies are part of what you are looking for.

Rewriting before understanding

A rewrite of a system you do not understand reproduces its bugs and loses its undocumented rules. Recover the rules first. Then decide whether to refactor, migrate or leave alone.

Letting knowledge leave again

If the rebuilt spec lives in one engineer's notes or one assistant's chat history, you are setting up the next takeover. Keep it in a shared place your team owns and can export.

---

9. How Lyriks helps

Everything above can be done by hand, and for a small module it should be, at least once. The work gets heavy when you have several modules, many sources and changes arriving every week. That is where Lyriks fits.

- **Reverse specification.** It reads the code and rebuilds one spec that covers it all: features, journeys, actions, business rules, screens, objects and attributes, invariants and testable requirements, and also architecture, modules, dependencies, data model, APIs, flows, integrations and technologies. Each rule points to the code that implements it.
- **One knowledge base.** Code, Jira, Confluence, Notion, Figma and GitHub land in a graph database, not a RAG, with every element tied to its source. Inconsistencies between sources are flagged.
- **Impact before code.** A planned change is analyzed against the spec: what it touches, what stays compatible, what conflicts, what needs a decision.
- **Rule-by-rule checks.** The approved spec goes to your AI assistant (Claude Code, Cursor, Copilot via MCP). The result is checked against each rule by a deterministic engine: same code, same spec, same verdict. Each release gets an implementation report: conforming, added, missing, regressed.
- **On your terms.** Open source (AGPL), installed in one command, on-premise or airgapped, with your own LLM and tokens. The project memory stays with you and can be exported in full.

The recommended start matches this guide: a pilot on one or two modules.

Take back the system you inherited

Start with one module. Rebuild its rules, tie them to the code, let the business settle the gaps, then change it under watch.

See the legacy use case at lyriks.io/use-cases/legacy or start a 14-day trial at lyriks.io/trial.