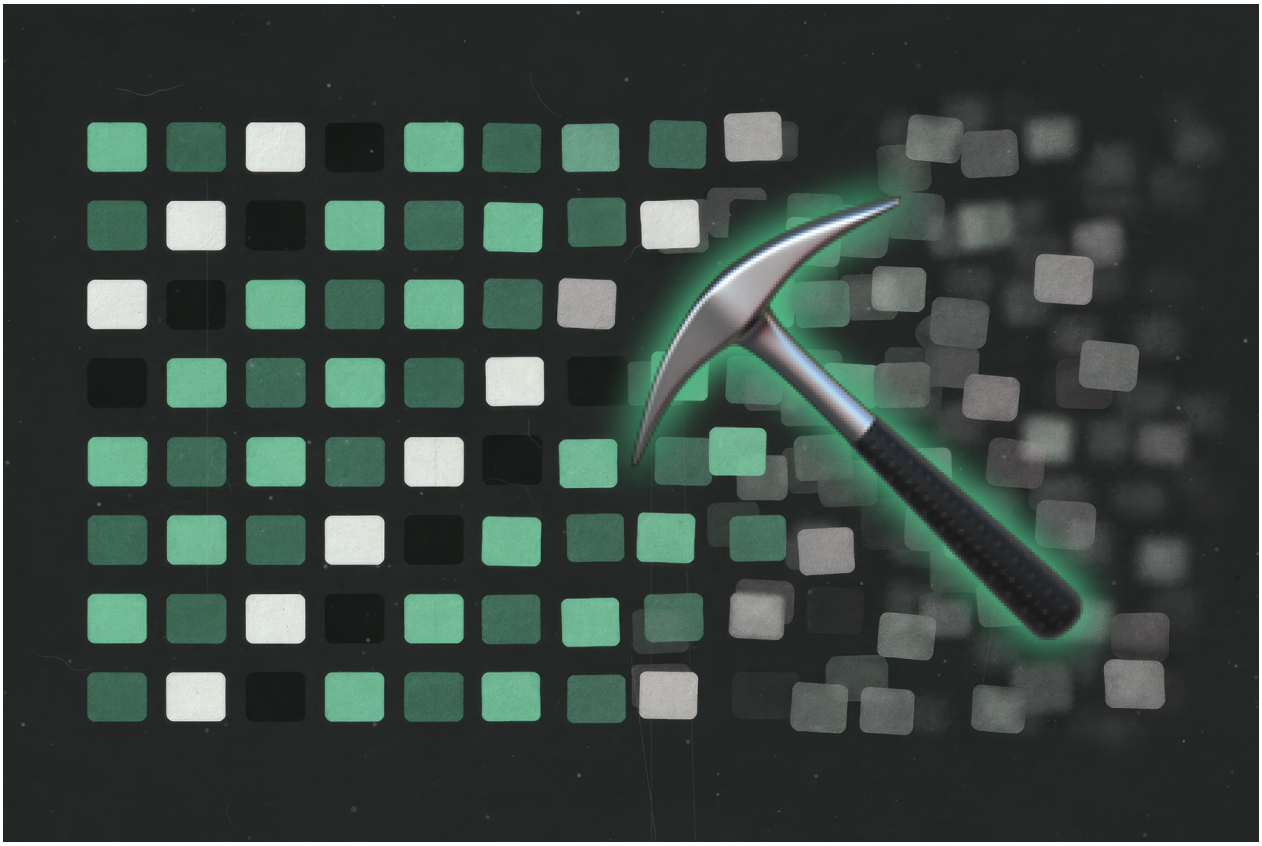
FIELD MANUAL · 2026

Mining the System

How to extract a design system out of a product you have already shipped — structuring it, exporting it to code, documenting it, and making it readable by the machines now building with it.



Lion Blau · Senior Product Designer

lionblau.com

CONTENTS

What's inside

Two books. The first builds the system; the second puts it to work.

Book One — The design system, building it

Part 0 · The mental model — why systems are two layers, not one 3

Part 1 · Extraction — two routes: from production, or from the Figma file 4

Part 2 · Primitives — the raw value layer in Figma variables 10

Part 3 · Semantic tokens — aliasing, naming, and modes 13

Part 4 · Type, spacing, radius, elevation — the non-colour tokens 16

Part 5 · Components — variants, properties, states, auto layout 18

Part 6 · Patterns — composing components into product-level pieces 20

Part 7 · File architecture and publishing the library 21

Part 8 · Export to code — Tokens Studio, Style Dictionary, Code Connect 22

Part 9 · Documentation — per-component docs and accessibility notes 27

Part 10 · Handover — maintenance and contribution guide 28

BOOK TWO

The Applied System — a separate manual, still being written. See the closing page.

How to use this: read Part 0 once, then work Parts 1–10 in order on a real product. The order is the method — every part consumes the output of the one before it. Budget roughly: extraction 2–3 days, tokens 2 days, components 1–2 weeks, publishing and export 2–3 days, documentation ongoing but finalised in 2–3 days.

Every value in this manual (hex codes, type sizes, spacing) is an illustrative example of what an extraction pass produces — harvest the real values yourself using the method in Part 1. Never present guessed values as audited ones.

PART 0

The mental model

A design system is not a component library. It is a decision structure with three floors, and each floor only references the floor directly below it.

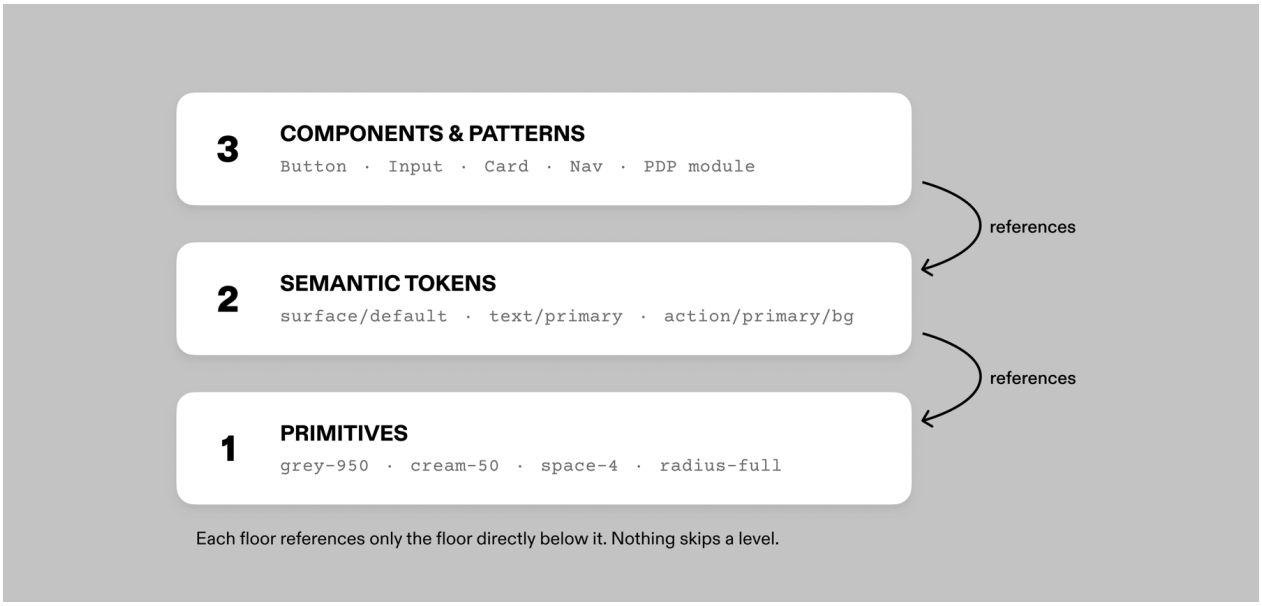


Fig. 1 — The three floors. Build one token layer instead of two and every later change becomes a component-by-component edit.

First principle: aliasing is the whole trick

The single most common failure is building one token layer instead of two. If a button's fill is bound directly to `grey-950`, then a rebrand, a dark mode, or a densification pass means touching every component. If the fill is bound to `action/primary/bg`, which aliases `grey-950`, you repoint one alias and the entire product updates. Everything else in this manual is execution detail.

Second principle: the system encodes decisions, not drawings

"We use exactly four spacing values inside components" is a system. Forty beautifully drawn screens is not. When you extract a system from a finished product, what you are extracting is the decisions its designers made — including the ones they made inconsistently, which you will have to adjudicate.

Third principle: a system is only real once someone else ships with it

Publishing, code export and documentation are not appendices to the design work — they are the difference between a system and a personal component file.

PART 1

Extraction: pulling a system out of what exists

You arrive at a product that already works. Patterns exist across it, they grew unevenly, and nobody owns them end to end. Somewhere in there is a design system nobody has written down, and your job is to mine it out.

The first question decides everything about how you spend the next week: where does the truth live? Sometimes it's the shipped product, because the Figma file drifted years ago or never existed. Sometimes it's a Figma file that is genuinely current and simply never got systematised. These are different jobs, and picking the wrong one wastes days.

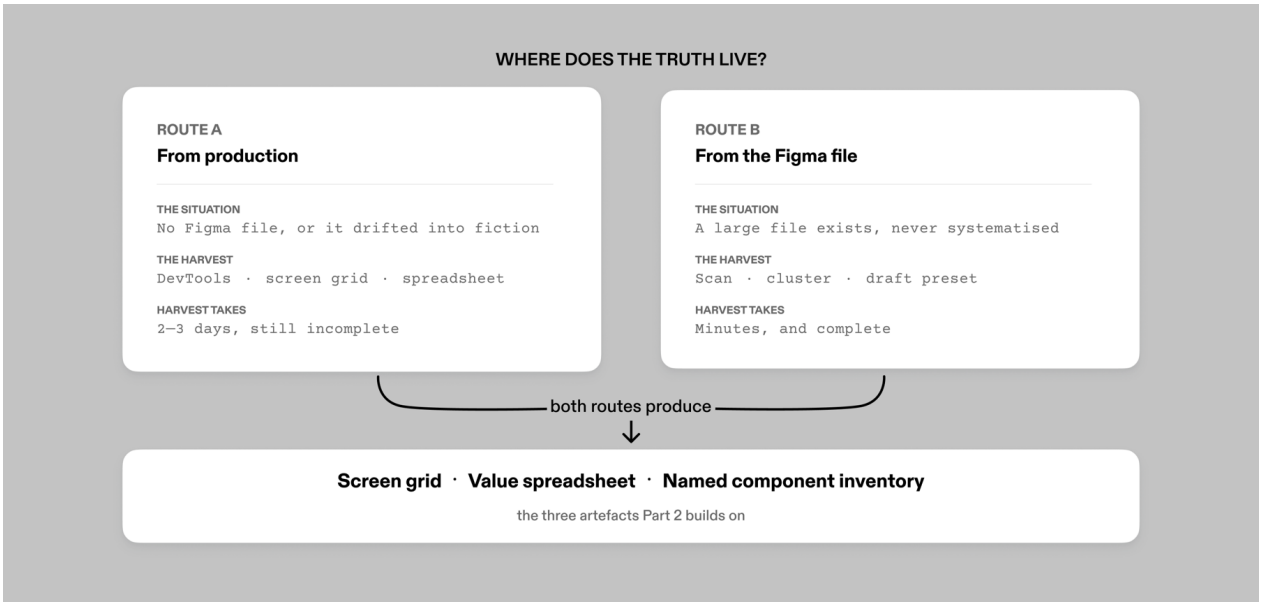


Fig. 2 — The two routes. The brief decides which one you're on; the output is the same either way.

Everything from Part 2 onward is identical regardless of which route you took. Only the harvest differs — and in both cases, the judgement that follows it is the same work and the same value.

ROUTE A

From production

Production is the truth when the design file has drifted or was never maintained. What ships is what users experience, so what ships is what you audit. This is the harder route and the one most contract briefs describe.

STEP 1 — BUILD THE SCREEN INVENTORY

Before touching a single colour, capture the territory. Screenshot every distinct screen and every distinct state you can reach: home, category listing, product detail, cart, checkout steps, account, search (empty, typing, results, no results), error pages,

cookie banners, modals, toasts. On mobile and desktop. Drop them all into one Figma page called `00 Audit / Screens` in a grid, labelled by URL and viewport. For a mid-size product expect 40–80 captures. This page is your evidence base for every later argument about what the system should contain.

STEP 2 — HARVEST THE RAW VALUES FROM PRODUCTION

Open the site in Chrome DevTools. Three places give you the truth fast:

1. **Computed styles.** Inspect representative elements — a heading, body text, a primary button, a card — and record font-family, font-size, line-height, letter-spacing, color, background-color, border-radius, padding.
2. **The CSS itself.** Search the stylesheets (Sources panel, Cmd+Shift+F) for `:root` or `--`. Many modern sites already ship CSS custom properties — if you find `--color-…` variables, half your primitive layer is handed to you, including the team's own naming.
3. **The eyedropper** for anything rendered in images or canvas.

Log everything into a spreadsheet with four columns: raw value, where seen, how often, candidate token name. Frequency matters — it is how you will separate pattern from accident.

Value	Where seen	Freq	Candidate
`#0B0B0B`	page bg, nav bg, footer	~all	`grey-950`
`#FFFFFF`	headings, body on dark	~all	`white`
`#B9B9B9`	secondary text, captions	high	`grey-400`
`#473BCE`	focus ring on careers embed	low	third-party — exclude
`9999px`	button corners (pill)	high	`radius-full`
`1px #2A2A2A`	card hairlines, dividers	high	`border-subtle`

Note the fourth row: extraction always surfaces values that belong to embedded third-party tools, A/B tests, or plain mistakes. Flag them, don't tokenise them. Values above are illustrative — harvest your own.

STEP 3 — CLUSTER AND ADJUDICATE

Real products never have 12 greys on purpose. You will find `#0B0B0B`, `#0D0D0D` and `#101010` doing the same job. Sort every colour by luminance, every font-size and spacing value ascending, and cluster values that are within perceptual noise of each other. Then make the call the role is actually paying for: pattern or one-off? A useful rule of thumb — if a value appears on 3+ screens in the same role, it's a

pattern and becomes a primitive; if it appears once, it is either a deliberate exception (document it) or drift (normalise it to the nearest primitive and note the change so engineering can reconcile production later).

Do the same for typography (you'll typically land on 6–9 real text styles even if you measured 15), for spacing (cluster to a 4px base grid; most products resolve to one), and for radii (editorial consumer brands usually resolve to something like: none, small card radius, pill).

STEP 4 — NAME THE COMPONENT INVENTORY

Go back over the screen grid with a different eye: circle every repeating UI structure and give it one canonical name. Reach for the industry-standard name before inventing one — Button is so settled that no alternative reads as correct, and the same is true of most of the parts you'll find. Keep names to one or two words; use slashes when you need to go narrower (`card/product`). Where two teams built two versions of the same thing (they will have), pick the survivor now and record the decision.

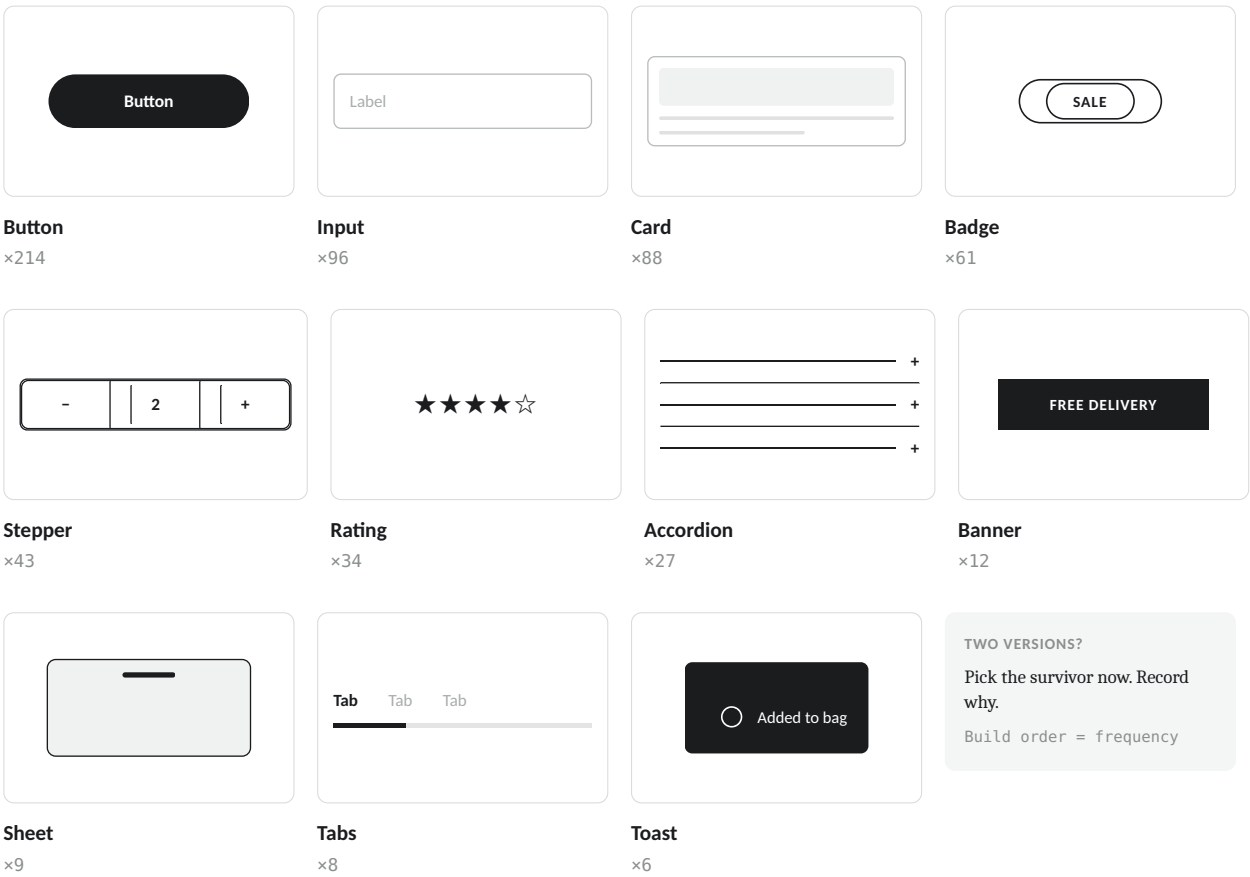


Fig. 3 — A component inventory, ranked. Each entry gets one canonical name and a count of where it appears across the audited screens. Counts are occurrences across the audited screens; names are the industry-standard ones — reach for those before inventing your own.

The output of Part 1 is three artefacts: the screen grid, the value spreadsheet, and a named component list ranked by frequency of use. Build order follows frequency — the parts that appear two hundred times earn their rigour, and the ones that appear twice can wait.

ROUTE B

From the Figma file

Now the opposite brief. There is a Figma file. It is current, engineering builds from it, and it is years of screens deep. It has simply never been systematised. Nobody wrote down which greys are real, nothing is aliased, and every value in it was typed by hand at some point by someone who had a deadline.

This sounds like the easier job. It isn't. It is a different job, and the difference is worth understanding before you start, because the instinct carried over from Route A will fail here.

PRESENCE IS NOT USE

There is a distinction inside this that only became obvious once I tried to automate around it: **matching a value is not the same as being bound to it.** A layer can be metrically identical to an established token — same size, same colour, same everything a script can compare — and still not actually reference it. Nothing forces the two to be connected. They can drift apart later and nothing will tell you.

A tool can find both cases equally well. It cannot decide which one matters to you. Sometimes an unbound match is fine — the value is right, ship it, move on. Sometimes it is exactly the thing you are being paid to fix, because unbound-but-matching today is one rebrand away from unbound-and-wrong. That call depends on how the team actually works, how disciplined the next six months are likely to be, and how much you trust the file to stay still. No amount of scanning answers it. Only someone who knows the team does.

How long this actually takes

It would be convenient to say Route B takes an afternoon. It doesn't, and the honest answer is more useful: the two halves of the work scale completely differently.

	Route A	Route B
Harvest	2–3 days of inspection, and still incomplete	Minutes, and complete
Adjudication	Scales with how many distinct values you found	Scales with how many distinct values you found
What decides it	How many screens you have to cover	How undisciplined the file was allowed to get

So the mechanical half collapses from days to minutes, and the judgement half does not move at all. A file that was loosely maintained by two designers might take a day to adjudicate. A file with six hundred colours accumulated across four years and eleven contributors is a week, and no plugin will change that — it will only make sure you are deciding against the complete set rather than a sample of it.

That is the honest pitch for tooling in this work. It does not make the job shorter. It makes the part that was unreliable reliable, and hands the time it saves to the part that was always the point.

When both sources exist and disagree

There is a third case, and it is the most interesting one. Sometimes you have a maintained Figma file *and* a shipped product, and they don't match. The file says `#0B0B0B`; production ships `#0D0D0D`. The file has an 8px radius; the build rounds to 10.

Run both routes and diff them. That gap is not noise to be quietly resolved — it is a measurement of how far design and engineering have drifted apart, and it is usually the most persuasive artefact you can put in front of a client in week one. It also answers something they often cannot articulate themselves: whether their problem is that the system is undocumented, or that it is documented and ignored. Those need completely different fixes, and the diff tells you which one you are being paid to solve.

The tools

Quarry is free, runs entirely inside Figma with no network access, deletes nothing, and reverses with one undo. Five tabs, in working order:

Tab	What it does	Used in
Scan	Every native colour in the file — fills, mixed text ranges, strokes, gradient stops, shadows, bound variables — with node type, layer path and usage counts. Read-only.	Route B
Import	Creates collections, variables and aliases from a token JSON in one validated pass. Idempotent.	Parts 2–3
Colour	Binds paints on screens you already built to your semantic variables.	Part 5
Dimensions	Finds the 1.75px stroke and the 10.5px radius; offers the nearest step on your real scale.	Part 4
Type	Groups text into recipes, builds the Text Style scale, and applies it. Has an apply-only mode for enforcing a settled scale.	Parts 4–5

None of it decides anything, and that is the design constraint rather than a limitation. Automate the harvest, never the adjudication: lionblau.com/mining-the-system

Quarry is live on the Figma Community — install it from figma.com/community/plugin/quarry, free, no account beyond the Figma one you already have.

One story from building it, because it is a cleaner argument than anything I could write abstractly. An early version of the Type tool would fail an entire batch — eleven good styles, not just the bad one — the moment it hit a single font that was not installed on the machine running it. The fix was mechanical: catch the failure, skip that one style, keep going. But notice what the tool still cannot do even fixed. It cannot tell you *why* the font is missing, whether that is a real problem (someone shipped a typeface no one else has) or a trivial one (it is only missing on this laptop), or what to do about it. A person glances at the message and knows in half a second which one it is. The machine will dutifully skip the same style forever unless a person tells it what happened.

That is the shape of almost every real failure in this work. The tool can be made resilient to the ones you anticipated. It cannot anticipate the next one, and it should not pretend to. What you are actually paying a contractor for is not the scan, the import, or the bind — those are increasingly free. You are paying for the person who looks at the thing that broke and knows, immediately, whether it matters.

PART 2

Primitives: the raw value layer

Primitives are meaning-free: they say what a value *is*, never what it is *for*. Nothing in your UI will ever reference them directly.

In Figma they live in a variable collection, and the path to one is:

```
1  Open the file.
2  Click VARIABLES in the left navigation rail.
3  Create a collection — name it Primitives.
4  Click + Create variable.
5  Pick the type:
       Color     brand and interface colours
       Number    spacing, radius, stroke width, font size
       String    font families, labels
       Boolean   true/false states
6  Name it, and enter the value.
```

Two things worth knowing before you start. Slashes create folders — `space/4` and `space/8` group themselves under `space` automatically. And a collection can hold every type at once, so colour and number primitives can share one collection; splitting numbers into their own (a *Dimensions* collection, say) is equally valid and often tidier on a large file.

Colour primitives

Create colour variables from your adjudicated clusters, on numeric ramps. Steps of 50–950 (Tailwind convention) keep the code export trivial later. For a dark editorial brand the ramp is grey-heavy with one or two accent ramps. Do it by hand for the judgement, or hand the adjudicated list to Quarry's Import tab as a token JSON and let it create every primitive in one pass — deciding which values survive is still yours either way.

grey/0	grey/100	grey/300	grey/400	grey/700	grey/800	grey/900	grey/950
#FFFFFF	#EAEAEA	#C9C9C9	#B9B9B9	#3A3A3A	#2A2A2A	#161616	#0B0B0B

Fig. 4 — A grey ramp as primitives. Numeric steps keep the code export trivial and let engineers predict a name they haven't seen. Primitives carry no meaning — only position on a ramp.

Number primitives

Spacing on the 4px grid (`space/1` = 4 … `space/12` = 48, plus what your audit demands), radii (`radius/none` 0, `radius/sm` 8, `radius/full` 9999), and border widths (1, 2). Create these as Number variables, grouped with slashes — or let Quarry's Dimensions tab build the whole scale for you in one pass and skip straight to naming it correctly.

Naming rules that save you later

> Lowercase, slash-separated, no spaces, no abbreviations you'd have to explain. The variable name becomes the JSON key becomes the CSS custom property becomes the Tailwind key — every awkward name you accept here is a rename across four tools later. Agree the convention with the engineers *before* creating fifty variables, not after: if production already uses `--color-grey-950`, mirror it exactly.

Once the token file is written, Quarry's Import tab creates every collection, variable and alias in one validated pass. It checks the whole file before writing anything, reuses collections by exact name, and never creates duplicates — so it is safe to re-run as the token file evolves. Doing this by hand for two hundred variables is an afternoon you will not enjoy.

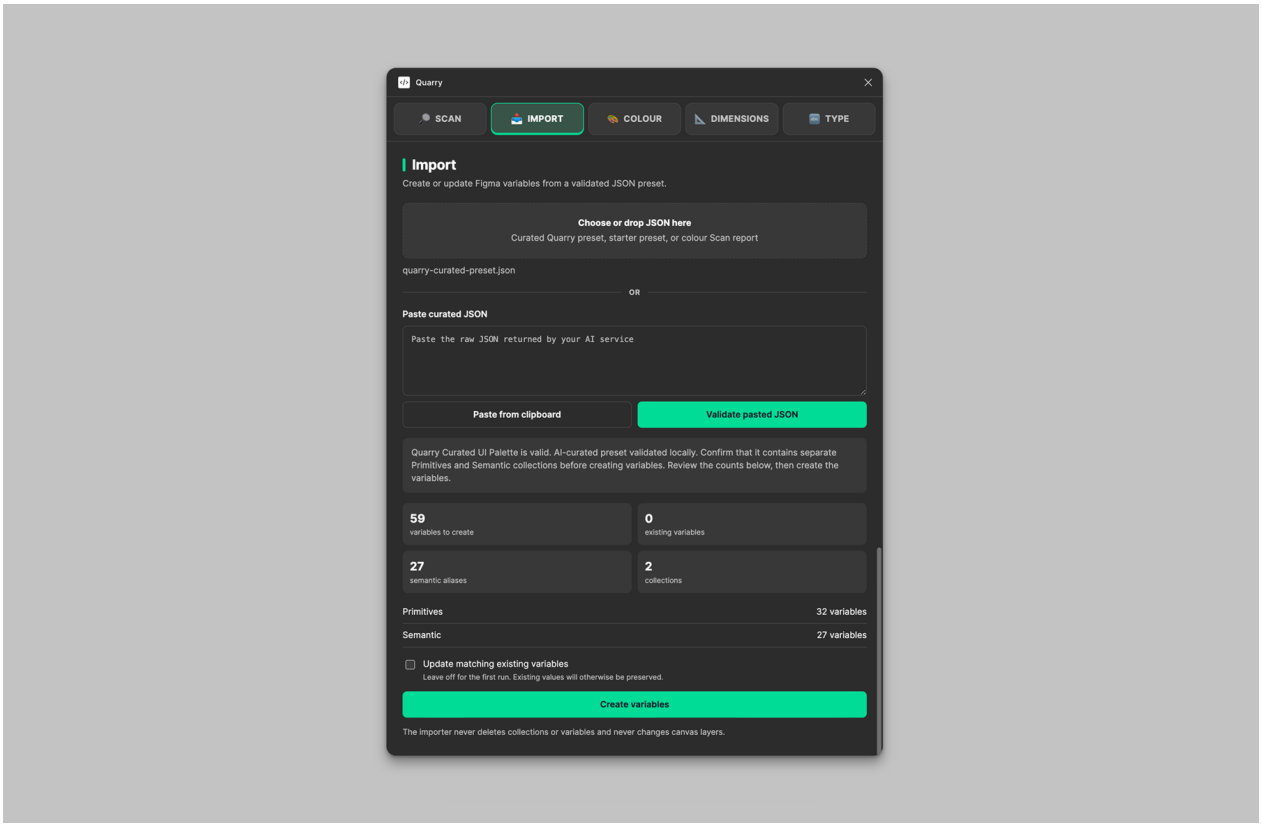


Fig. 5 — Quarry, Figma plugin. Import, after validating a curated preset. 59 variables, 27 aliases, split across Primitives and Semantic — read before it writes anything.

PART 3

Semantic tokens: aliasing and modes

Create a second collection named **Tokens**. Every variable here is created the same way — but when you set its value, you do not type a hex.

You click the value swatch and choose **another variable** (Figma calls this an alias; the panel shows a pill with the source name). Semantic tokens name the *role*. This is the other half of what Quarry's Import tab builds from a token JSON in the same pass as the primitives — 27 aliases alongside 32 primitives is a normal split, and it never guesses which primitive a role should point to; that mapping comes from the file you wrote:

Token	Aliases (Default mode)	Used for
`surface/default`	`grey/950`	Page and section backgrounds
`surface/raised`	`grey/900`	Cards, sheets, dropdowns
`text/primary`	`grey/0`	Headings, body
`text/secondary`	`grey/400`	Captions, meta, helper text
`border/subtle`	`grey/800`	Hairlines, card strokes, dividers
`action/primary/bg`	`grey/0`	Primary button fill
`action/primary/text`	`grey/950`	Primary button label
`action/primary/bg-hover`	`grey/100`	Primary button hover fill
`focus/ring`	`grey/0`	Focus outlines (2px, offset 2)

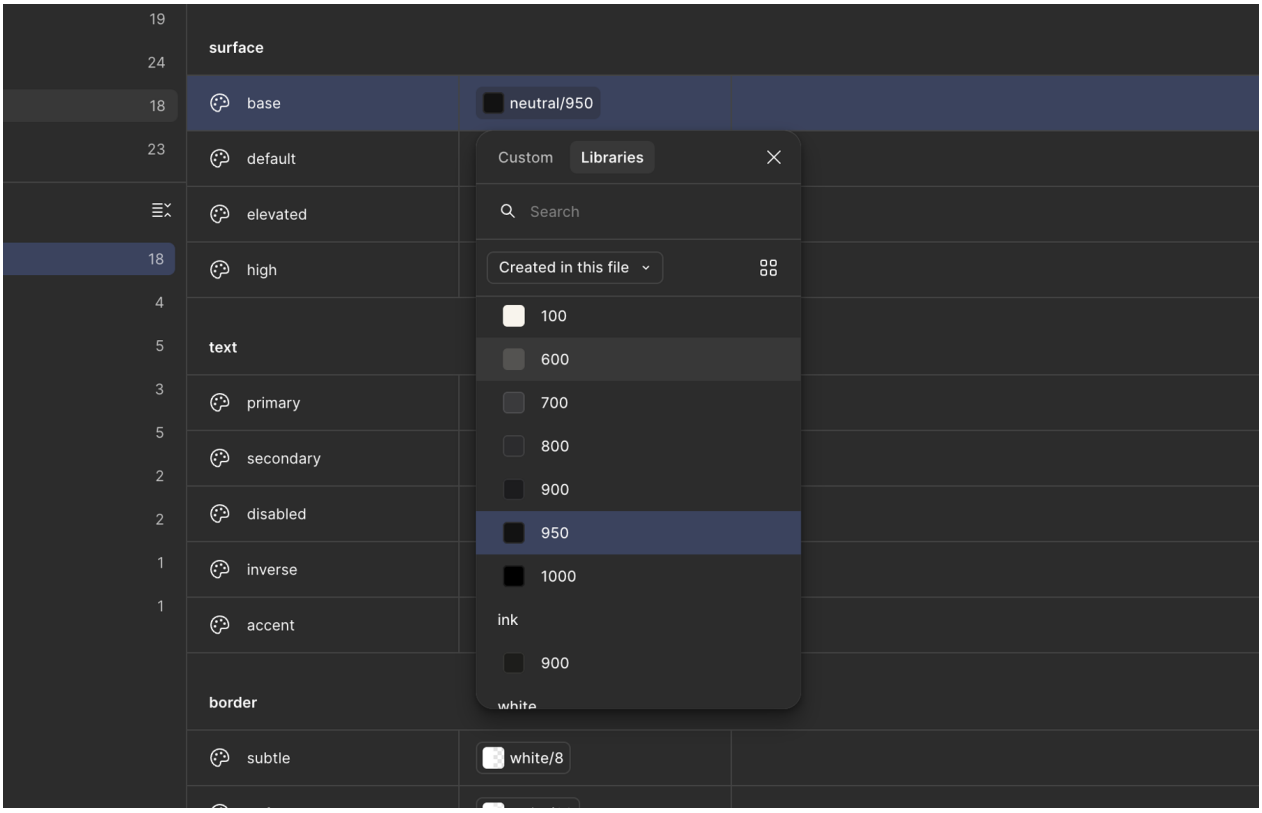


Fig. 6 — Native Figma, not Quarry. The Semantic collection's own value picker. `surface/base` already resolved to a pill, `neutral/950`, instead of a raw swatch — that pill is the alias.

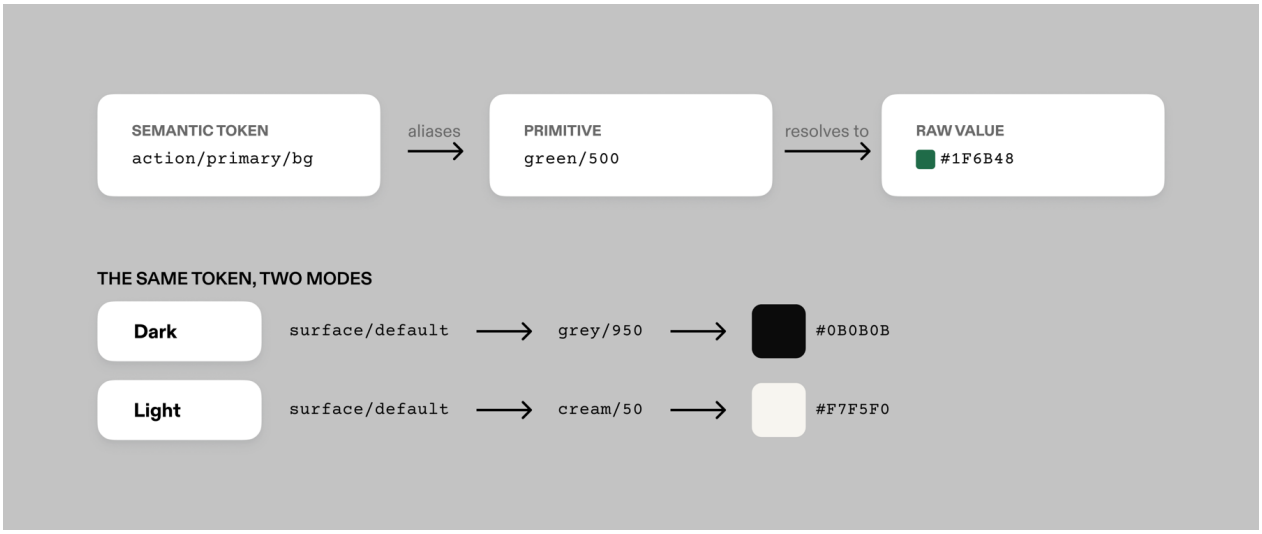


Fig. 7 — Aliasing, and what modes do to it. The component binds to the leftmost name only; everything to its right can change without touching the component.

Modes

In the Tokens collection header, add a mode (the "+" next to the mode column; this requires a paid Figma plan). Name the columns **Dark** and **Light**. Each semantic token now holds one alias per mode: `surface/default` is `grey/950` in Dark and `cream/50` in Light; `text/primary` flips from `grey/0` to `grey/950`. Primitives never get modes — only semantics do. Then select any frame, and in the right

panel's variable-mode control set it to Light: everything bound to semantic tokens flips at once. That moment is your proof the aliasing is wired correctly — do it as a test with two tokens before building out all of them.

Modes are not just themes. The same mechanism handles density (comfortable/compact spacing modes on a spacing-token collection) and breakpoints (type-scale modes per viewport). One mechanism, three of the hardest product problems.

PART 4

Type, spacing, radius, elevation

Typography

Turn your audit's 6–9 real styles into a named scale: `display/xl` … `body/md` … `caption`. Build each one as a Figma **text style** so it applies in one click, but store the actual font-size and line-height as number primitives underneath it, not typed directly into the style. That's what lets the same values export to code and support breakpoint modes later — the text style alone can't do either.

Capture letter-spacing in the same scale, not as an afterthought. Headings usually want to track tighter (-1 to -2%); small caps and labels usually want to track wider (+4 to +8%). Get this wrong and every heading in the file looks slightly off in a way nobody can name.

Quarry's Type tab automates the mechanical half of this: it groups your file's actual text into recipes, proposes a name for each, and creates the Text Style plus its backing variables once you approve it — which sizes make the scale is still a call only you can make.

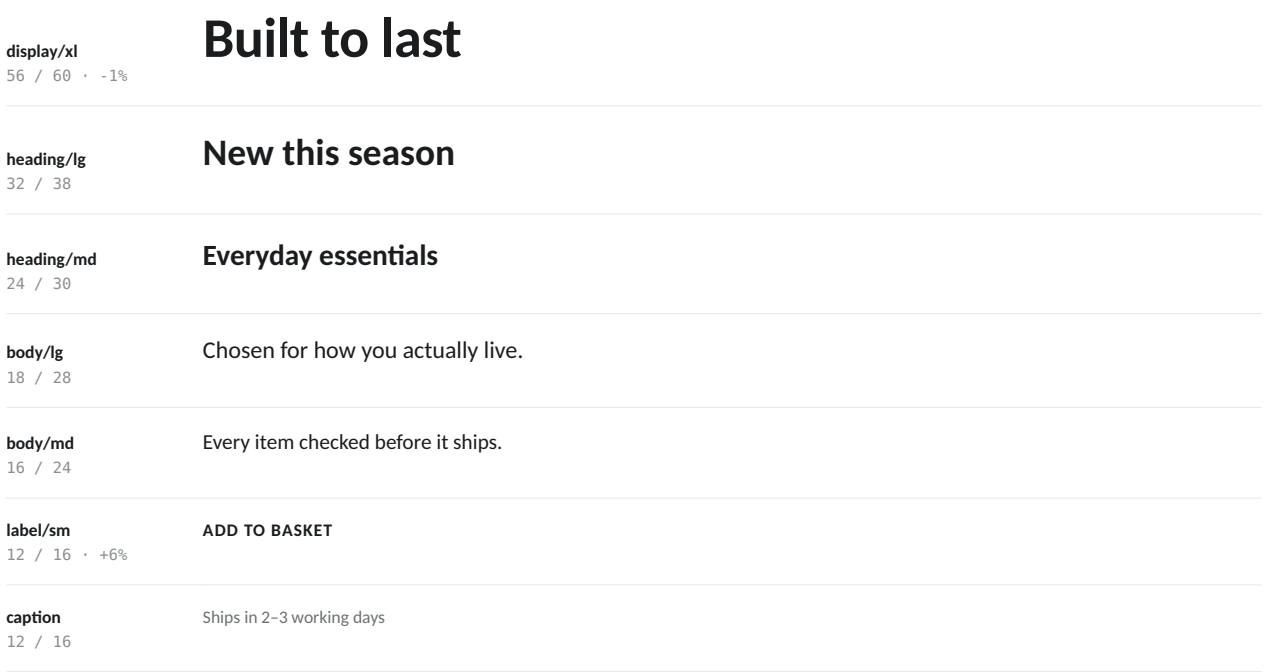


Fig. 8 — A type scale as a specimen sheet. Six to nine styles is almost always the real answer, however many you measured. Sizes shown proportionally reduced.

Spacing in practice

Bind auto-layout gaps and padding to spacing variables (click the field → variable icon). Adopt one discipline: components use only `space/1`–`space/6` internally; page-level rhythm uses `space/8` and up. This single rule is most of what makes

screens built by different people look related. Quarry's Dimensions tab can do this binding pass across a whole selection at once, on values that already match your scale exactly — the off-scale cases below are the ones worth your own attention.

Legacy files accumulate values that sit off the scale — a 1.75px stroke from a resize, a 10.5px radius nobody chose. Quarry's Dimensions tab finds them and offers the nearest step from your real collection, sorting the ones needing a decision above the exact matches. It reads your scale rather than inventing one.

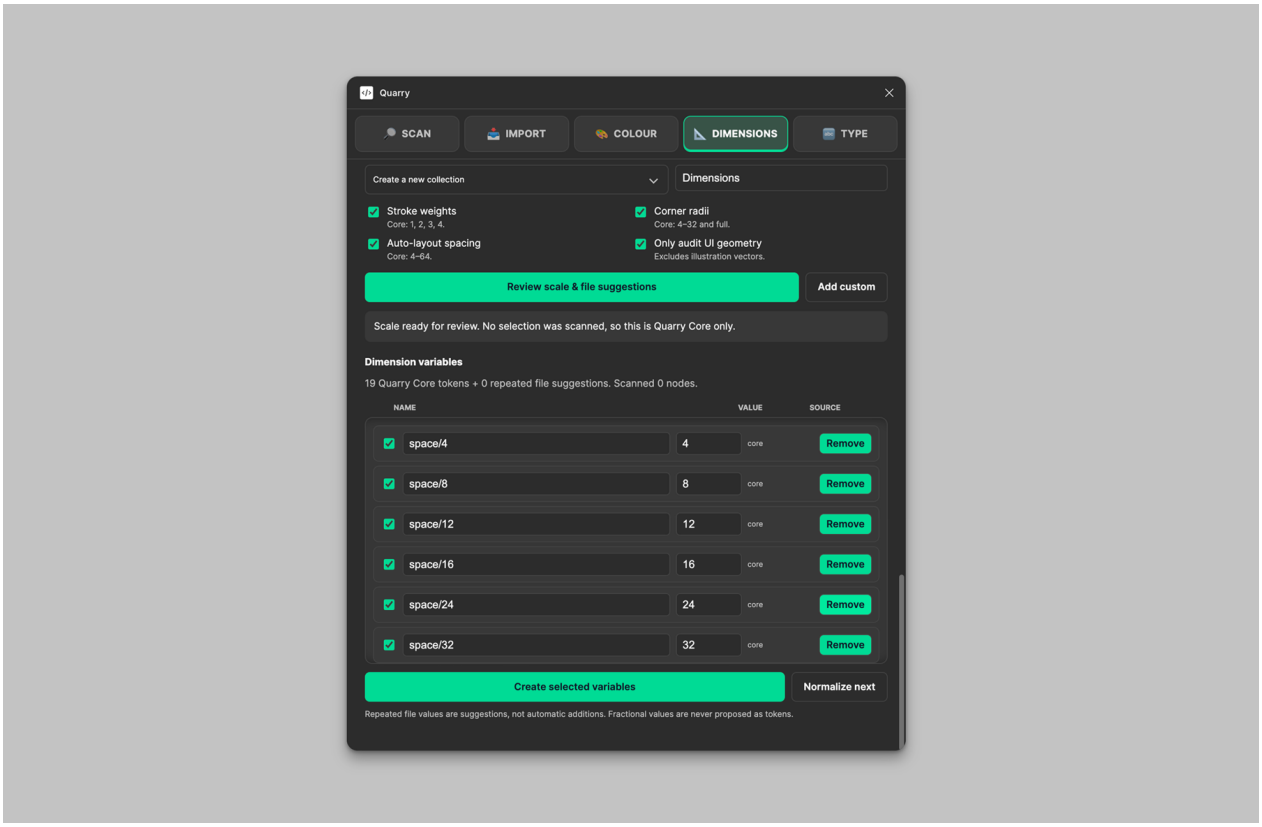


Fig. 9 — Quarry, Figma plugin. Dimensions, building the base scale before anything is normalized against it. Nineteen core tokens, none of it guessed.

Radius and elevation

Bind radii to the radius primitives — the same Dimensions pass above handles this alongside spacing and stroke. For elevation on a dark UI, shadows barely read — dark systems usually encode elevation as *surface steps* (raised surfaces use a lighter surface token) plus a subtle border, rather than drop shadows. Encode whichever your audit found as effect styles or documented conventions.

PART 5

Components

Build order comes from the ranked component inventory you built in Part 1, Step 4 (Fig. 3) — start with whatever appeared most often in your audit, not whatever feels most important.

A commerce-heavy product usually resolves to something like Button → Input → Product Card → Price → Badge → Nav → Accordion → Modal/Sheet → Toast. A content or tools product will rank differently — that's expected. The list is supposed to be your product's, not a template. For every component, the same five disciplines apply:

1 · Auto layout everywhere. Every component is an auto-layout frame; padding and gap bound to spacing variables; min/max widths set where the design has opinions. Test each component by stuffing it with a two-line German product name — if it breaks, production would have broken it.

2 · Variants for mutually exclusive appearance. A Button gets variant properties *variant* (primary / secondary / ghost) and *size* (sm / md / lg). Name variant properties exactly as the code names its props — this is what makes Code Connect mapping and Dev Mode handoff coherent later.

3 · Component properties for everything else. Boolean props for optional elements (*hasIcon*, *hasBadge*), instance-swap for the icon slot, text props for the label. The test: if you multiplied it out as variants would you get a 200-cell matrix? Then it should have been a property. Variants × properties should stay small enough to read in one screen.

4 · The full state set. Default, hover, focus, active, disabled — and for anything that carries data: error, empty, loading. States are a *state* variant property, and interactive states get wired as prototype interactions on the component set (While hovering → change to, Mouse down → change to) so every instance inherits the behaviour. Empty and loading are the states teams forget and engineers improvise; designing them is cheap insurance.

Fig. 10 — The state set, each bound to its own token. Six states off one ramp; the steps between default, hover and active are deliberately small — large enough to register, small enough to stay one brand.

5 · Zero raw values. Before marking a component done, select it and check the right panel: every fill, stroke, radius, gap and padding shows a token pill, not a hex or a number. Figma's "selection colours" section exposes stragglers instantly — anything listed as a raw hex there is unfinished work.

Screens built before the system existed can be brought into it rather than rebuilt. Quarry's Colour tab binds existing paints to your semantic variables, and the Type tab groups text into recipes and applies the Text Style scale. Both preview everything before writing, skip instance internals by default, and reverse with one undo. Neither invents a token you did not create.

COMPONENT CHECKLIST, PER ITEM

[ ] Tokens only — no raw hex or number values anywhere
[ ] Every state built, including empty and loading
[ ] Auto layout resilient to long content (tested with a two-line string)
[ ] Properties named exactly like the code's props
[ ] Description field filled — one line: what it's for, plus a link to its doc page

PART 6

Patterns: product-level composition

Components answer "what does a button look like"; patterns answer "how does this product actually sell what it's selling."

A pattern is a documented composition — several components plus fixed spacing between them, repeated wherever that same job comes up. On a commerce product that's usually a product card; on a content product it might be an article preview; on a tools product it might be a result row. Whatever it is for you, the same move applies: break it into its parts, name the token each part uses, and write down why it's assembled that way. Below is a commerce example, worked in full, since a real one is more useful than a description of one — the method transfers regardless of what you're building.

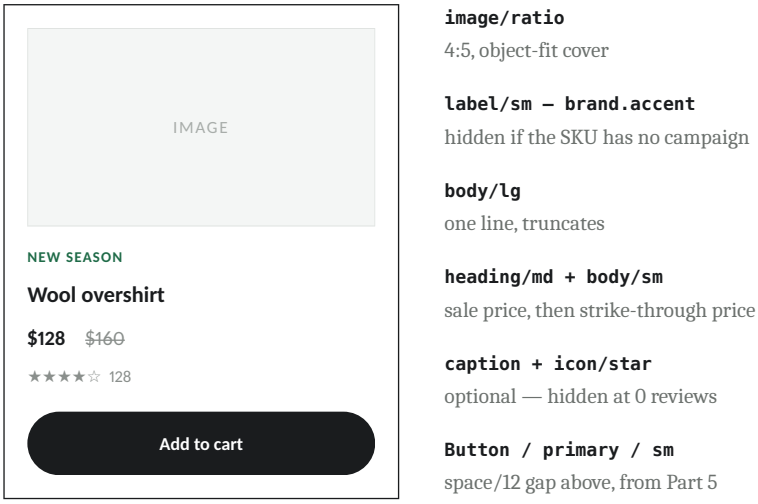


Fig. 11 — A pattern, broken into its parts. Each label names the token that part actually uses, the same discipline as a single component in Part 5.

Build patterns as components too where they repeat verbatim (cards, PDP buy-box, cart line item), but document them even when they stay as guidance (page grids, form layout rules, when a modal vs a bottom sheet).

Patterns are where extraction judgement shows up most: on the audited product you'll find the price row rendered three slightly different ways. Pick one, write down why, list the screens that must change. That written decision log is deliverable material — it is the visible evidence of the "what should the system enforce vs leave open" thinking.

PART 7

File architecture and publishing

Structure the library as its own Figma file (or files), separate from product design files.

```
[Team] Design System

File: DS · Foundations    pages: Cover / Primitives / Tokens /
                           Type / Iconography / Decision log

File: DS · Components     pages: Cover / Buttons / Forms / Cards /
                           Navigation / Overlays / Feedback /
                           _Sandbox (never published)

File: DS · Docs (optional) if docs live in Figma rather than Notion
```

Product files subscribe to the library; nobody designs inside it.

One practical note if you built this system the way this manual describes — with Quarry, inside the product file that was already live, rather than a blank foundations file. That's the correct call for this kind of work, not a shortcut. Quarry's binding tools only reach layers in the file they're running in, and Figma still has no reliable way to move a variable collection to another file while keeping what's already bound to it — moving or copying breaks the reference; only publishing preserves it. So don't move anything. Publish the product file's own collections as a library, the exact step below, and other files can consume them with every existing binding untouched. Save an actual separate DS file for the next system you build from nothing — that's where this structure earns its keep. For a retrofit, the file you already migrated is correctly the file of record.

Publishing: Assets panel → library icon → Publish on the system file (team libraries require a paid plan). Write a change note every time — one sentence, imperative: "Add Toast; rename `input/border` to `border/interactive`."

Consumers enable the library in their file's Assets panel and receive components and variables. From then on, edits you publish arrive in product files as an update review — designers see a diff and accept. That review step is your safety net: it means you can evolve the system without silently repainting someone's in-flight work.

Hygiene that separates professional libraries from hobby ones: a `_Sandbox` page prefixed with an underscore or dot so drafts never publish; deprecations handled by renaming to `[deprecated] Old Button` and pointing its description to the replacement rather than deleting (deletion breaks live files); one release note thread (Slack/Notion) where every publish is announced.

PART 8

Export to code

Right now your tokens are real inside Figma, but Figma is not where the product actually runs.

Someone still has to write `#0B0B0B` — or `surface/default` — into a stylesheet. If that happens by a person reading your Figma file and typing what they see, you have two sources of truth again, and they will drift the first time either one changes without the other. Exporting to code closes that gap: engineers write against real named values instead of copying hexes by eye, and any AI coding tool generating UI for you has an actual file to read instead of a guess. This part is mostly not your hands-on-keyboard work — it's engineering territory from Step 3 onward — but you need to understand the shape of it to hand it off correctly and know what to ask for.

There are two depths worth knowing. The **full pipeline** below is the standard for a team with its own build process: tokens flow through a code repository and a dedicated transform step, reviewed like any other code change. The **quick export**, a few paragraphs down, is Tokens Studio doing that transform itself, no repository or build step required — good enough to get real CSS in your hands today, before engineering has set anything up.

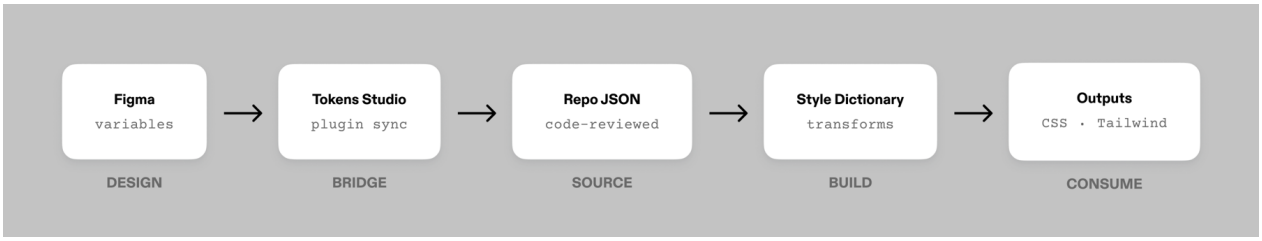


Fig. 12 — The full pipeline. Five stages, left to right: design, bridge, source, build, consume. Your two-layer architecture survives the whole way across — aliases are preserved in the JSON, not flattened to raw values.

Step 1 — Figma (design)

This is everything before this Part: primitives, semantic aliases, modes, tokens bound to real components. Nothing new to do here — Step 2 reads what you already built.

Step 2 — Tokens Studio (bridge)

Two parts work together here, not one. Inside Figma, the **Tokens Studio plugin** is what actually reads your file — **Tools → Import variables** pulls in everything you built with Quarry, every collection and every alias, as token sets. Connect that plugin to a **Tokens Studio account** (the web app at tokens.studio) and it syncs

there continuously in the background — no manual push, no re-import when you change something in Figma. From that point on, the web app is where you actually work.

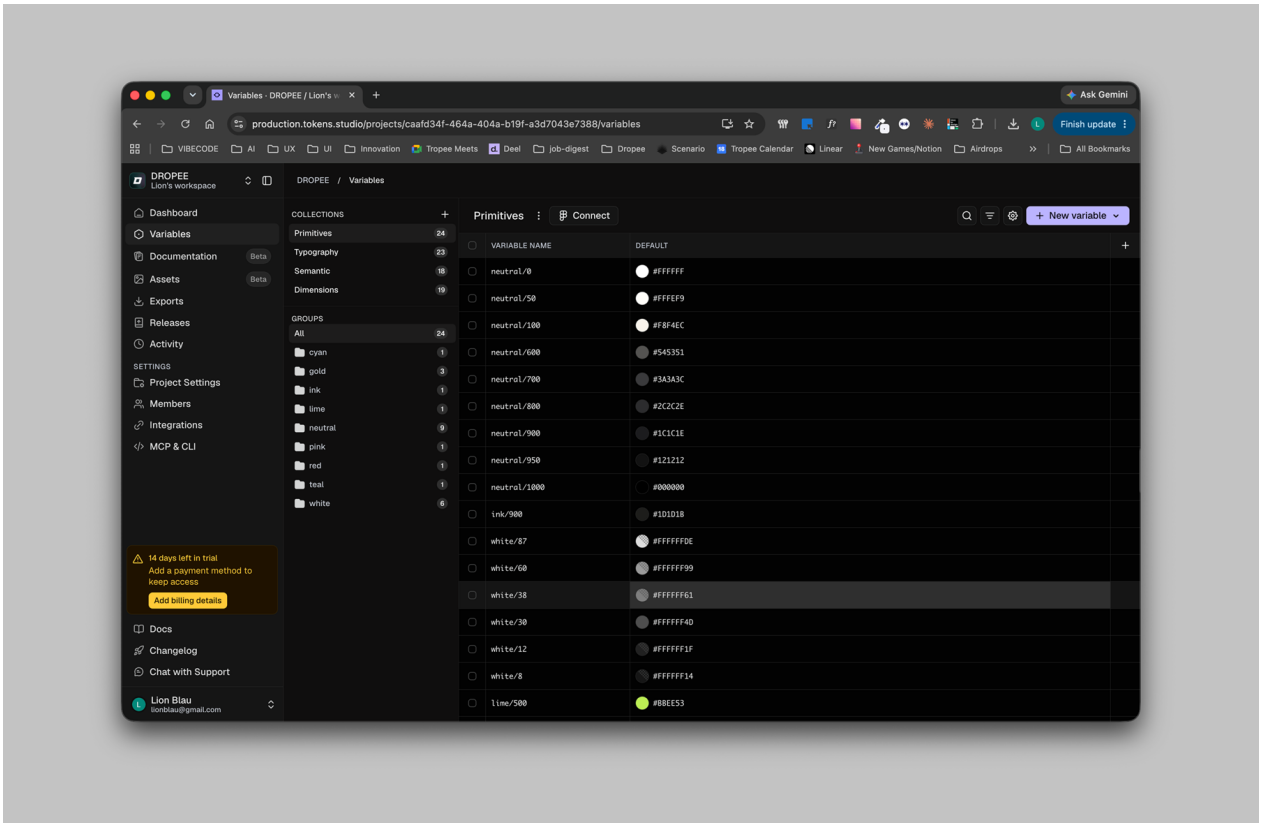


Fig. 13 — The Tokens Studio web app, after the plugin's sync. Every collection Quarry built in Figma — Primitives, Typography, Semantic, Dimensions — arrives here as a token set, aliases intact.

From here, your tokens can go one of two ways, and you don't have to choose yet.

Step 3 — Repo JSON (source) — the full-pipeline path

For a team with its own codebase, tokens continue on into a code repository — a Git provider (GitHub, GitLab, and others) connected as a sync target, so the same tokens land as JSON files inside the actual product repo, reviewed like any other code change. The exact screen for this sits under your project's sync settings rather than the Exports page below; if you haven't set up a repo connection yet, this is the step to hand to whoever owns that repo — it's a one-time setup, not something you repeat per export.

```
tokens/primitives.json (excerpt)
{
  "grey":  { "950": { "value": "#0B0B0B", "type": "color" } },
  "space": { "4":   { "value": "16",      "type": "dimension" } }
}

tokens/semantic.json (excerpt)
{
  "surface": { "default": { "value": "{grey.950}", "type": "color" } },
  "text":    { "primary": { "value": "{grey.0}",   "type": "color" } }
}
```

Note the braces: `{grey.950}` is the alias, preserved into code. Your two-layer architecture survives the export intact — this is why the layering mattered.

The quick path — exporting directly, no repo required

Tokens Studio can also hand you finished, usable files right now, without Step 3 at all — this is what its own **Exports** page does, and it's a genuine shortcut, not a preview of something else. Useful the moment you've imported, before any engineering process exists yet.

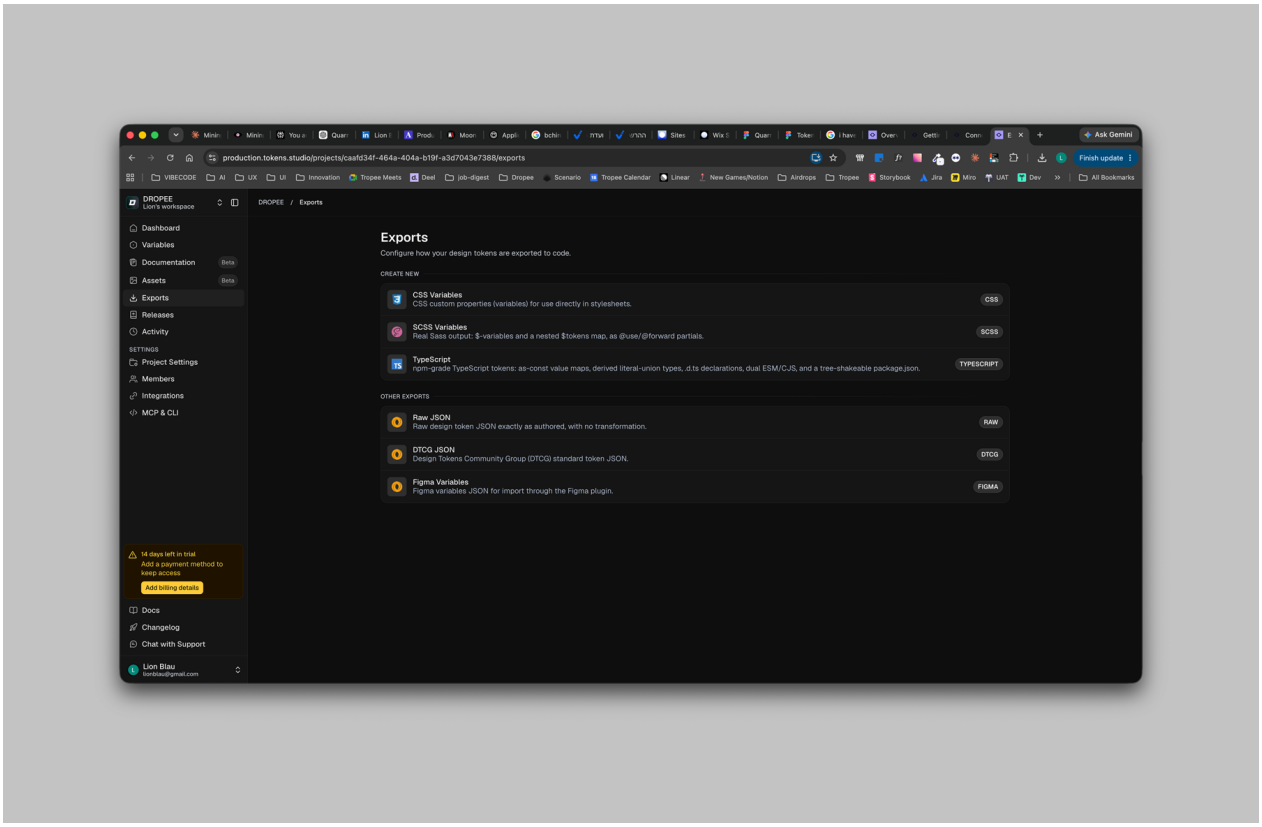


Fig. 14 — Tokens Studio's own Exports page. Each card is a ready file, no build step required.

In plain terms, each option on that page is: **CSS Variables** — a stylesheet defining every token as a real CSS custom property (`--surface-default`), droppable straight into any website's styles. **SCSS Variables** — the same idea, for a Sass-based codebase, which some engineering teams still use instead of plain CSS. **TypeScript** — the tokens as typed JavaScript/TypeScript objects, for a codebase that reads its design values in code rather than a stylesheet, common in React apps. **Raw JSON** and **DTCG JSON** — the tokens exactly as authored, useful only if a downstream tool is going to process them further itself. **Figma Variables** — round-trips back into another Figma file. For a straightforward web product, CSS Variables is usually the one to hand engineering first.

This path is enough for a small team or a first pass. It stops being enough once you need custom transforms per platform, or the export has to run automatically on every change rather than by someone clicking a button — that's exactly the gap Step 4 fills.

Step 4 — Style Dictionary (build) — the full-pipeline path

Style Dictionary is a well-known open-source tool that does the same transform Tokens Studio's Exports page does, but configurable and automated: point it at the repo JSON from Step 3, and it writes CSS, a Tailwind config, TypeScript, iOS or Android resources — however many platforms need outputs, from the same one source. Two unfamiliar words worth knowing, in plain terms: it ships as an **npm package**, which just means it's a small piece of software engineers install as part of their normal toolchain — you never install or run it yourself. And it typically runs in **CI** (continuous integration), meaning it fires automatically every time the token JSON changes, not as a manual step someone remembers or forgets to repeat.

```
// style-dictionary build → outputs
:root {                              /* css/variables.css */
  --surface-default: #0B0B0B;
  --text-primary:    #FFFFFF;
}

// tailwind.config.js (generated theme excerpt)
theme: { colors: { surface: { DEFAULT: "var(--surface-default)" },
                   text:    { primary: "var(--text-primary)" } } }
```

Dark/light modes export as separate CSS scopes (`:root` vs `[data-theme=light]`) generated straight from your Figma modes — nobody hand-writes a second theme file.

Step 5 — Outputs (consume)

This is who actually uses everything above. **Engineers** building the product write `bg-surface text-text-primary` in their components and never type a hex — the token name is the only spelling of that value that exists anywhere. **AI coding**

tools generating or editing UI code read the same CSS or TypeScript file as their source for what a given token means, which is the entire reason this export has to be real and current rather than a one-time snapshot — a stale export teaches the model a colour that's already wrong. Neither audience ever opens Figma to find out what a value is; the export is the interface between your system and everyone who builds from it.

A SEPARATE MECHANISM

Dev Mode and Code Connect are worth knowing about here, but they are not a Step 6 and have nothing to do with the token JSON pipeline specifically. They operate on components, not tokens.

Dev Mode is free behaviour on your side: because everything is token-bound, engineers inspecting a frame see token names, not hexes. Code Connect goes further — it maps a Figma component to the real code component so Dev Mode shows actual production usage (`<Button variant="primary" size="md">`) instead of generated CSS. The mapping lives in the repo and is owned by whoever owns the code side; your contribution as the designer is having named Figma variant properties identically to code props (Part 5), which makes the mapping one-to-one instead of a translation table.

PART 9

Documentation

Documentation is the deliverable that makes the system usable by people who weren't in your head.

Where it lives matters less than that it is one place — a Figma docs file, Notion, Storybook docs pages, or Tokens Studio's own Documentation section, which builds pages from live blocks that stay connected to your actual token values, so they update themselves as the system changes — linked from every component's description field. Per component, the template:

Section	What it contains
Purpose	One sentence. "Primary action of a view. One per view."
When not to use	The mistakes you predict. "Not for navigation — use Link. Not for destructive actions — use Button/danger."
Anatomy	Labelled diagram: container, icon slot, label, spacing tokens between them
Props & variants	Table mirroring code props: variant, size, state, hasIcon…
States	Visual row of all 8 states with the tokens each one uses
Content rules	Label casing, max length, truncation behaviour, i18n notes
Accessibility	Contrast ratios per state, focus behaviour, target size, scaling
Do / Don't	2–4 image pairs from real product screens

Accessibility notes, concretely

Per component, verify and record: text contrast ≥ 4.5:1 (normal) or 3:1 (large text) against its actual surface token *in both modes* — dark themes routinely pass in dark and fail in light; non-text contrast ≥ 3:1 for borders and icons that carry meaning; visible focus (your `focus/ring` token, 2px, offset, contrast-checked); touch targets ≥ 44×44 even when the visual is smaller (pad the hit area); and behaviour at 200% text scaling — auto layout done right in Part 5 is what makes this pass. Run a contrast plugin across the published library as a batch, and record the numbers in the doc rather than the word "passes".

Documentation debt compounds fastest at handover. The rule that keeps it honest: **a component is not "done" until its doc page exists** — done-ness lives in the doc, not the canvas.

PART 10

Handover: maintenance and contribution

If the engagement is finite — a 1–3 month contract, say — the system's survival depends on a short written guide plus a live walkthrough.

The maintenance and contribution guide fits on two pages, and covers five things:

Ownership. Name who approves changes after you leave — a role, not a hope. If nobody is named, nobody will feel responsible, and the system quietly stops being maintained the week you go.

Change process. The path a change actually follows: propose it in the decision log, build it in `_Sandbox`, get it reviewed, publish it with a change note, announce it in the release thread, then let it sync to code through the usual Tokens Studio pull request.

Adding a component. The bar a new component has to clear before it's accepted — used in three or more places, tokens only, every state built, a doc page written. Write down the same bar you held yourself to; don't make the next person guess it.

Deprecating. Rename it, redirect its description to the replacement, never delete it outright — deletion breaks any live file still referencing it.

What's deliberately out of scope. List the one-offs you chose not to systematise, and why. Written down, the next person doesn't have to re-litigate a decision you already made.

Then run the walkthrough live with the design team and the app developer together: build one small real feature using only the library, in front of them, and have one of them do the second one while you watch. That hour transfers more than the document does; the document exists for month three.

WHAT'S NEXT

Book Two — coming separately

Book One ends here, and it is a complete, usable thing on its own: a real design system, extracted, tokenised, componentised, published, exported to code, and documented.

Everything through Part 10 is the whole method for building the artefact.

Book Two is a separate manual, still being written, about what the finished system is *for* once it exists: running it as the substrate for an AI-assisted prototyping loop — where a prompt returns a working screen assembled from your real components, not a picture of one. It covers what to automate confidently versus what has to stay human judgement, and the day-to-day discipline of building that way without the library rotting under you.

If you've made it through Book One, you already have the harder half done. Book Two is shorter, and it will be free the same way this one is.

Lion Blau · Senior Product Designer · lionblau.com/mining-the-system · info@lion-blau.com