
A RESOURCE FOR DESIGN SYSTEM AND ENGINEERING TEAMS

Component contract checklist.



For systems that people and agents both build with.

Enumerate it. Encode it. Test it.

63

checks, each with its evidence

08

sections, identity to CI

01

component at a time

Write down what the component allows.

A component contract is the single place where a component's intent lives: its props, the combinations that are allowed, its states, what it may contain and how it behaves for keyboard and screen-reader users.

Nathan Curtis describes it as the artifact that lets intent be "implemented, verified and evolved across implementations" [S01]. The contract is what a reviewer checks a pull request against, and what an AI agent reads before it writes `<Button>`.

Agents fail in predictable ways when that artifact is missing or vague. They invent props that sound plausible, recreate components the system already ships, reach for deprecated patterns and drop accessible names [S15, S18]. The WebAIM Million 2026 still finds missing form labels on 51% of home pages and empty buttons on 30.6% [S19]. Each item below closes one of those gaps and names the evidence that proves it is closed.

The kit that comes with this guide makes the checklist executable: a JSON Schema for contract files, a complete Button contract, discriminated-union types with type tests, a CI validator and a generator that expands a contract into only the prop combinations it allows.

Version 1.0 / Sources checked 24 September 2026

Field Guide 02 of the Design × AI series. Pairs with Field Guide 03 (token layer rules) and 04 (promptable docs template), which use the same Button.

Practical guidance, not a standard. Field names in the contract schema are a recommendation that maps onto Custom Elements Manifest, Storybook manifests and Figma Code Connect where they overlap. WCAG references are to WCAG 2.2 (W3C Recommendation, 12 December 2024). Prepared with AI assistance and edited by hand.

START HERE

Pick your route, then read the labels.

Use one copy per component. Work through the route that matches where the component is today; not every item applies on day one, and the stable gate says which ones must.

Starting a new component

Sections 01 to 04 first. Write the contract before the implementation, so the API review happens on data rather than on a pull request.

Hardening an existing one

Run the validator against a first draft contract, then work section 05 (accessibility) and section 08 (enforcement). Existing drift shows up as prop mismatches.

Opening it to agents

Section 06 (agent-facing guidance) and the canonical examples in section 01. Agents read JSDoc, manifests and MCP responses, not your Figma annotations.

Promoting to stable

Every item tagged STABLE GATE. The schema refuses status stable until those fields exist, so the promotion is a passing build, not a meeting.

Read the labels before the checks.

Label	Meaning
SCHEMA	Captured as a field in the contract file and validated by the schema.
TEST	Proven by an automated test: type test, interaction test, axe, visual regression.
WCAG A / AA / AAA	Traces to a WCAG 2.2 success criterion at that level. AAA items are system targets, not legal minimums.
AGENT	Exists so an AI agent can read, choose and use the component correctly.
STABLE GATE	Enforced by the kit's schema when status is stable.
PRACTICE	A recommended working method rather than a checkable artifact.

For every item, keep three things together.

Status: open, evidence recorded, or not applicable with a reason. Owner: a named person. Record: the link to the story, test run or contract line that proves it.

A checked box is a claim. The evidence line under each item says what makes it true.

SECTION 01

Name the component and its API.

Agents choose components by name and summary, then write props from whatever list they can find. Make both complete and identical everywhere.

Suggested owners: Design-system lead + engineering lead

☐ One canonical name everywhere

01.01 **SCHEMA**

The same name in the code export, the Figma component, the Storybook title and the contract `name`. No `PrimaryButton` in Figma and `Button` in code.

Evidence: `name` field; a CI script that diffs names across the Storybook manifest, Code Connect map and package exports.

Recommended practice. S01, S09, S13.

☐ The exact import is written down

01.02 **SCHEMA** **AGENT**

State the import statement an agent should copy, including the package. Agents that guess paths recreate components instead of importing them.

Evidence: `import` field; a fixture app that resolves the import in CI.

Storybook manifests expose `import` per component. S09, S15.

☐ Summary and description live in the source

01.03 **SCHEMA** **AGENT**

A one-sentence summary and a longer description, written in JSDoc on the component. Storybook builds its agent-facing manifest from JSDoc, not from MDX pages.

Evidence: `summary` field (10 to 200 characters); manifest lint that the description is non-empty.

S09, S18.

☐ When to use, and when not to

01.04 **SCHEMA** **AGENT**

Each "do not use" names an alternative component that exists: navigation goes to `Link`, a setting goes to `Switch`.

Evidence: `whenToUse[]`, `whenNotToUse[].alternative`; the validator warns when an alternative has no contract.

Recommended practice.

☐ Lifecycle status from a fixed set

01.05 **SCHEMA**

Experimental, beta, stable or deprecated. Consumers and agents need to know what they may depend on.

Evidence: `status` enum; shown as a label in docs.

Primer and Carbon run explicit lifecycles. S03, S04.

☐ **Every public prop is documented**01.06 **SCHEMA** **TEST**

Type, required flag, default and a description for each public prop. Nothing public exists only in the TypeScript file.

Evidence: `props{}`; a bidirectional diff between the contract and docgen props that fails on a mismatch either way.

Carbon requires a fully typed, exported prop interface. S03, S09.

☐ **Closed sets are enums, never open strings**01.07 **SCHEMA** **TEST**

If only four tones exist, the type is a union of four literals, the schema is an enum of four values and an unlisted value fails.

Evidence: `enum` in `props` and `propsSchema`; a type test where an unlisted value is `@ts-expect-error`.

S07, S17.

☐ **Defaults match in code, design and contract**01.08 **SCHEMA** **TEST**

The default tone in Figma is the default tone in code is the `default` in the contract.

Evidence: `default` fields; a render test with no props that asserts the computed defaults.

Recommended practice. S02.

☐ **Props named by one published convention**01.09 **PRACTICE**

The same concept has the same name in every component: always `tone` (never `kind` in one place and `appearance` in another), always `on*` for events, booleans named as states (`disabled`, `loading`).

Evidence: A lint pass across all contracts for synonym props.

Curtis: favor normalized over redundant. S01.

☐ **Pass-through behavior is explicit**01.10 **SCHEMA**

Say which native attributes are forwarded, whether `className` and `style` are accepted and whether a ref is forwarded. Agents otherwise assume everything passes through.

Evidence: Documented in `props` or `meta`; a type test.

Recommended practice.

An agent can only respect a boundary it can read.

SECTION 02

Enumerate what may be combined.

The variant space is the product of the enums, minus the combinations that should never exist. Write the minus down as rules, or a generator will produce every permutation it can imagine.

Suggested owners: Design-system lead + engineering lead

☐ Forbidden combinations are rules, not folklore

02.01 **SCHEMA** **STABLE GATE**

Every combination that must not exist is a named rule: a ghost Button is never full width, an icon-only Button needs an accessible name.

Evidence: `constraints[]` indexing `if/then/not` rules in `propsSchema.allOf`; the validator checks that each constraint id has a rule.

JSON Schema 2020-12 conditionals. S07.

☐ Invented props fail validation

02.02 **SCHEMA**

`propsSchema` sets `additionalProperties: false`, so `shadow`, `elevated` or any plausible-sounding prop an agent invents is rejected.

Evidence: The validator feeds `propsSchema` an unknown prop and fails if it passes.

Storybook's AI docs name invented props as the common agent failure. S10.

☐ Forbidden combinations are compile errors too

02.03 **TEST**

Model the props as a union so the forbidden set does not type-check. Test with runtime values too, not only literals: a union that rejects `loading={isSaving}` is a union nobody can use. The kit's own first draft failed exactly that test.

Evidence: `button.test-d.ts` with one `@ts-expect-error` per rule and a block of runtime-value cases, run with `tsc` or `vitest --typecheck`.

S17. Worked example in the kit.

☐ Runtime behavior on violation is defined

02.04 **SCHEMA**

For each rule: type error, dev-mode warning, throw, or a documented coercion (for example: `loading` wins over `disabled`).

Evidence: `constraints[].onViolation`; a unit test that asserts the warning.

Recommended practice.

☐ Required-together props are declared

02.05 **SCHEMA** **TEST**

When one prop requires another (icon-only requires `aria-label`, `href` rules out `loading`), the rule is data, not a sentence in a doc.

Evidence: `if/then` in `propsSchema` (use `dependentRequired` only for presence-based rules); a type test.

S07.

☐ **Stories cover the allowed set, and only that**02.06 **TEST**

Generate the matrix story from the contract, not by hand. For the kit's Button, 192 raw permutations reduce to 99 allowed combinations.

Evidence: `allowed-combinations.mjs --json` feeds the matrix story; CI compares the story count with the computed allowed set.

Kit script. See the appendix.

☐ **Figma variants match the allowed set**02.07 **TEST**

No Figma-only variants and no missing ones. Code Connect maps are where design and code meet.

Evidence: A scheduled script that compares Figma variant properties (via Code Connect `figma.enum` maps) with contract enums.

S13.

Coverage means every allowed combination, not every possible one.

SECTION 03

Specify sizes, states and layout.

States are where a component meets real use. Keep interactive, data and validation states apart, and give every size a target that a finger or pointer can hit.

Suggested owners: Design lead + engineering lead + accessibility reviewer

☐ Sizes are a closed set with token references

03.01 **SCHEMA** **STABLE GATE**

Each size lists its height and the tokens behind it. No free-form `height` prop.

Evidence: `sizes[]` with `tokens`; one visual story per size.

Carbon: tokens only in the design spec. S03.

☐ Every interactive size meets the target minimum

03.02 **SCHEMA** **WCAG AA**

At least 24 by 24 CSS px, or a documented exception (spacing, equivalent, inline, user-agent control, essential). The schema refuses a smaller `minTargetPx`.

Evidence: `sizes[].minTargetPx` (minimum 24); a bounding-box assertion per size story.

WCAG 2.2 SC 2.5.8 Target Size (Minimum), AA. S05.

☐ Width behavior and small viewports are declared

03.03 **SCHEMA** **WCAG AA**

Intrinsic, fill or fixed width, and verified behavior at 320 CSS px without two-dimensional scrolling.

Evidence: `layout.width`, `layout.minViewportPx`; a visual mode at 320 px.

WCAG 2.2 SC 1.4.10 Reflow, AA. S05.

☐ Text-spacing overrides do not break it

03.04 **TEST** **WCAG AA**

Line height 1.5, paragraph spacing 2x, letter spacing 0.12em and word spacing 0.16em must not clip or overlap text.

Evidence: A story that injects the SC 1.4.12 spacing CSS and asserts no overflow.

WCAG 2.2 SC 1.4.12 Text Spacing, AA. S05.

☐ Interactive states are enumerated

03.05 **SCHEMA** **STABLE GATE**

Rest, hover, focus-visible, active, disabled and read-only where relevant, each with a story.

Evidence: `states.interactive[]`; one story per state (pseudo-state addon or play function).

Carbon lists hover, focus, selected, disabled, read-only, error and warning. S03.

☐ Data and validation states are their own sets

03.06 **SCHEMA**

Loading, empty, selected or expanded are data states. Validation is an enum (error, success, warning), not a single `error` boolean.

Evidence: `states.data[]`, `states.validation[]`; union types.

Primer's TextInput uses `validationStatus: 'error' | 'success'`.

☐ **States are visible without color alone**

03.07

TEST

WCAG AA

Every state indicator reaches 3:1 against adjacent colors and uses more than hue. Disabled components are exempt from the contrast requirement.

Evidence: axe per state story; contrast pairs checked at token level (Field Guide 03).

WCAG 2.2 SC 1.4.11 Non-text Contrast, AA. S05.

☐ **Disabled and loading semantics are decided**

03.08

SCHEMA

WCAG AA

Native `disabled` removes focus; `aria-disabled` keeps it discoverable. Loading states say what is announced and whether the control stays operable.

Evidence: `accessibility.disabledStrategy`; an interaction test for focusability and for the loading announcement.

WCAG 2.2 SC 4.1.3 Status Messages, AA. S05, S06.

A state that is not in the contract will be invented in the product.

SECTION 04

Constrain composition and content.

What a component may contain, and what may contain it, is where generated UI drifts furthest from the design. Allow-lists beat descriptions.

Suggested owners: Design-system lead + content designer

☐ Every slot is named and described

04.01 **SCHEMA**

Name, description and whether it is required. Anonymous `children` with no rules is an open door.

Evidence: `slots{}`; Custom Elements Manifest `slots[]` for web components.

S08.

☐ Slots take an allow-list, not free-form content

04.02 **SCHEMA** **AGENT**

Each slot lists the components it accepts, or text only. Figma slots can require preferred instances; code must enforce the same list.

Evidence: `slots.<name>.allowed[]`; a dev-mode check or lint rule.

Figma slots with preferred instances, GA 10 June 2026. S14.

☐ Item limits are stated where layout depends on them

04.03 **SCHEMA**

A toolbar that breaks at seven actions says so. Figma layer limits guide but do not block, so code has to.

Evidence: `minItems` / `maxItems`; a story at the maximum.

S14.

☐ Parents and forbidden descendants are declared

04.04 **SCHEMA** **TEST**

No interactive element inside a Button; `Tab` only inside `TabList`.

Evidence: `composition.allowedParents[]`, `composition.forbiddenDescendants[]`; axe `nested-interactive`.

S06.

☐ Compound components publish their canonical tree

04.05 **SCHEMA** **AGENT**

List the sub-components and the minimal correct structure as a canonical example. Agents copy structure more reliably than they infer it.

Evidence: `composition.subcomponents[]`; an example marked `canonical: true` with a matching story.

S09.

☐ Label rules are data

04.06 **SCHEMA**

Maximum length, casing and voice (verb first for actions), so content lint and agents apply the same rule.

Evidence: `content.label`; content lint in stories.

Recommended practice.

☐ **Overflow behavior is declared**

04.07

SCHEMA

TEST

Truncate, wrap or grow, and whether truncated text stays in the accessible name or a tooltip.

Evidence: `content.overflow`, `content.fullTextExposure`; a story with a 200-character label.

Recommended practice.

☐ **Every string is a prop, and RTL is decided**

04.08

SCHEMA

TEST

Including internal strings such as a loading label or a close button. State whether the component mirrors in right-to-left languages.

Evidence: `i18n.strings[]`, `i18n.rtl`; a check for hard-coded English in the source; a `dir=rtl` visual mode.

Carbon requires all strings configurable via props. S03.

☐ **The visible label is part of the accessible name**

04.09

TEST

WCAG A

Voice-control users say what they see. An `aria-label` that replaces a visible label must contain it.

Evidence: A test comparing visible text with the computed accessible name.

WCAG 2.2 SC 2.5.3 Label in Name, A. S05.

Describe the allowed content once, as a list a machine can check.

SECTION 05

Put the accessibility contract in writing.

Accessibility minimums are part of the API, not a review step after it. Each item names the success criterion it traces to.

Suggested owners: Accessibility lead + engineering lead

☐ The APG pattern is linked, or its absence explained

05.01 **SCHEMA**

Point to the WAI-ARIA Authoring Practices pattern the component implements, or state that it is a native element with no pattern.

Evidence: `accessibility.apgPattern`.

S06.

☐ Role, name source and ARIA are listed

05.02 **SCHEMA** **WCAG A**

Which role, where the accessible name comes from and which ARIA states and properties apply.

Evidence: `accessibility.role`, `.name`, `.aria[]`; role and name assertions in interaction tests.

WCAG 2.2 SC 4.1.2 Name, Role, Value, A. S05.

☐ The keyboard table is in the contract

05.03 **SCHEMA** **TEST** **WCAG A**

One row per key, matching APG. Button: Enter and Space activate. Tabs: arrow keys move between tabs.

Evidence: `accessibility.keyboard[]`; one play-function test per row.

WCAG 2.2 SC 2.1.1 Keyboard, A. S05, S06.

☐ No keyboard traps

05.04 **TEST** **WCAG A**

Tab moves in and out. Modal focus containment is deliberate and escapable.

Evidence: An interaction test that tabs through and out.

WCAG 2.2 SC 2.1.2 No Keyboard Trap, A. S05.

☐ Focus management is specified

05.05 **SCHEMA** **WCAG A**

Initial focus, where focus returns on close and the focus order. Never a positive `tabindex`.

Evidence: `accessibility.focus`; interaction tests.

WCAG 2.2 SC 2.4.3 Focus Order, A. S05.

☐ The focus indicator is visible, and aims higher

05.06 **TEST** **WCAG AA**

Visible focus is the AA minimum. As a system target, meet Focus Appearance: a ring at least as large as a 2 CSS px perimeter with 3:1 change against the unfocused state.

Evidence: A focus-visible story; the focus ring token recorded in `accessibility.focus.indicatorToken`.

WCAG 2.2 SC 2.4.7 (AA) and SC 2.4.13 Focus Appearance (AAA). S05.

☐ **Focus is never entirelyly hidden**

05.07

TEST

WCAG AA

Sticky headers and overlays the component creates must not fully cover the focused element.

Evidence: An interaction test in a layout story with a sticky header.

WCAG 2.2 SC 2.4.11 Focus Not Obscured (Minimum), AA. S05.

☐ **Focus and input never change context by themselves**

05.08

TEST

WCAG A

Tabbing to a control or changing a value does not navigate or submit.

Evidence: An interaction test asserting no navigation or submit on focus or change.

WCAG 2.2 SC 3.2.1 On Focus and 3.2.2 On Input, A. S05.

☐ **Form controls carry labels and linked errors**

05.09

SCHEMA

TEST

WCAG A

A label is required; errors are identified in text, linked to the control and paired with a suggestion slot.

Evidence: A required-label rule in propsSchema; axe `label`; a test that the error is referenced by `aria-describedby`.

WCAG 2.2 SC 3.3.1 and 3.3.2 (A), 3.3.3 (AA). S05.

☐ **Axe fails the build, and people test too**

05.10

TEST

STABLE GATE

Every story runs axe with violations as failures, including open and expanded states. Screen-reader passes are recorded with the AT, browser and date.

Evidence: Storybook `parameters.a11y.test: 'error'; accessibility.manualAT[]`. The schema requires `a11yTest: "error"` for stable.

S11. Carbon requires JAWS, VoiceOver and NVDA passes for stable. S03.

If the keyboard table is not in the contract, it is not in the component.

SECTION 06

Make it legible to agents.

Agents read whatever your pipeline exposes: JSDoc through a Storybook manifest, a contract through an MCP server, examples through retrieval. Put the load-bearing guidance there.

Suggested owners: Design-system lead + docs owner + engineering lead

☐ **Visual values come from tokens, and the tokens are listed**

06.01 **SCHEMA** **TEST**

No raw colors or magic numbers in the component. The contract lists every token it uses.

Evidence: `theming.tokens[]` as DTCG aliases; a token lint rule (Field Guide 03).

Atlassian `ensure-design-token-usage`. S16.

☐ **Theming hooks, and what is locked**

06.02 **SCHEMA**

List the supported hooks (custom properties, parts, modes) and what consumers must not override: focus ring, minimum target size, label contrast.

Evidence: `theming.cssProperties[]`, `.modes[]`, `.locked[]`; visual coverage per mode.

Custom Elements Manifest `cssProperties`, `cssParts`, `cssStates`. S08.

☐ **Canonical examples come first**

06.03 **SCHEMA** **AGENT** **STABLE GATE**

At least one copy-pasteable example per common use, marked canonical and ordered first. Storybook's manifest shows the first three stories in full.

Evidence: `examples[]` with `canonical: true` and `storyId`; a story-order check.

S09.

☐ **Anti-patterns show the fix**

06.04 **SCHEMA** **AGENT** **STABLE GATE**

Each known mistake pairs the bad code, the good code and the reason. Use the mistakes agents actually make: invented variants, missing names, navigation in a Button.

Evidence: `antiPatterns[]` (`bad`, `good`, `reason`).

S10, S18.

☐ **Nothing load-bearing lives only in prose**

06.05 **AGENT** **TEST**

Guidance an agent needs is in JSDoc, the manifest or the contract, not only in an MDX page or a Figma annotation. Lint the manifest the agent reads: descriptions present, required props documented, docgen extraction succeeded.

Evidence: Manifest lint in CI.

Rachel Cantor documents agents reading a different system from the one in the docs site. S18.

☐ **Deprecations are machine-readable**

06.06

SCHEMA

AGENT

Deprecated props and components carry the version, the reason and the replacement, and the `@deprecated` tag survives into the manifest.

Evidence: `deprecated` object; manifest lint that the tag reaches the output.

S09, S18.

☐ **Agent instructions say: verify, never invent**

06.07

AGENT

PRACTICE

Your AGENTS.md or equivalent tells agents to look up props through the design-system MCP or CLI before use and never to invent props or variants. Field Guide 04 has a ready-made block.

Evidence: The instruction file exists and names the tools.

Storybook's MCP docs recommend checking every property with its docs tools. S10.

Write for the reader that cannot ask a follow-up question.

SECTION 07

Version the contract like an API.

A contract that changes silently is worse than none. Give it a version, a deprecation path and explicit promotion criteria.

Suggested owners: Design-system lead + release owner

☐ **The contract has its own semantic version**

07.01 **SCHEMA**

Removing or renaming a prop is a major bump; adding an optional prop is a minor bump.

Evidence: `contractVersion`; a CI diff against the previous release that classifies the change.

Recommended practice.

☐ **Deprecate before you remove**

07.02 **SCHEMA** **PRACTICE**

State a minimum deprecation window and the release in which removal is planned.

Evidence: `deprecated.removalPlannedIn`; the release checklist.

Carbon: always deprecate before removal, with long deprecation periods. S03.

☐ **Deprecated usage is loud**

07.03 **TEST**

A dev-mode warning and a lint error, so neither a person nor an agent keeps using it quietly.

Evidence: A unit test for the console warning; a lint rule such as Atlassian `no-deprecated-apis`.

Primer shows deprecation warnings to consumers. S04, S16.

☐ **Breaking changes ship with a migration path**

07.04 **SCHEMA**

A codemod where the change is mechanical, a migration guide where it is not.

Evidence: `deprecated.codemod`; a codemod test fixture.

Carbon ships codemods for migrations. S03.

☐ **Stable means evidence, not consensus**

07.05 **STABLE GATE**

Promotion to stable is a passing build: the schema requires states, constraints, sizes, canonical examples, anti-patterns and evidence links, with axe set to fail.

Evidence: The `allOf` stable gate in `component-contract.schema.json`.

Kit schema.

Stable is a state the build can prove.

SECTION 08

Enforce it in CI.

Each earlier item named its evidence. This section is the pipeline that runs it on every pull request.

Suggested owners: Engineering lead + design-system lead

☐ Contracts validate on every pull request

08.01 **TEST**

Every `*.contract.json` validates against the contract schema, and propsSchema compiles.

Evidence: `node validate-contracts.mjs src/components` in CI.

Kit script. S07.

☐ Contract and code props cannot drift

08.02 **TEST**

A bidirectional diff between the contract and the props your docgen extracts. A prop added in code without the contract fails, and so does the reverse.

Evidence: A script over the Storybook manifest (`reactDocgen.props`) or Custom Elements Manifest.

S08, S09.

☐ Type tests run

08.03 **TEST**

Every forbidden combination and required-together rule has a `@ts-expect-error` line that must keep failing.

Evidence: `tsc --noEmit` or `vitest --typecheck` over `*.test-d.ts`.

S17. Kit: `button.test-d.ts`.

☐ Interaction tests cover keyboard and focus

08.04 **TEST**

One test per keyboard row and focus rule in the contract.

Evidence: Storybook play functions run through the Vitest addon or test runner.

S11.

☐ Visual tests cover the allowed matrix in every mode

08.05 **TEST**

Every allowed variant, size and state, in each theme mode and at 320 px.

Evidence: Chromatic or Percy with modes; the matrix story generated from the contract.

S21. Kit: `allowed-combinations.mjs`.

☐ Design parity is checked on a schedule

08.06 **TEST**

Figma component properties (variant, boolean, text, instance swap, slot) compared with the contract.

Evidence: A scheduled job over Code Connect maps.

S13, S14.

☐ **Agents are evaluated, not assumed**

08.07

TEST

AGENT

A fixed set of UI tasks is generated with your system and graded automatically. A regression blocks the docs or contract change that caused it.

Evidence: Eval results stored per release.

Storybook and Atlassian both report running such benchmarks; their figures are vendor-reported. S10, S15.

The deliverable is a failing build when the contract is broken, not a document that says it is not.

APPENDIX A

The contract file.

`component-contract.schema.json` defines the format; `button.contract.json` is a complete, valid example. The props schema inside each contract is itself JSON Schema, so any proposed prop object can be checked with an off-the-shelf validator.

Field	Answers	Checklist items
<code>name</code> , <code>import</code> , <code>summary</code>	What is it and how do I import it?	01.01 to 01.03
<code>whenToUse</code> , <code>whenNotToUse</code>	Should I use it here, or what instead?	01.04
<code>props</code>	What can I pass, with which defaults?	01.06 to 01.10
<code>propsSchema</code> , <code>constraints</code>	Which combinations are allowed?	02.01 to 02.05
<code>sizes</code> , <code>states</code> , <code>layout</code>	How big, in which states, how wide?	03.01 to 03.08
<code>slots</code> , <code>composition</code> , <code>content</code> , <code>i18n</code>	What goes inside, and what text rules apply?	04.01 to 04.09
<code>accessibility</code>	Role, name, keyboard, focus, WCAG trace	05.01 to 05.10
<code>theming</code> , <code>examples</code> , <code>antiPatterns</code>	What an agent needs to use it correctly	06.01 to 06.06
<code>contractVersion</code> , <code>status</code> , <code>deprecated</code>	Can I depend on it, and what replaces it?	07.01 to 07.05
<code>evidence</code>	Where is the proof?	08.01 to 08.07

examples/button.contract.json (excerpt)

```
{
  "name": "Button",
  "import": "import { Button } from '@acme/ui';",
  "status": "stable",
  "props": {
    "tone": { "type": "'primary' | 'secondary' | 'ghost' | 'destructive'",
              "enum": ["primary", "secondary", "ghost", "destructive"],
              "default": "primary", "required": false, "matrix": true,
              "description": "Visual weight and intent." },
  },
  "propsSchema": {
    "type": "object",
    "additionalProperties": false,
    "allOf": [
      { "$comment": "ghost-never-full-width",
        "if": { "properties": { "tone": { "const": "ghost" } }, "required": ["tone"] },
        "then": { "not": { "properties": { "fullWidth": { "const": true } },
                      "required": ["fullWidth"] } } }
    ]
  },
  "accessibility": {
    "apgPattern": "https://www.w3.org/WAI/ARIA/apg/patterns/button/",
    "role": "button",
    "keyboard": [ { "key": "Enter", "action": "Activates the Button." },
                  { "key": "Space", "action": "Activates the Button." } ]
  }
}
```

The full file also covers anatomy, sizes, states, slots, content, theming, examples, anti-patterns and evidence.

APPENDIX B

Generate only what the contract allows.

Automated variant generation shipped twelve unusable button states at one client, because the generator produced every permutation of size, variant and state. The fix is to generate from the contract.

terminal

```
$ node scripts/allowed-combinations.mjs examples/button.contract.json
Button: matrix props tone, size, iconOnly, loading, disabled, fullWidth
192 combinations in the cross-product, 99 allowed by the contract.
  rejected by loading-or-disabled: 48
  rejected by icon-only-is-intrinsic: 48
  rejected by ghost-never-full-width: 24
```

Counts per rule overlap: some combinations break more than one rule. Output from the kit, not a mock-up.

Button.stories.tsx

```
// node allowed-combinations.mjs button.contract.json --json > button.combinations.json
import combinations from "./button.combinations.json";
import { Button } from "@acme/ui";

export const Matrix = {
  render: () => (
    <div style={{ display: "grid", gap: 12, gridTemplateColumns: "repeat(4, max-content)" }}>
      {combinations.map((props, i) => (
        <Button key={i} aria-label="Save changes" {...props}>Save changes</Button>
      ))}
    </div>
  ),
};
```

One matrix story, covered by visual tests in every mode (item 08.05).

examples/button.types.ts (excerpt)

```
/** A ghost Button has no container, so it is never full width. */
type Emphasis =
  | { tone?: ButtonTone; fullWidth?: false }
  | { tone?: Exclude<ButtonTone, "ghost">; fullWidth?: boolean };

/** Either prop may be a runtime boolean, as long as the other one is off. */
type Activity =
  | { loading?: boolean; disabled?: false }
  | { loading?: false; disabled?: boolean };

// button.test-d.ts
// @ts-expect-error ghost-primary is not a tone
export const inventedTone: ButtonProps = { tone: "ghost-primary" };
export const saving: ButtonProps = { loading: isSaving, children: "Save" }; // compiles
```

The same rules as the schema, enforced by the compiler (items 02.03 and 08.03).

.github/workflows/contracts.yml (steps)

```
- run: npm ci
- name: Validate contracts
  run: node scripts/validate-contracts.mjs src/components
- name: Type tests (forbidden combinations must not compile)
  run: npx tsc --noEmit -p tsconfig.json
- name: Generate matrix data for visual tests
  run: node scripts/allowed-combinations.mjs src/components/button/button.contract.json --json
> src/components/button/button.combinations.json
```

Adapt paths to your repo. Run your Storybook interaction, axe and visual tests after these steps.

KEEP WITH THE COMPONENT

Leave a review record.

A record for one component review. It is not a certification; it is the trail that lets the next reviewer see what was proven and what was waived.

Component / contract version	Status requested (beta, stable)
Owner (design) / owner (engineering)	Contract validation run (link)
Allowed combinations (count) / matrix story	Type tests / interaction tests (links)
Axe and visual runs, all modes (links)	Manual screen-reader pass (AT, browser, date)
Waived items and reasons	Next review trigger

Do not waive what the schema enforces.
If a stable-gate field is missing, the component is not stable yet. Change the status, not the schema.

SOURCES / MAINTENANCE

Keep the guide current.

Sources checked 24 September 2026. Vendor figures are cited as vendor-reported. The contract field names are a recommendation that maps to these formats; no single standard defines them.

S01 / Nathan Curtis, Component Contracts and Schemas

Definitions of contract and schema; seven principles (July 2026).

<https://nathanacurtis.substack.com/p/component-contracts-and-schemas>

S02 / Nathan Curtis, Components as Data

Anatomy, props, variants and styles as a data model (2025).

<https://nathanacurtis.substack.com/p/components-as-data-2be178777f21>

S03 / IBM Carbon, component checklist

Definition of done for preview and stable components.

<https://carbondesignsystem.com/contributing/component-checklist/>

S04 / GitHub Primer, component lifecycle

Experimental, ready and deprecated; deprecation warnings.

<https://primer.style/design/guides/component-lifecycle/>

S05 / W3C, WCAG 2.2

Recommendation, 12 December 2024. Success criteria and levels cited in section 05.

<https://www.w3.org/TR/WCAG22/>

S06 / W3C, ARIA Authoring Practices Guide patterns

Keyboard interaction and roles per pattern.

<https://www.w3.org/WAI/ARIA/apg/patterns/>

S07 / JSON Schema 2020-12, conditionals

if/then/not and dependentRequired, used for forbidden and required-together combinations.

<https://json-schema.org/understanding-json-schema/reference/conditionals>

S08 / Custom Elements Manifest

custom-elements.json: attributes, slots, events, CSS properties, parts and states.

<https://github.com/webcomponents/custom-elements-manifest>

S09 / Storybook, component manifests

What agents read: props from docgen, JSDoc descriptions, the first three stories in full.

<https://storybook.js.org/docs/ai/manifests>

S10 / Storybook, MCP server

Docs, story and test tools for agents; guidance against invented props. Preview feature.

<https://storybook.js.org/docs/ai/mcp/overview>

S11 / Storybook, accessibility testing

parameters.a11y.test set to error fails stories on axe violations.

<https://storybook.js.org/docs/writing-tests/accessibility-testing>

S13 / Figma, Code Connect

Maps Figma properties to code props; feeds the Figma MCP server.

<https://developers.figma.com/docs/code-connect/>

S14 / Figma, slots (plugin API update)

SLOT component property, generally available 10 June 2026.

<https://developers.figma.com/docs/plugins/updates/2026/06/10/update/>

S15 / Atlassian, Teaching AI to speak our design language

Agent failure modes and the ADS MCP server; figures are vendor-reported.

<https://www.atlassian.com/blog/ai-at-work/teaching-ai-to-speak-our-design-language>

S16 / Atlassian, ESLint plugin for the design system

ensure-design-token-usage, no-deprecated-apis and related rules.

<https://atlassian.design/components/eslint-plugin-design-system/ensure-design-token-usage/>

S17 / Vitest, testing types

expectTypeOf, @ts-expect-error and *.test-d.ts files.

<https://vitest.dev/guide/testing-types>

S18 / Rachel Cantor, Your agent is reading a different design system

Guidance lost between docs and manifests (July 2026).

<https://rachel.fyi/posts/your-agent-is-reading-a-different-design-system>

S19 / WebAIM Million 2026

Missing form labels 51%, empty buttons 30.6% of home pages.

<https://webaim.org/projects/million/>

S21 / Chromatic documentation

Visual, interaction and accessibility tests with modes.

<https://www.chromatic.com/docs/>

Maintenance: recheck the Storybook MCP and manifest pages at each Storybook minor release (tool names are marked preview), and update the WCAG references if WCAG 3 reaches Recommendation. Update the PDF, HTML, Markdown and JSON together.