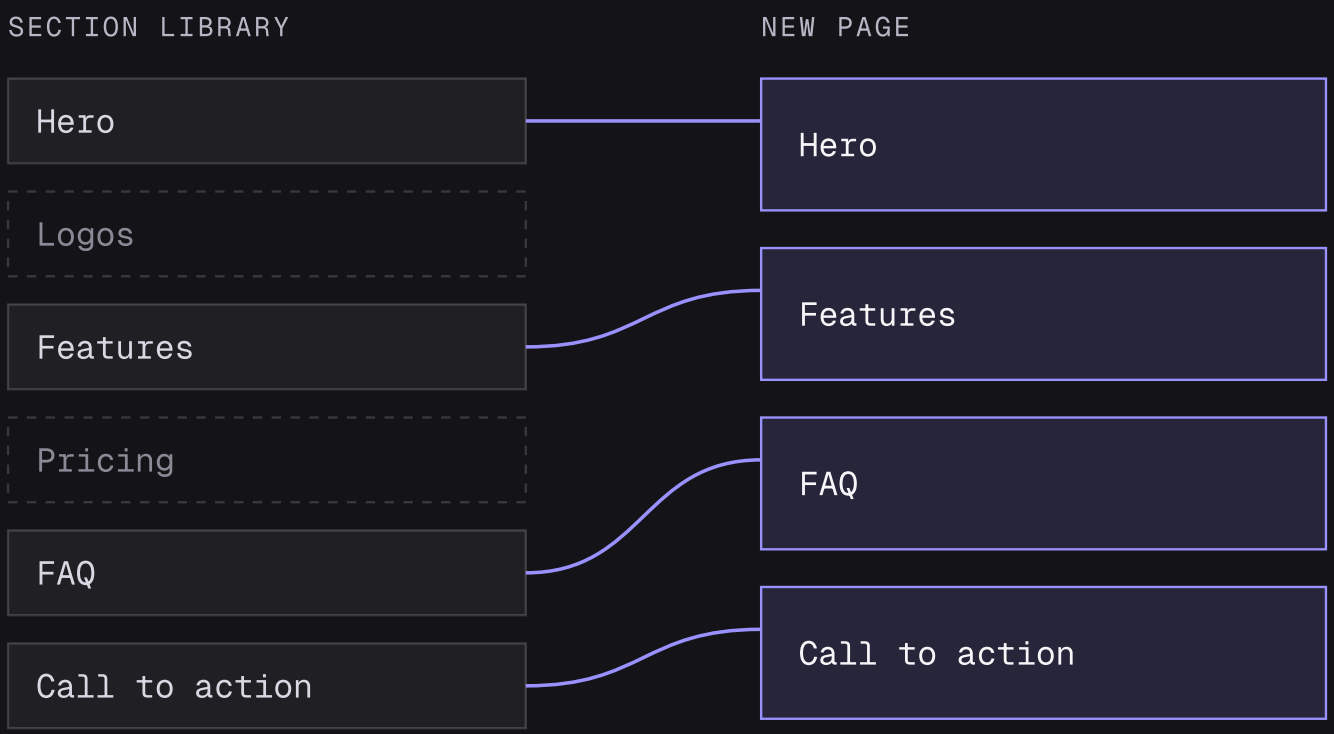
KOOKIES

GUIDE

Section Libraries for Astro and Sanity

6 CHAPTERS



ASTRO + SANITY

Build a section library your team and AI can use.

A section library is a set of reusable page sections, each with its own design, Sanity schema, Astro component and usage note. Editors build pages by stacking them, and an AI tool can draft pages from the same list. To build one, audit your real pages, merge what repeats, model each section with narrow fields and rules, set up the Studio for editors, keep AI to drafts with permissions, and name an owner. This guide covers each step.

Who it is for. Marketing leads, designers and developers who run a marketing site on Astro and Sanity, or who are planning one, and want editors or AI to build pages safely.

01 Why sections, not pages

02 Planning the library

03 Sanity schemas for sections

04 Studio for marketers

05 AI page building with guardrails

06 Testing and governance

Read online at www.kookies.io/guides/section-libraries. Updated 2026-11-07.

CHAPTER 1 OF 6

Why sections, not pages

01

A page is a one-off. A section is a part you can use again, with rules attached. Here is what that changes for your team, and how to tell if you need it.

The short answer

Build from sections when your team makes new pages often and those pages share the same building blocks. A section is one reusable part of a page, such as a hero or a FAQ, with its own fields, a few approved variants and rules. A page becomes an ordered list of those parts. Design and code happen once, per section. New pages then cost mostly writing time.

If you want the plain overview first, read Astro and Sanity for marketing teams. Our section library page shows what we build. This guide is the working manual: how to plan, model, test and run a library on Astro and Sanity, including how AI fits in.

What goes wrong when you build by the page

Most sites start page by page, and that is fine for the first ten pages. The trouble starts when someone needs page eleven by Friday.

- **Pages get copied.** Someone duplicates the closest page and edits it. Each copy now has its own small differences.
- **Fixes do not travel.** A better form or a clearer button goes on one page. The others keep the old one, and nobody can list which ones.
- **Design drifts.** Spacing, heading sizes and button labels slowly stop matching.
- **The queue grows.** Every new layout needs a designer and a developer, so simple pages wait behind product work.
- **Speed slips.** One-off pages bring one-off scripts and images, and no one owns the total.

None of this is anyone's fault. It is what happens when the unit of work is the page. Change the unit, and most of these problems get smaller.

What a section is, exactly

"Section" gets used loosely. In this guide, it means four things that ship together. If one is missing, you have a design idea, not a library item.

PART	WHERE IT LIVES	WHAT IT DOES
Design	Figma or your design file	Shows the section at every screen size, in each variant
Schema	Sanity, defined in code	Sets the fields an editor fills in, and what counts as valid

PART	WHERE IT LIVES	WHAT IT DOES
Component	Astro code	Turns the saved content into HTML and CSS
Usage note	The Studio and a short written guide	Says when to use this section, and when to pick another

The usage note is the part teams skip, and it matters more once AI is involved. An AI tool, like a new hire, can only follow rules someone wrote down.

How a page is stored and built

Here is the whole idea in five steps. Chapter 3 shows the code.

1. A page in Sanity is a document with a few page-level fields, such as title and slug, and one array called `sections`.
2. Each item in that array is an object with its own type, such as `hero` or `faq`. Sanity stores that type name with the content as `_type`.
3. At build time, Astro fetches the page and walks through the array in order.
4. For each item, it looks up the component with the same name and passes in the fields.
5. The result is a normal HTML page. Visitors never touch the CMS.

Because a page is data, reordering is easy, fixes reach every page that uses a section at the next build, and a tool can read exactly what is on each page.

Three ways to run a marketing site

A section library sits between two other models. Each one is right for someone.

	FIXED TEMPLATES	SECTION LIBRARY	FREE CANVAS
How a page is made	Fill in one set layout	Stack approved sections in any sensible order	Place anything anywhere
Editor freedom	Low	Medium, on purpose	High
Risk of off-brand pages	Low	Low	High
Setup work	Low	Higher, up front	Depends on the platform
Best for	Many pages with one shape, such as blog posts	Campaign, feature and landing pages that vary	Small teams with a designer doing the editing

Most real sites mix them. Blog posts and CMS content stay on fixed templates. Marketing pages use the library. Keep them apart in your head: a section library is for pages that need different shapes.

When a library is not worth it

A library is an investment, and it does not always pay back. Skip it, or wait, if any of these is true:

- You make a new page only a few times a year.
- Every page you make is truly different, such as a brand site led by art direction.
- Your site is stable and your team is happy on its current tool. If the only pain is a messy CMS, tidy that first.
- Nobody will own the library after launch. Without an owner, it decays like anything else.

DECISION RULE

List the pages you made in the last six months. Sort them by shape. If most of them could be built from the same small set of parts, a library will help. If you made only a handful, or each one needed a new design, spend the money elsewhere.

A section library also does not require Sanity. You can run a similar idea in other tools. See headless CMS options for marketing sites, and our Webflow CMS work if a simpler setup suits you better.

The shape this guide works towards

The team gives an AI tool the content. The AI picks sections from the library, fills them in and hands back a staging link. If the team likes the page, it goes live. If something needs to change, they say what.

We have built this setup with a section library in Sanity, connected to AI through the Sanity MCP, with marketing skills we wrote. The sample schemas in later chapters are examples we wrote for this guide. They are not any client's code.

Who needs to be in the room

A library touches several jobs. Bring these people in before anyone designs a section.

- ✓ **Marketing lead.** Knows which pages are coming, and decides what the library must make easy.
- ✓ **Designer.** Owns how sections look, and which variants are allowed.
- ✓ **Developer.** Owns schemas, components and the build.
- ✓ **Editor.** The person who will really build pages. Test everything on them.
- ✓ **One named owner.** Often the marketing lead or the developer. Decides what gets added, and what gets retired.

One person can hold several of these jobs. What you cannot do is leave one empty.

DONE WHEN

You can say why you want a library in one sentence with a person in it, such as "Our campaign lead should be able to build a landing page without a developer," and you have named the owner.

TAKEAWAYS

- A section is four things together: design, schema, component and usage note.
- A page becomes an ordered list of typed sections, which makes it easy to reorder, fix and read.
- Fixed templates still suit blog posts. A library suits pages that need different shapes.
- List your last six months of pages. If they share parts, a library will help.
- No named owner, no library.

CHAPTER 2 OF 6

Planning the library

02

Most of a good library is decided before any code. Audit what you have, merge what repeats, name each section by its job and write down the recipes.

The short answer

Plan a library from your real pages. Audit the pages you have and the ones coming next, group what you find by the job each block does, merge near duplicates, then name each section for its job. Write page recipes to prove the set works. Only then design what is missing.

Our usual starting point for a marketing site is 20 or more sections, each with two to four variants, as our section library page says. Your audit decides the real number. A site with one offer may need fewer. Fewer, better sections are always easier to use.

Step 1: audit what you have

Take your most important pages and the page types on your roadmap. You do not need every blog post, because those stay on a template. Screenshot each page, cut it into blocks, and log each block in one sheet.

COLUMN	EXAMPLE	WHY YOU NEED IT
Page	/pricing	Shows where the block is used
Block job	Answer objections	Used for grouping in step 2
Current look	Two-column accordion	Spots near duplicates
Content needed	Question, answer, up to 8 pairs	Becomes the field list
Facts inside	Prices, client names	Tells you what should be a reference
Behaviour	Opens on click	Sets the build effort
Earns?	High leads, keep as is	Protects what already works

Ask the marketing lead what is coming in the next few months too. A library built only from the past will miss the next campaign. If you are also moving platform, this sheet sits alongside your URL inventory.

Step 2: group by job, then merge

Sort the sheet by job, not by how a block looks. Five blocks that all exist to prove you are trusted belong in one conversation. A common set of jobs:

- **Introduce:** say what the page offers.
- **Explain:** features, how it works, steps.

- **Prove:** logos, quotes, numbers, case studies.
- **Compare:** plans, tables, before and after.
- **Convert:** call to action, form, pricing.
- **Reassure:** FAQ, security, guarantees you can truly stand behind.
- **Navigate:** related links, tabs, table of contents.

Within each job, merge hard. Two feature grids that differ only in columns are one section. A logo strip and a case study card are two sections, because they need different content.

> DECISION RULE
>
> Same fields, different look: one section with a variant. Different fields: two sections. If a variant would hide more than a couple of fields, it is probably a separate section too.

How to name sections

Names last. Every saved section stores its type name, your code uses it, and your AI instructions will quote it. Renaming later means a content migration, so spend ten minutes on this now.

AVOID	USE	WHY
`section3`	`featureGrid`	Says what it is
`homeHero`	`hero`	A section is not tied to a page
`purpleBanner`	`ctaBand`	Colours change, jobs do not
`heroNew`	`hero` with a variant	"New" goes stale quickly
`summer24Promo`	`offerBand`	Campaigns end, the section stays

- ✓ Use camelCase, the same in the schema and the component file, so `featureGrid` maps to `FeatureGrid.astro`.
- ✓ Give each section a plain title for editors, such as "Feature grid".
- ✓ Name fields the same way across sections. If one section says `heading`, none should say `title` or `headline`.
- ✓ Name variants by what changes: `imageLeft`, `centred`, `dark`.

How many sections, and how many variants

There is no magic number. There is a clear sign of too few, and a clear sign of too many.

SIGN	WHAT IT MEANS	WHAT TO DO
Editors keep asking for a new section for ordinary pages	Too few	Look for a missing job, then add one section
Editors use a section in ways it was not meant for	Too few, or poor fields	Watch them work, then fix the fields

SIGN	WHAT IT MEANS	WHAT TO DO
Editors scroll past most of the list and pick the same few	Too many	Retire what nobody uses
Two people choose different sections for the same job	Too many, or poor usage notes	Merge, or write the rule down

Variants need the same discipline. Every variant must be designed, built, tested and explained, and each one is a new choice the editor, or the AI, has to make. Keep two to four per section, and make each one look clearly different. Avoid open-ended options such as "custom colour" or "padding size". Those are how design drifts back in.

Write page recipes

A recipe is an allowed order of sections for one page type. Recipes do three jobs. They test your list, they guide editors, and later they give AI a safe starting point. Here is a sample:

PAGE TYPE	RECIPE
Feature page	`hero` , `logoStrip` , `featureGrid` , `steps` , `quoteBlock` , `faq` , `ctaBand`
Ad landing page	`hero` , `logoStrip` , `problem` , `quoteBlock` , `form`
Comparison page	`hero` , `comparisonTable` , `quoteBlock` , `faq` , `ctaBand`
Event page	`hero` , `agenda` , `speakers` , `form` , `faq`

Now do the paper test. Pick three real pages from your roadmap and build each one from your list, on paper. Each time you get stuck, ask whether you found a truly new job or just a page that should change shape. Add a section only for the first.

Separate content from sections

While you plan, mark anything in your sheet that must be true everywhere: prices, client names, testimonials, team bios, stats. These should not be typed into sections. They should be their own documents, which a section points to. Structuring Sanity content for AI explains why, and it is the most useful habit to adopt even if you never use AI.

The same goes for shared blocks that must stay identical on many pages, such as a standard closing call to action. Decide per block: is this the same everywhere, so edit once, or is it filled in per page?

What to build first

Do not build all of it at once. Order the work so that something usable arrives early.

1. **Core set.** The sections that appear in nearly every recipe: hero, proof, features, call to action, FAQ.
2. **First real page.** Build one live page from the core set. This finds most mistakes while they are cheap.
3. **Next set.** Add what the next recipes need, in roadmap order.
4. **Long tail.** Leave rarely used sections until someone has a dated need for them.

DONE WHEN

You have a list of sections with one job each, a field list for each, named variants, and recipes that cover three real upcoming pages without a new design.

TAKEAWAYS

- Audit real pages and your roadmap, then log every block in one sheet.
- Group by job. Same fields means a variant, different fields means a new section.
- Name sections for their job, because names are expensive to change later.
- Keep two to four clearly different variants, and avoid open-ended style options.
- Paper test recipes on three real pages before you build anything.

CHAPTER 3 OF 6

Sanity schemas for sections

03

How to turn your plan into code: a page with a sections array, narrow fields, variants as fixed lists, rules on every field and an Astro component for each type.

The short answer

In Sanity, model each section as a named object type. Give the page document one array that accepts only those types. Use fixed lists for variants, length limits on text and required alt text on images. In Astro, map each section's `_type` to a component.

The samples here are examples we wrote for this guide, not code from a client project. We checked the syntax against Sanity's and Astro's docs. These tools change, so check the current docs before you copy anything. For a fuller list of modelling rules, read Structuring Sanity content for AI.

The page document

A page has a few page-level fields and one array. Listing the allowed members in `of` is what limits the page to your library. Anything not listed cannot be added in the Studio.

```
defineType({ name: 'page', type: 'document', fields: [ defineField({ name: 'title', type: 'string', validation: (rule) => rule.required() }), defineField({ name: 'slug', type: 'slug', options: { source: 'title' }, validation: (rule) => rule.required() }), defineField({ name: 'sections', type: 'array', of: [ defineArrayMember({ type: 'hero' }), defineArrayMember({ type: 'featureGrid' }), defineArrayMember({ type: 'faq' }) ], validation: (rule) => rule.required().min(1) }) ] })
```

Sanity gives each array item a unique `_key`, so you do not set it yourself. Add your SEO fields here too, in a field group so they sit in their own tab.

A section as a named object

Each section is its own object type, in its own file. One section, one file. Register it in your schema types with the others. Here is a sample hero, shortened:

```
defineType({ name: 'hero', title: 'Hero', type: 'object', fields: [ defineField({ name: 'heading', type: 'string', description: 'One line. Say what the page offers.', validation: (rule) => rule.required().max(60) }), defineField({ name: 'lead', type: 'text', rows: 3, validation: (rule) => rule.max(180) }) ] })
```

Set the limits from your design. Test the longest heading your layout can hold at phone width, and use that number. The 60 here is only a sample. Use `description` for short guidance on every field. Editors read it, and it is a good place to state the rule in plain words.

Variants as fixed lists

A variant is a string field with a fixed list of values. Radio buttons show every choice at once. Sanity's docs note that they also avoid the blank option a dropdown shows at the top.

```
defineField({ name: 'layout', type: 'string', options: { list: [ { title: 'Image right', value: 'imageRight' }, { title: 'Centred', value: 'centred' } ], layout: 'radio' }, initialValue: 'imageRight', validation: (rule) => rule.required() })
```

An initial value means a new section already looks right when added. If a field only applies to one variant, you can hide it with a callback:

```
hidden: ({ parent }) => parent?.layout === 'centred'
```

The docs we read do not say how a hidden required field behaves in validation, so test that combination in your Studio. And if you find yourself hiding half the fields, go back to the decision rule in chapter 2. It is probably two sections.

Images, rich text and references

Three field types need extra care, because this is where layouts and facts tend to break.

Images. Add alt text as a custom field on the image, and turn on the hotspot tool so crops keep the subject in view.

```
defineField({ name: 'image', type: 'image', options: { hotspot: true }, fields: [ defineField({ name: 'alt', type: 'string', validation: (rule) => rule.required() }) ] })
```

Rich text. Portable Text is powerful, so narrow it. Declaring `styles`, `lists` and `marks` replaces the default sets. A section body rarely needs its own headings, because the section already has one.

```
of: [{ type: 'block', styles: [{ title: 'Normal', value: 'normal' }], lists: [{ title: 'Bullet', value: 'bullet' }], marks: { decorators: [{ title: 'Strong', value: 'strong' }] } }]
```

Links stay available here, because the default link annotation remains unless you declare your own `annotations`.

References. Point to facts held elsewhere. A strong reference, the default, stops anyone deleting a testimonial that a live page still uses. That is usually what you want.

```
defineField({ name: 'quote', type: 'reference', to: [{ type: 'testimonial' }] })
```

Query the page with GROQ

Fetch the page and its sections with GROQ. The spread (`...`) returns every field of each section. A conditional adds what a particular type needs, such as following a reference with `->`:

```
*[_type == 'page' && slug.current == $slug][0]{ title, sections[]{ ..., _type == 'quoteBlock' => { quote->{ text, name, role } } } }
```

For a static build, the official `@sanity/astro` integration gives you a client through `import { sanityClient } from 'sanity:client'`, and its docs say to set `useCdn: false`. By default, recent API versions read published content only, which is what you want on the live site. Older API versions default to a different view, so set your API version on purpose.

Render sections in Astro

Astro supports dynamic tags, so one small loop can render any page. In the frontmatter, map each type name to its component:

```
const map = { hero: Hero, featureGrid: FeatureGrid, faq: Faq };
```

Then in the template:

```
{page.sections.map((s) => { const Section = map[s._type]; if (!Section) throw new Error('No component for ' + s._type); return <Section {...s} />; })}
```

- **Capitalise the variable.** Astro treats a lower-case name as a literal HTML tag.
- **Fail the build on an unknown type.** A schema with no component should stop the build, not leave a hole in a live page.
- **Keep client directives inside each component.** Astro's docs say `client:*` does not work on a dynamic tag. Put the interactive island inside the section component instead.
- **Type your props.** Add an `interface Props` to each section component.

Keep schema and code in step

Schema and components can drift apart, so make the machine check them.

- ✓ Run `sanity schemas validate` to catch problems in your schema types.
- ✓ Run Sanity TypeGen: `npx sanity schema extract`, then `npx sanity typegen generate`. It writes TypeScript types for your schema and your GROQ results, so a renamed field shows up as an error in your code.
- ✓ Keep schemas in Git with the Astro code, and change them in the same pull request.

DONE WHEN

A new page can be built in the Studio using only listed sections, every text field has a limit, every image needs alt text, and an unknown section type fails the build.

TAKEAWAYS

- One sections array, with only your named types listed in of.
- Variants are fixed lists. Limits come from testing your design at phone width.
- Narrow rich text, require alt text, and point to facts with references.
- Map _type to a component, and fail the build on an unknown type.
- Check schemas and generate types, so code and content stay in step.

CHAPTER 4 OF 6

Studio for marketers

04

A good schema can still feel bad to use. How to set up the insert menu, previews, validation messages, live preview and publishing so editors need no help.

The short answer

Your editors should be able to find the right section by sight, know what each field wants, see the real page while they work, and publish with confidence. In Sanity Studio that means a grouped insert menu with pictures, clear list previews, plain validation messages, a live preview and the right roles.

This chapter covers setup. For what a normal week looks like for your team afterwards, read Astro and Sanity for marketing teams.

An insert menu people can read

By default, an array shows a plain list of type names. With `options.insertMenu` you can group sections by job and show a grid of pictures. Use the same jobs you found in chapter 2.

```
options: { insertMenu: { groups: [ { name: 'introduce', title: 'Introduce', of: ['hero'] }, { name: 'prove', title: 'Prove', of: ['logoStrip', 'quoteBlock'] } ], views: [ { name: 'grid', previewImageUrl: (name) => `/static/preview-${name}.png` }, { name: 'list' } ] } }
```

If you list both views, the first shows by default, and a toggle switches between them. A filter box appears on its own once the array allows more than five types. Each section needs a picture for the grid, so add "make the preview image" to your definition of done for a new section. Without one, the Studio falls back to the section's icon.

Previews that say what is in the page

In the Studio, a page is a list of collapsed sections. If that list reads "Hero, Hero, Feature grid", nobody can find anything. Set a preview on each section, showing its type and its actual words:

```
preview: { select: { title: 'heading', layout: 'layout' }, prepare({ title, layout }) { return { title: title || 'Empty hero', subtitle: `Hero, ${layout}` }; } }
```

Now the list reads like an outline of the page. Showing "Empty hero" also makes unfinished work easy to spot. Sanity's docs warn that `prepare()` runs wherever a preview shows, including global search, so pass what it needs through `select`.

Validation messages that help

Rules are only as good as their messages. "Required" tells an editor nothing about what to do. Sanity has levels, and picking the right one matters: errors block publishing, warnings do not.

USE	FOR	SAMPLE MESSAGE
Error	Anything that breaks the layout or the page	"Keep this under 60 characters, or it wraps badly on phones."
Error	Accessibility and SEO basics	"Describe this image for people who cannot see it."
Warning	Advice that sometimes has good exceptions	"Most pages work better with a call to action near the end."
Warning	Editorial style	"This heading is long. Could it be shorter?"

A custom rule can check the whole page. For example, a rule on the `sections` array can warn if the first section is not a hero. Each `error()` or `warning()` applies to the rule right before it, so chain them with care:

```
validation: (rule) => rule.required().max(60).error('Keep this under 60 characters.')
```

DECISION RULE

If breaking the rule would look wrong or hurt a visitor, use an error. If it is a matter of judgement, use a warning. Too many errors teach people to resent the tool, and they will start asking developers to remove rules.

Live preview on an Astro site

Editors trust what they can see. Sanity's Visual Editing shows drafts on a preview of the site as people type. Clicking an element jumps to its field, and sections can be dragged to reorder them. The Presentation tool hosts this inside the Studio.

There is one catch for Astro. The `@sanity/astro` docs say Visual Editing works only on server-rendered pages, and recommend `output: 'server'` with an adapter. Your live site can still be static. A common answer is two builds from one codebase:

	LIVE SITE	PREVIEW SITE
Rendering	Static	Server
Content	Published perspective	Drafts perspective, with `useCdn: false`
Read token	Not needed for public data	Needed, kept on the server
Search engines	Open	Blocked with `noindex` and a login

This is more work than one build, so price it in. Keep the preview's token out of any public code. Our glossary entry on preview deployments covers the basics.

Publishing paths

Pick the simplest path that fits your team, and write it down.

- **Publish now.** The editor fixes any errors and publishes. A webhook tells your host to rebuild. By default, these webhooks fire only on published changes, so typing in a draft will not trigger builds. Your host does the rebuild, so test that it works end to end.
- **Publish later.** Scheduled drafts set one draft to go live at a set time. Remember that a static site still needs a rebuild after the publish, so test the timing.
- **Publish together.** Content Releases group several documents to preview, schedule and publish as one. Sanity lists this as a paid feature on certain Enterprise plans, so check yours.

Plans and features change, so confirm what yours includes in the current docs before you promise a workflow to your team.

Roles and history

Give each person the least access that lets them do their job. Sanity lists built-in roles, and which ones you get depends on your plan.

ROLE	SANITY SAYS	GOOD FOR
Contributor	Can write documents, but not publish them	Freelance writers, junior editors, AI tools
Editor	Read and write on all datasets, limited project settings	Your marketing team
Developer	Read and write, with project settings access for developers	Whoever maintains the code
Administrator	Full access to project settings	One or two people

Mistakes will happen, so show people document history early. They can review what changed and when from the document's status or menu. How far back it goes depends on your plan.

Test it on a real editor

Before you call the Studio finished, run this test. Give the person who will really build pages a real brief. Do not explain anything. Sit beside them and watch.

- ✓ Could they find the right sections without asking?
- ✓ Did they know what each field wanted?
- ✓ Did any error message confuse them?
- ✓ Did the preview match the page they expected?
- ✓ Did they feel safe pressing publish?
- ✓ Did they know who to ask if a section was missing?

Every time they stop, write it down. Each stop is a fix to a field name, a description or a picture, and those fixes are usually small.

DONE WHEN

An editor can take a brief, build a page and publish it without help, and can explain which sections they chose and why.

TAKEAWAYS

- Group the insert menu by job, and give every section a preview picture.
- Make the list of sections read like an outline of the page.
- Errors for what breaks the page, warnings for what needs judgement.
- Visual Editing on Astro needs server rendering, so plan for a preview build.
- Test the Studio by watching a real editor build a real page.

CHAPTER 5 OF 6

AI page building with guardrails

05

What to let an AI change, how to limit its access through the Sanity MCP, what to write in its instructions, and how a person reviews every page before it goes live.

The short answer

Let AI draft pages from your library, and have a person publish them. Connect it through the Sanity MCP, limit its access at the permission level, give it written instructions, check its work with your own validation step, and review every page on a staging link. Prompts alone are not a guardrail. Permissions and structure are.

For the case for this approach, read AI page builder for your team. For the content model it depends on, read Structuring Sanity content for AI. This chapter is about how to set the limits.

What the finished setup looks like

A section library in Sanity, AI connected to it through the Sanity MCP, and marketing skills written for it. The team sends the content. The AI picks the right sections from the library, builds the page and shares a staging link. If the team likes it, it goes live. If something needs to change, they say what, and it gets changed.

The rest of this chapter is general advice for setting up your own. It is not a description of any client's settings.

What the MCP gives an AI

The Sanity MCP server is hosted by Sanity at `mcp.sanity.io`. It lets AI tools work with your project through the Model Context Protocol. Its tools change over time, so check the current docs. When we read them, they fell into these groups:

AREA	SAMPLE TOOLS	RISK
Read	`query_documents`, `get_schema`	Low on public content, higher if your dataset holds private data
Draft	`create_documents`, `patch_documents`	Medium. Patches go to the draft or release version, not to published content. Watch types with `liveEdit` on: the docs say create can make published documents for those
Publish	`publish_documents`, `unpublish_documents`	High. Changes the live site at the next build
Schema	`deploy_schema`, `schema_fields_patch`	High. Changes the library itself

The takeaway: the server can do far more than draft. If you connect it with broad access and only ask the AI nicely to stay in drafts, the AI still has a publish button.

What the AI may and may not change

Write this table for your own team, and agree it before connecting anything. Here is a starting point.

THING	AI	PERSON
Choose sections from the library	Yes, within recipes	Reviews
Choose variants from fixed lists	Yes	Reviews
Fill in text fields, from the brief	Yes	Reviews every word
Select existing images	Yes, if they fit	Reviews
Point to existing facts by reference	Yes	Checks the right one was used
Create or edit prices, clients, quotes, stats	No	Yes, in their own documents
Change a published page in place	Draft only	Reviews the change, then publishes
Publish or unpublish	No	Yes
Change schemas, add sections	No	Developer, through Git
Change redirects, tracking, code	No	Developer

DECISION RULE

If a wrong value would be a false claim about your company, the AI may only point to it, never type it. If a mistake would reach visitors with no person seeing it first, the AI may not do it.

Enforce the limits with access

Now make the table real. The MCP docs say OAuth is the default, and that with an API token, tool calls are limited to that token's role and permissions. They also say a personal token shares your own role. So if an administrator connects the AI under their own account, assume the AI has the same reach.

- **Use a dedicated token for the AI,** so you can remove its access without touching anyone else's.
- **Pick the narrowest role that can write drafts.** Sanity's Contributor role can write but not publish, on the plans that include it. Sanity's token docs describe tokens holding role memberships, so check in the management console which role yours holds.
- **Test it.** Ask the AI to publish a test draft, on a test dataset first, and confirm it is refused. Do the same for a schema change. Do not assume, try it.
- **Use a separate dataset for your pilot,** if your plan allows, so early mistakes cannot touch live content.
- **Check your AI client.** Some let you approve or switch off single tools. Use that as a second layer, not the only one.
- **Use one token per application,** as Sanity's docs advise, and set an expiry.

Write the skills

"Skills" here means written instructions the AI reads before it works. Yours should be short enough to read in one sitting, and specific enough to act on. Include:

- **Library guide.** Each section: its job, when to use it, when not to. Copy this from your usage notes.
- **Recipes.** The allowed orders from chapter 2, by page type.
- **Voice.** Your tone, your banned words, and examples of good copy from your own site.
- **Fact rules.** Use references for facts. Use only what is in the brief. If something is missing, leave a note instead of guessing.
- **Process.** Read the schema first, save as a draft, run the checks, then send the staging link with a list of open questions.

Then test with bad briefs as well as good ones. Give it a brief with no proof points and see if it invents some. Give it a brief asking for a section you do not have and see what it does. It should ask, or flag it.

Do not rely on Studio validation

This is the one surprise that catches people. Sanity's validation docs say schema rules only run in the Studio, so API writes are not checked. The MCP docs say create and patch use looser validation, that publishing runs stricter checks, and that custom validation rules are skipped through MCP.

So your length limits and custom rules may not stop an AI draft. Add your own check before anyone reviews the page. Sanity's CLI has `sanity documents validate`, which checks a dataset's documents against your Studio schema. A small script can check the rest, such as approved recipes, and whether references resolve. Then render the page: your Astro build fails on unknown types, and a staging build shows the page as a visitor would see it.

Review before publish

Every page gets a person's review on a staging link, marked `noindex`. See staging if the idea is new. The reviewer checks:

- ✓ Every claim is true, and every number has a source.
- ✓ Names, prices and quotes come from references, not typed text.
- ✓ Sections follow a recipe, and the order makes sense for this reader.
- ✓ The copy sounds like you, and it says something specific.
- ✓ The page works on a phone, with the longest headings.
- ✓ Title, description, slug, headings and alt text are all set.
- ✓ Links and forms work, and any tracking fires.
- ✓ The page does not compete with one you already have.

DONE WHEN

The AI has been refused a publish in a test, a bad brief produced a question and not an invention, and a named person has reviewed and published one real page.

TAKEAWAYS

- AI drafts, a person publishes. Enforce that with permissions, not prompts.
- Write a table of what the AI may and may not change, and agree it first.
- Facts are references. The AI may point to them, never type them.
- Custom validation is skipped through MCP, so add your own check.
- Test the limits: confirm a publish attempt is refused.

CHAPTER 6 OF 6

Testing and governance

06

Libraries decay without rules. What to test on each section, how a new one gets added, how to change a section without breaking live pages, and who owns what.

The short answer

Test each section at its limits, on one page that shows every variant. Add new sections through one short request process. Change live sections carefully, because saved content does not update itself when the schema changes. Retire sections in stages. Name one owner, and review the library on a regular rhythm.

The aim is a boring library: one that still makes sense to a new team member a year from now.

What to test on every section

Test the extremes. Average content rarely breaks anything. Run this for each variant.

TEST	WHAT TO TRY	PASS WHEN
Longest content	Text at the validation limit, in your longest real words	No overflow or broken layout at phone width
Shortest content	Only required fields filled	No gaps or empty wrappers
Item counts	The fewest and most items a list allows	Grid still looks intended
Images	Tall, wide and small images, odd hotspots	Crops are fine, and no layout shift
Neighbours	Next to each other section, dark next to dark	Spacing holds
Screens	Phone to wide desktop	No sideways scroll
Keyboard and screen reader	Tab through it, check heading order	Everything can be reached, and headings make sense
No JavaScript, reduced motion	Turn each off	Content can still be read
Speed	Lighthouse on a page with the section	Meets your budget

Build a kitchen sink page

Make one private page that holds every section in every variant, at both the longest and shortest content. Mark it `noindex` and leave it out of your sitemap. It becomes your test bench, and a living catalogue for editors.

Use it for:

- **Screenshot tests.** Compare before and after on each code change, so you see what really moved.

- **Reviews.** A designer can check the whole library in one scroll.
- **Onboarding.** A new editor can see what exists.
- **AI checks.** Compare what the AI built with what a section should look like.

Keep this page's content in code or in a seed file, if you can, so a stray edit cannot hide a bug.

Automatic checks

Run these in your CI on each pull request that touches sections, so problems stop before they reach an editor.

- ✓ Schema check: `sanity schemas validate`.
- ✓ Type check: TypeGen, then your type checker.
- ✓ Content check: `sanity documents validate` against a copy of your dataset, to find saved content that no longer fits.
- ✓ Build: every section type has a component.
- ✓ Pages: crawl the built site for errors, broken links and sideways scroll.
- ✓ Screenshots of the kitchen sink page.
- ✓ Speed and page weight budgets.

You do not need all of these on day one. Start with the build and the schema check, then add the rest as the library grows.

How a new section gets added

When adding is too easy, the library bloats. When it is too hard, people work around it. A short written request is enough.

1. **Request.** Which page needs it? What job does it do? Which existing section came closest, and why does it fall short?
2. **Owner decides.** One of four answers: use an existing section, add a variant, add a section, or make a one-off.
3. **Design.** Every screen size, every variant, the longest content.
4. **Build.** Schema, component, preview image, usage note, kitchen sink entry.
5. **Test.** The table above, then watch an editor try it.
6. **Announce.** Tell editors, and add it to the AI's library guide. If the AI's guide is not updated, it will not know the section exists, or when to use it.

> DECISION RULE
>
> Add a section when at least two planned pages need it, or one page needs it and it earns real money. Otherwise, build a one-off page and look again if the need comes back. Writing down why you said no saves the same debate later.

Change sections without breaking pages

Sanity's migration docs are clear: changing a schema will not change or delete content already in your dataset. Old documents stay as they are, so they may no longer match. So sort every change by how risky it is.

CHANGE	RISK	HOW TO HANDLE IT
New optional field	Low	Ship it, and make sure the component copes when it is empty
Tighter length limit	Medium	Check saved content first. Long text will start failing validation
Remove a variant	Medium	Move pages off it first, then remove
Rename a field	High	Add the new one, migrate content, ship the code, then remove the old one
Rename a section type	High	Avoid it if you can. Plan and test a migration on a copy first

For high-risk changes, use the migration tool. `sanity migrations create` makes a script, and `sanity migrations run <id>` runs it. The docs say it always runs as a dry run unless you add `--no-dry-run`, so you can read the patches first. They also say to back up first with `sanity datasets export`. Do not keep two sections called `hero` and `heroV2` for long. Parallel versions are fine during a move, but they should have an end date.

Retire sections in stages

Removing a section from `of` does not remove it from saved pages. So go in order:

1. **Find it.** Use GROQ to list the pages that still use the section.
2. **Warn.** Add a warning in the Studio, and tell editors and the AI's guide to stop using it. Sanity also has a `deprecated` property for document types and fields. Try it on your section fields, and check how it behaves there before relying on it.
3. **Move.** Replace it on each page, by hand or with a migration.
4. **Remove.** Take it out of `of`, then delete the schema, the component and its preview image. Only do this when your search returns no pages.

Who owns what

Write this down once, and put names in it. Titles are only examples.

JOB	WHO	HOW OFTEN
Decide on section requests	Library owner	As they come in
Design and variant rules	Designer	Per change
Schemas, components, checks	Developer	Per change
Fact documents: prices, quotes, clients	Marketing lead	When facts change
AI skills and recipes	Library owner, with marketing	Each new section

JOB	WHO	HOW OFTEN
Usage review	Library owner	Every few months

In the usage review, count how often each section is used with a GROQ query, ask editors what they avoided, and read a few AI drafts. Merge or retire what nobody uses.

DONE WHEN

A new person can find out what exists, how to request a section and who decides, without asking anyone.

If you want this built for you, an Astro and Sanity site starts at $2,000 and includes a section library, as our pricing page shows. Adding a library to an existing Sanity site gets a fixed quote after a 20 minute call. See section library.

TAKEAWAYS

- Test at the extremes, on one page that shows every section and variant.
- One short request process, and one owner who can say no.
- Schema changes do not touch saved content, so sort changes by risk and migrate.
- Retire in stages. Search first, and remove last.
- Put names on the jobs, and review usage every few months.

QUESTIONS

Questions people ask.

What is a section library?

A set of reusable page sections, each with a design, a Sanity schema, an Astro component and a note on when to use it. Pages are ordered lists of these sections. See our section library page for what we build.

How many sections do we need?

Your audit decides. Our usual starting point for a marketing site is 20 or more, each with two to four variants, but a site with one offer may need fewer. If editors skip most of the list, you have too many.

When should something be a variant and when a new section?

Same fields with a different look is a variant. Different fields is a new section. If a variant would hide more than a couple of fields, make it a separate section.

Does Visual Editing work on a static Astro site?

The @sanity/astro docs say Visual Editing works only on server-rendered pages. A common answer is a static live site plus a separate server-rendered preview build, from the same code. Check the current docs.

Can AI publish pages by itself?

The Sanity MCP has a publish tool, so this depends on the access you give it. We set AI up to save drafts, and a person reviews and publishes. Limit this with roles, then test that a publish attempt is refused.

Do Sanity validation rules stop bad AI drafts?

Not on their own. Sanity says schema rules only run in the Studio, and its MCP docs say custom validation rules are skipped through MCP. Add your own check before review, such as sanity documents validate.

What happens to live pages when we change a section?

Changing a schema does not change saved content, so old pages may stop matching. Sort changes by risk. New optional fields are low risk. Renames need a planned migration, tested on a copy first.

Keep going

- Astro and Sanity
- Section library
- Astro and Sanity for marketing teams
- AI page builder for your team
- Structuring Sanity content for AI
- Headless CMS for marketing sites
- Sanity CMS
- Webflow to Astro
- What is GROQ?
- What is Portable Text?
- What is Model Context Protocol?

Planning a section library? Book a free 20 minute call

Kookies. www.kookies.io. hello@kookies.io