DESIGNSYSTEMS.ONE · PLAYBOOK TRACK

The Migrate Playbook

Five chapters on getting an existing product onto a design system without burning out the team or stalling product work. Print edition, with the worksheets.

CONTENTS

41	5	11	10	A4
PAGES	CHAPTERS	FIGURES	WORKSHEETS	PORTRAIT

Every chapter in this book is published free, in full, at designsystems.one/playbook/migrate. It was free before this file existed and it stays free. What you bought is the print typesetting, the figures and the worksheets — and a licence to hand all three to your team.

CONTENTS

Everything in this file, with page numbers.

The web version is free. This is the same text, packaged.

All five chapters of this playbook are readable in full, for nothing, at designsystems.one/playbook/migrate. They were free before this PDF existed and they stay free after it. There is no bonus chapter behind this cover.

WHAT YOU ACTUALLY PAID FOR

Three things. **Print.** The chapters typeset for A4 and for the reader who marks up margins rather than keeping fourteen tabs open. **Figures.** Eleven of them, drawn for this edition — the arc, the tier diagram, the prioritisation matrix, the rollback flow, the burn-down. **Worksheets.** Ten printable pages that turn each chapter's homework into a form you can fill in with a pen and hand to whoever is running the audit.

And a team licence: one purchase covers internal use across your company. Print it, drop it in the shared drive, hand the worksheets round at the kickoff.

Read it in the order it is bound

The sequence is the argument. Audit before architecture, because the plan you write before the count is the plan you throw away. Bridge before big-bang, because the giant pull request does not merge. Primitives before composites, because composites built on unstable primitives are rework. Engineering path before enforcement, because a lint rule at error level against a team with no codemod is just a way of making enemies. Skipping ahead is allowed; skipping the order is the mistake the whole book is about.

About the numbers in the figures

Every figure in this book is an illustrative worked example, and every caption says so. They are drawn from the shapes migrations actually make — the eighty-odd hex values where twenty were expected, the plateau at seventy per cent — not from a measurement of a specific company. Use them to recognise your own shape, then fill the worksheets with your own numbers. Your numbers are the only ones that will convince anyone.

The worksheets

Two per chapter, at the end of each chapter, on their own pages so they photocopy cleanly. They chain: the collapse table in 1B becomes the alias map in 2A; the score column in 1B becomes the plot in 3B; the plot in 3B becomes the wave plan in 3A; the retirement contract in 2B is the first line of the cutover checklist in 5B. Filling them in order produces a migration plan. Filling in only the interesting ones produces a document.

THE WHOLE THING, AT A GLANCE

The migration arc.

Five chapters, five phases, sized by their share of an illustrative twenty-eight week migration. The widths are the point: the audit is short, the componentisation is long, and the enforcement tail is longer than anyone budgets for.

FIGURE 0.1 — THE ARC, PROPORTIONAL

- **Audit** — Ch. 01 — inventory, ownership, cost scores
- **Bridge** — Ch. 02 — aliases, lint at warn, retirement criterion
- **Componentise** — Ch. 03 — four waves, primitives before composites
- **Engineer the path** — Ch. 04 — codemods, escape hatches, office hours
- **Adopt and enforce** — Ch. 05 — lint at error, per-product metrics, cutover

Illustrative durations. The shape is what transfers: a short, cheap counting phase; a bridge that must exist before any bulk change; a long middle where most of the work happens in four ordered waves; and a final phase that is mostly social — lint escalation, per-product conversations, and a ceremony that says the thing is finished.

THE ONE THING TO TAKE FROM THIS PAGE

Phases three, four and five overlap in practice — you will be componentising wave 3 while engineering the path for wave 2 and already enforcing on wave 1. They are drawn end to end because the *dependencies* are strictly ordered, not because the calendar is. What you cannot do is start phase two before phase one, or run phase five against a surface that never got phase four.

01

Audit the surface area

Before you migrate anything, count what's there — and what's pretending to be one component but is actually six.

6	~80	0
Button implementations found where the team expected one	**Distinct hex values, against an expected ~20**	**Lines of migration code written during the audit**
illustrative audit	the usual gap	counting is the work

Every legacy product UI is a museum of accumulated decisions. Three button styles that look identical but render with different padding. Five card components built by five teams over five years. Two form-validation patterns, one of which is a half-finished refactor someone abandoned in 2021. The migration starts with an honest inventory of what you actually have.

Audit before architecture. The migration plan you write before the audit is the migration plan you'll throw out three months in.

> **The output is a flat spreadsheet, not a Figma file. Volume tells you sequence.**

IN THIS CHAPTER

01 Inventory components, not pages

Walk every product surface and list every component — Button, Input, Card, Modal, Toast — that gets used. Note variants that should be the same component but aren't (PrimaryButton vs Button with primary prop). The output is a flat spreadsheet, not a Figma file.

Count instances. The Button used on 800 pages is not the same migration as the Banner used on 3 pages. Volume tells you sequence.

Do the walk route by route, with the running app on one screen and the spreadsheet on the other. Six columns is enough: component name, the file it actually resolves to, variant, instance count, owning team, notes. Resist the urge to editorialise in the notes column during the walk — you are counting, not designing. Judgement comes after the count, and it is a better judgement for having the count in front of it.

Get instance counts from the codebase, not from memory. A ripgrep for the import specifier across the repo is a truer number than anyone's estimate, and it takes a minute. Keep the raw command in the spreadsheet so the number is reproducible next quarter, when someone asks whether the migration moved at all.

02 Inventory tokens — primitives and semantics

Same audit, color and spacing layer. Pull every hex value from the codebase. Pull every spacing value (px, rem, em). Group near-duplicates: #4f46e5 and #5046e5 are probably the same color, just typo'd.

Most teams discover they have ~80 colors when they expected ~20. That's the gap the system has to close.

Group by perceptual distance, not string equality. Convert everything to a uniform space and collapse anything within a small delta into one candidate token, then eyeball the collapsed set. Two greys that differ in the third hex digit are one grey with a typo in its history; two greys that differ visibly are two greys, and one of them probably has a reason nobody wrote down.

The collapse is the first real decision of the migration, and it is a design decision made by counting. Record, for each collapsed group, the value you kept and the values you retired. That record becomes the first half of the bridge in the next chapter — you will not have to derive it twice.

03 Map ownership

Each component has a current owner — usually informally. Track them down. The migration won't happen by talking to design or platform; it happens by talking to the team that ships the surface using the component.

If three teams all own the same Card with three different implementations, the migration starts with a conversation between them, not with code.

Ownership that exists only in someone's head is not ownership; it is a single point of failure with a friendly face. Write the name in the spreadsheet, then send that person the row and ask them to confirm it. Half of them will say it is not theirs and name someone else. That correction is worth more than the original guess.

Components with no owner are the most useful finding in the audit. They are usually old, widely used, and the reason the last refactor stalled. Assign them before you migrate them, even if the assignment is temporary and to your own team.

04 Score the migration cost

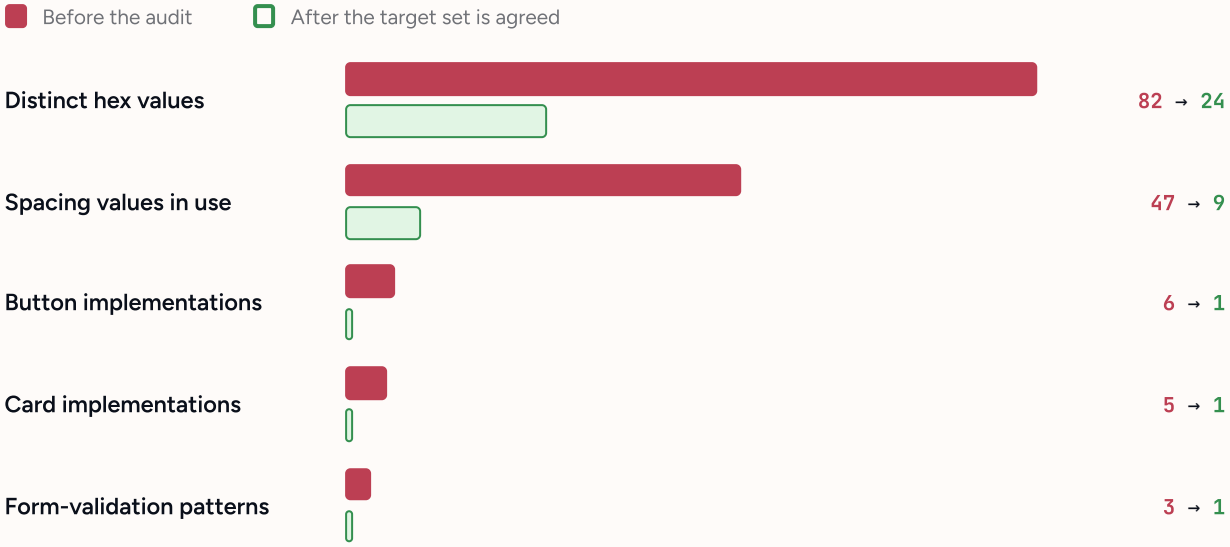
For each component, score: complexity (1-3), instances (low/med/high), API surface (compatible / partially / breaking). The cheapest migrations are high-instance simple components with compatible APIs — Button, Input. The expensive ones are low-instance complex components with breaking APIs — Modal, DataTable.

Sequence the work: cheap-and-high-impact first, build momentum, then tackle the expensive ones with the team's buy-in already won.

Multiply, do not add. Complexity 3 with a breaking API is not one point worse than complexity 3 with a compatible API — it is a different category of work, because a breaking API means every call site needs a human read. Scoring multiplicatively spreads the components out along a range you can actually sequence against, instead of clustering them all in the middle.

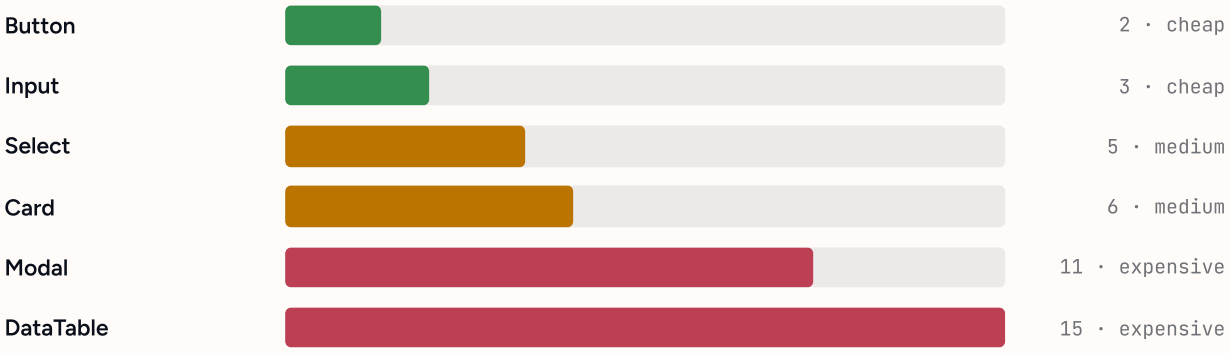
Sanity-check the ranking against the team's intuition once, and then trust the ranking. Where they disagree, the usual cause is an unrecorded constraint — a contract deadline, a regulated surface, a rewrite already in flight. Add it as a column rather than fudging the score.

FIGURE 1.1 — SURFACE AREA, BEFORE AND AFTER



Illustrative worked example, not a measurement of any particular product. Solid bars are what the audit found; outlined bars are the target set agreed after the collapse. The point is the ratio, not the absolute numbers: the audit's job is to make that ratio visible before anyone writes a migration plan against it.

FIGURE 1.2 — MIGRATION COST SCORE, CHEAPEST FIRST



Complexity × instance count × API compatibility, plotted for one illustrative component set. Read it as a sequence, not a schedule: the cheap end is where momentum comes from, the expensive end is where it gets spent.

TAKEAWAYS

01 Inventory components and tokens flatly — every variant, every instance, every hex value.

02 Map current ownership of each component before proposing changes.

03 Score migrations by complexity × instance count × API compatibility.

04 Sequence cheap-and-high-impact first to build momentum before tackling the expensive cases.

05 Keep the audit reproducible: store the commands, not just their answers.

USE THE SYSTEM

TOOL

Token Generator

Use the generator's eleven-format export to model what your target token set will look like before migration starts.

designsystems.one/tools/token-generator

FOUNDATION

Maturity Model

Where you are vs where you're migrating to — calibrate the audit against the maturity stages.

designsystems.one/foundations/maturity-model

TURN THE PAGE · TWO WORKSHEETS

Worksheet 1A — Component inventory
One row per component, filled during the walk.

Worksheet 1B — Token collapse and cost score
Top: the colour and spacing collapse.

This chapter, free and linkable: designsystems.one/playbook/migrate/audit-the-surface-area

WORKSHEET

Component inventory

One row per component, filled during the walk. Do not skip the 'resolves to' column: the gap between what the import says and what actually renders is where the duplicate implementations hide.

COMPONENT	RESOLVES TO (PATH)	VARIANTS	INSTANCES	OWNER	API COMPAT

WORKSHEET

Token collapse and cost score

Top: the colour and spacing collapse. Every retired value here becomes an alias row in the chapter 02 bridge. Bottom: the score, multiplied, and the unrecorded constraints that beat the score.

VALUE KEPT	VALUES RETIRED (LIST)	SEMANTIC NAME	COUNT

COMPONENT	COMPLEXITY 1–3	INSTANCES L/M/H	API COMPAT	SCORE (×)	CONSTRAINT THAT OVERRIDES

END OF SAMPLE · CHAPTER 01 OF 5

The other four chapters, and their eight worksheets.

You have just read chapter 01 in full — the same typesetting, figures and worksheets as the rest of the file. Here is what the remaining 30 pages cover.

02 **Build a token bridge**

The intermediate layer that lets old and new ship side-by-side without breaking either. Two worksheets: the alias map, and a retirement contract you sign and date.

03 **Componentize the right things first**

Why order decides the timeline: high-volume primitives unlock the rest. Two worksheets: the wave planner, and a matrix you plot your own components on.

04 **The engineering migration path**

Codemods, escape hatches, and the discipline that keeps a migration from blocking product work. Two worksheets: the sweep runbook, and an escape-hatch register.

05 **Adopt and enforce**

Turning a migration that is 70% done into one that is 100% done, and stays there. Two worksheets: the enforcement ladder, and the cutover checklist.

The full book is 41 pages, with 10 worksheets.

Every chapter is also readable free on the site. What the file adds is print typesetting, the eleven figures, the worksheets, and a licence covering your whole team.

designsystems.one/products/migrate-playbook