Systems Thinking and Token Naming

What Is Systems Thinking?

Before connecting it to naming, it helps to understand what systems thinking actually means.

Systems thinking is a way of seeing the **whole** rather than just the parts. It asks:

- How do the parts relate to each other?
- What are the rules that govern those relationships?
- What happens downstream when one part changes?
- How does the system behave over time?

The opposite of systems thinking is **local thinking** — solving the immediate problem in front of you without considering how that solution affects everything connected to it.

> *"In everyday terms: systems thinking is the difference between fixing a leak and understanding the plumbing."*

Why Naming Is a Systems Problem

Most people approach naming as a **creative or linguistic problem**. They ask:

"What should I call this?"

Systems thinking reframes the question as:

"How does this name need to behave within the larger system it lives in?"

This is a fundamental shift. Here's why it matters:

Names Don't Exist in Isolation

Every token name exists in relationship to:

- Other tokens in the same system

- The component that consumes it
- The tier above and below it in the token hierarchy
- The platform or tool rendering it
- The human or LLM trying to locate it

When you name a token in isolation — without considering those relationships — you create a name that might be locally logical but **globally confusing**.

Example:

You create a token called `blue`. It makes perfect sense to you right now. But:

- What happens when you add a second shade of blue? (`blue-2`?)
- What happens when blue becomes your action color AND your link color?
- What happens when a new brand variant uses teal instead of blue?

The name `blue` has no system. It can't scale. It breaks the moment anything around it changes.

The Three Properties of a Well-Functioning System

Systems thinking teaches us that healthy systems share three properties. Each one has a direct parallel in token naming.

1. Interconnectedness

Parts of a system are connected and affect each other.

In token naming:
Token names should form a **connected vocabulary**. A change at one tier (say, renaming a primitive) should have a predictable, traceable effect on the tiers that reference it. If names aren't connected by a shared logic, changes become unpredictable.

---

2. Hierarchy and Structure

Systems have levels of organization. Each level has its own role.

In token naming:
This is exactly what the three-tier structure provides. Primitive tokens live at one level, semantic tokens at another, component tokens at a third. Each level has a specific job:

- Primitives **store raw values**
- Semantic tokens **assign meaning**
- Component tokens **apply decisions**

Mixing the roles of these levels — putting meaning in primitive names, or raw values in semantic names — breaks the hierarchy and makes the system unpredictable.

---

3. Feedback Loops

Systems have mechanisms that signal when something is wrong.

In token naming:
A naming system creates its own feedback loop. When naming is working, designers find what they need quickly, engineers don't question token choices, and documentation stays current. When it's broken, the feedback is: confusion, workarounds, and people asking for hex codes.

> *"Recognizing confusion as **feedback from the system** — not just a people problem — is a core systems thinking skill."*

The Fundamental Principle: Names Are Rules, Not Labels

This is perhaps the most important shift systems thinking brings to naming.

When you think locally, a name is a **label** — it identifies a specific thing.

When you think systematically, a name is a **rule** — it encodes where something lives in the system, what it does, and how it relates to everything else.

Consider these two token names:

```
#1:  blue-500

#2:  color-background-action-primary-default
```

`blue-500` is a label. It tells you what the value *is*.

`color-background-action-primary-default` is a rule. It tells you:

- What category it belongs to (`color`)
- Where it applies (`background`)
- What role it plays (`action`)
- Its position in emphasis hierarchy (`primary`)
- What state it represents (`default`)

The second name is longer — but it is **locatable**. Someone who has never seen your design system before can make a reasonable guess about what that token does and where it belongs.

Systems Thinking Applied: The Four Questions

Before naming any token, systems thinking asks you to answer four questions. These are your orientation points.

Question 1: Where does this token live in the hierarchy?

Is this a primitive, a semantic alias, or a component-scoped token? The answer determines what kind of name it should have.

- Primitives describe **what the value is**
- Semantic tokens describe **what the value means**
- Component tokens describe **where and how the value is applied**

Question 2: What is this token's job?

Every token has a single responsibility. A token that tries to do two things creates ambiguity.

- Is it storing a value?
- Is it expressing a role (action, feedback, brand)?
- Is it expressing a state (hover, disabled, focus)?
- Is it scoping a decision to a specific component?

The name should reflect the job — not the current value.

Question 3: How does this token relate to adjacent tokens?

Tokens rarely exist alone. They belong to families:

- A scale family (size-1, size-2, size-3...)
- An emphasis family (primary, secondary, tertiary...)
- A state family (default, hover, active, disabled...)

Systems thinking asks: *does this name make sense in the context of its family?* If you can't read the family together and understand the progression, the naming system has broken down.

Question 4: What changes, and what stays stable?

Good token names are stable even when values change. This is one of the most practical tests of systematic naming.

Ask yourself: *if the underlying value of this token changed tomorrow, would the name still be accurate?*

- `color-action-primary` → still accurate if the action color changes from blue to teal ✓
- `blue-button-background` → immediately broken if the button color changes ×

Systematic names survive value changes. Label-based names don't.

The Mental Model: Names as Coordinates

The workshop introduces a powerful mental model worth dwelling on:

> *“"Think of names as coordinates that map to design decisions within your system."”*

This is worth unpacking carefully.

A coordinate system works because:

- Every point has a **unique address**
- The address is **constructed from known axes**
- Anyone who knows the axes can **locate any point** without a map
- New points can be **added predictably** without breaking existing ones

Applied to token naming, the "axes" of your coordinate system might be:

Axis	Examples
Category	color, space, typography, border, shadow
Role / Usage	background, foreground, border, action, feedback
Emphasis	primary, secondary, tertiary, subtle
State	default, hover, active, focus, disabled
Scale	100, 200, 300 or xs, sm, md, lg, xl

A token name is built by selecting values along these axes that accurately describe the decision.

> *“When your axes are well-defined and consistently applied, naming stops being a creative act and becomes a **systematic one**. You're filling in coordinates, not inventing descriptions.”*

What Breaks Down Without Systems Thinking

It's useful to see the failure modes clearly, because most of us have inherited or created systems with these problems.

Failure Mode 1: Naming from the value

`red`, `#FF0000`, `blue-500`

These names will break the moment values change. They also don't tell you *why* something is that color.

Failure Mode 2: Naming from a specific use case

`checkout-button-color`, `nav-link-hover`

These names don't scale. What happens when the same color appears in a non-checkout context? You either duplicate the token or create a name that lies.

Failure Mode 3: Naming without hierarchy awareness

Using the same naming pattern for primitives and semantic tokens creates confusion about which tier you're working in and what should reference what.

Failure Mode 4: Naming by committee without a system

Ten people vote on names. Each person uses different mental models. The result is a mix of conventions that feels inconsistent because *it is* — it reflects ten different local decisions rather than one systematic approach.

The Practical Takeaway Before Moving Forward

Systems thinking, applied to naming, gives you this foundation:

1. **Names are rules, not labels** — encode meaning and position, not just identity
2. **Names are coordinates** — built from consistent axes your whole team understands
3. **Names must survive change** — stable when values shift, only changing when meaning changes
4. **Names serve the whole system** — not just the person creating them right now

5. **Confusion is feedback** — when people can't find or understand tokens, the system is telling you something

---

Where This Leads

With this foundation in place, the next step is understanding **component anatomy and naming** — how to decompose a component into its parts, name those parts consistently, and align that naming between design tools and code.

That's where the systems thinking principles you've just learned get applied to something concrete and hands-on.

Layer 2 — Component Anatomy and Naming

Starting Point: What Is Component Anatomy?

Component anatomy is the practice of **deliberately decomposing a component into its named parts** before you build it, name it, or token it.

Think of it like an anatomical diagram. A human body isn't just "a person" — it has a skeletal system, a circulatory system, organs, tissues, cells. Each level has a name, a function, and a relationship to the levels above and below it.

A component works the same way. A button isn't just "a button." It has:

- A container
- A label
- An icon (optional)
- States (default, hover, active, focus, disabled)
- Variants (primary, secondary, destructive)

Naming the anatomy before naming the tokens is what makes token naming systematic rather than ad hoc.

Why Anatomy-First Matters for Naming

Without a clear anatomy, naming decisions get made **at the surface** — you name what you can see, not what the system actually needs.

The result is token names that:

- Reflect the visual appearance rather than the structural role
- Can't be reused across components with similar parts
- Break down when the component evolves
- Don't match what engineers call the same parts in code

> *"**The core principle:** If designers and engineers don't share a vocabulary for component parts, every handoff is a translation exercise. Anatomy-first naming eliminates that translation layer."*

The Four Component Types

The workshop framework identifies four levels of component complexity. Understanding which type you're working with shapes how you name it and its parts.

Type 1: Simple Components

What they are:
Single-purpose components with a minimal set of parts. They don't contain other components.

Characteristics:

- One clear job
- Limited anatomy
- Few variants
- Directly mappable to a single HTML element or small group of elements

Examples:

- Badge
- Avatar
- Divider
- Tag/Chip
- Icon

Naming implication:
Simple components have shallow anatomy. Token names tend to be straightforward because there aren't many parts to differentiate.

```
badge-background-default
badge-text-default
badge-border-default
```

Type 2: Sophisticated Components

What they are:
More complex single components with richer anatomy — more parts, more states, more variants — but still functioning as one unified unit.

Characteristics:

- One clear job, done with more nuance
- Richer anatomy (multiple named sub-parts)
- Multiple states and variants
- Still a single component, not assembled from others

Examples:

- Button (with icon, label, loading state)
- Input field (with label, helper text, error state, prefix/suffix)
- Tooltip
- Toggle / Switch

Naming implication:
Sophisticated components require you to be precise about part names because multiple parts exist within one component. The anatomy mapping becomes critical here.

```
input-label-text-default
input-field-background-default
input-field-background-focus
input-helper-text-default
input-error-text-default
input-border-default
input-border-focus
input-border-error
```

Notice how the **part name** (`label` , `field` , `helper` , `error` , `border`) sits between the **component name** and the **property/state**. This is anatomy-driven naming.

Type 3: Compound Components

What they are:
Components assembled from multiple simple or sophisticated components working together as a coordinated unit.

Characteristics:

- Made of other components
- The sub-components retain their own identity
- The compound component adds compositional logic on top
- Often has its own layout and spacing concerns

Examples:

- Form (composed of inputs, labels, buttons)
- Card (composed of image, heading, body text, action)
- Navigation bar (composed of logo, links, button, avatar)
- Modal (composed of header, body, footer, close button)

Naming implication:
Compound components introduce a **scoping challenge**. The sub-components inside them may need tokens scoped specifically to that context — because a button inside a card might behave differently than a standalone button.

```
card-container-background-default
card-header-text-default
card-body-text-default
card-action-background-default     ← scoped to card context
card-action-background-hover       ← not the same as button-background-hover
```

Type 4: Composed Components

What they are:
Higher-order assemblies built from compound components. These are full interface patterns rather than individual components.

Characteristics:

- Made of compound components
- Often represent a complete UI section or feature
- May be page-level or feature-level patterns
- Sometimes called "templates" or "patterns" in design systems

Examples:

- Hero section
- Data table with pagination and filters
- Settings page layout
- Onboarding flow

Naming implication:
At this level, token naming typically doesn't go deeper than the compound components inside. You're more likely to be naming the **pattern** itself and documenting its compositional rules than creating new component-scoped tokens.

The Anatomy Mapping Process

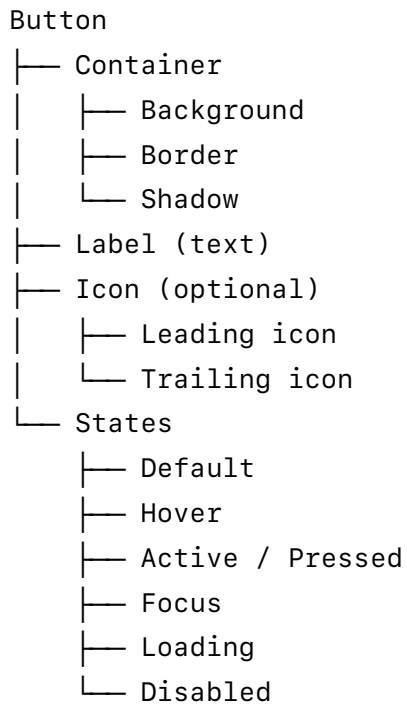
This is the hands-on skill the workshop builds. Here's how it works systematically.

Step 1: Draw the Anatomy

Before writing a single token name, map the component visually. Identify every distinct part that:

- Has a visual property that could change
- Has a state that differs from other states
- Could be independently modified by a design decision

For a **button**, this might look like:



Step 2: Name the Parts Consistently

Once you have the anatomy mapped, assign names to each part. The goal is names that:

- Are **structurally descriptive** (what is this part's role?)
- **Match engineering terminology** (what does the developer call it?)
- Are **reusable across components** where the same part type appears

Common part names and their meanings:

Part Name	What It Refers To
`container`	The outermost wrapping element
`surface`	A background plane (often interchangeable with container)
`label`	Text that identifies or names something
`description`	Supporting or helper text
`icon`	An icon element within the component
`leading`	An element positioned at the start
`trailing`	An element positioned at the end
`overlay`	A layer placed on top of content
`track`	The background element of a slider or toggle
`thumb`	The interactive handle of a slider or toggle
`indicator`	A visual marker of status or selection

> *"**Key principle:** Use the same part name across components whenever the structural role is the same. An icon in a button and an icon in an input field are both `icon`. Consistent part names create a consistent vocabulary."*

Step 3: Map Design Properties to Parts

Each part has **design properties** — the visual characteristics that can be expressed as tokens. Systematically mapping these prevents gaps and ensures you're not missing any tokenizable decision.

Common design properties:

Property	What It Controls
`background`	Fill color of a surface or container
`foreground`	Text or icon color on a surface
`text`	Typography-specific color
`border`	Stroke color
`border-width`	Stroke thickness
`border-radius`	Corner rounding
`shadow`	Drop shadow or elevation
`opacity`	Transparency
`spacing`	Internal padding or gap
`size`	Width, height, or both

Step 4: Layer in States

States are the conditions under which a part's design properties change. They're the last layer of the anatomy mapping, applied on top of parts and properties.

Common states:

State	When It Applies
`default`	The resting, uninteracted state
`hover`	Cursor is over the element
`active`	Element is being clicked/pressed
`focus`	Element has keyboard focus
`focus-visible`	Focus ring visible (keyboard nav)
`disabled`	Element cannot be interacted with
`loading`	Async action is in progress
`selected`	Element is in a chosen/on state
`checked`	Specific to checkboxes and radios
`error`	Validation has failed
`success`	Validation has passed
`readonly`	Visible but not editable

Putting It Together: The Naming Formula

With anatomy mapped, the token naming formula becomes:

```
[component]-[part]-[property]-[state]
```

Applied to a button:

```
button-container-background-default
button-container-background-hover
button-container-background-active
button-container-background-disabled
button-container-border-default
button-label-text-default
button-label-text-disabled
button-icon-foreground-default
```

This formula is **predictable and generative**. Once your team knows the axes, anyone can construct or locate a token name without asking.

Aligning Design Tools With Code

This is where the anatomy-first approach pays off most concretely.

The Alignment Problem

In Figma, designers often name layers and properties based on visual appearance:

- "Blue Fill"
- "Large Text"
- "Rounded Box"

In code, engineers name things based on function and structure:

- `backgroundColor`
- `fontSize`
- `borderRadius`

These two vocabularies don't connect. Every handoff requires translation.

The Solution: Anatomy as Shared Language

When both the design file and the codebase use the same **anatomy vocabulary**:

- A Figma component named `Button / Primary / Default` maps directly to a code component `Button` with `variant="primary"` and no state modifier
- A Figma layer named `container` maps to the wrapping `div` or equivalent in code
- A Figma token named `button-container-background-hover` maps to the CSS custom property `--button-container-background-hover`

The anatomy becomes the **contract** between design and development.

A Practical Exercise to Try

Take any component you work with regularly and apply the anatomy mapping process:

1. **List every visible part** of the component
2. **Name each part** using structural, role-based language
3. **List the design properties** for each part
4. **List the states** that affect each part
5. **Construct token names** using the formula: `[component]-[part]-[property]-[state]`

Then ask yourself:

- Do any of these names already exist in your system under a different name?
- Are there parts you named differently here than you have elsewhere?
- Are there any names that would confuse an engineer? A designer who didn't create this?

The gaps and inconsistencies you find are exactly where a naming system adds value.

What This Layer Has Built

You now have the tools to:

- Classify components by type (simple, sophisticated, compound, composed)
- Decompose any component into its anatomy
- Name parts consistently using structural, role-based vocabulary
- Map design properties and states onto those parts
- Construct token names from a predictable formula

Where This Leads

Layer 3 takes everything you've built here and moves it into the **three-tier token structure** — where the component-level names you've just learned to construct sit within a larger system of primitive and semantic tokens that feed into them.

That's where the full picture comes together.

Layer 3 — The Three-Tier Token Structure

Why Three Tiers?

At this point in the workshop, you've internalized two foundational ideas:

1. Names are **rules, not labels** — they encode position and meaning within a system
2. Component names should reflect **anatomy** — the structural parts, properties, and states that define a component

Now comes the question those two ideas were always building toward: *where do tokens actually live, and what are they responsible for?*

The answer is the three-tier token structure. It's not a naming convention — it's an **architecture**. Each tier has a distinct job, and getting clear on those jobs is what makes the rest of the naming decisions fall into place.

The Three Tiers at a Glance

```
Tier 1: Global / Primitive Tokens
  → Raw values: what something IS
  → e.g., color-blue-500, space-16, font-size-14

Tier 2: Semantic / Alias Tokens
  → Purposeful meaning: what something MEANS
  → e.g., color-action-primary, space-component-padding

Tier 3: Component-Specific Tokens
  → Scoped application: where and how something is APPLIED
  → e.g., button-container-background-default, button-label-text-hover
```

Read those three jobs again carefully: **IS**, **MEANS**, **APPLIED**.

That progression is the entire architecture. Every naming decision in a well-structured token system can be traced back to one of those three jobs. If a token is trying to do more than one job, that's a signal it's in the wrong tier — or that a tier is missing.

Tier 1: Global / Primitive Tokens

What They Are

Primitive tokens are the **raw material** of your design system. They store a value — a specific color, a specific spacing unit, a specific font size — and nothing else.

They don't tell you what to do with the value. They don't express an opinion about context or meaning. They are the full universe of design decisions available to your system, laid out in one place.

Think of them as a **paint palette**. Every color is there. None of them know what they'll be used for yet.

What They're Named For

Primitive token names describe **what the value is** — specifically, where it falls within a scale.

```
color-blue-100
color-blue-200
color-blue-300
color-blue-400
color-blue-500
color-blue-600
color-blue-700
color-blue-800
color-blue-900
```

The name `color-blue-500` tells you: this is a color, it's in the blue family, and it sits at the midpoint of the scale. That's all it tells you — and that's intentional.

Other examples:

```
space-4
space-8
space-12
space-16
space-24
space-32

font-size-12
font-size-14
font-size-16
font-size-18
font-size-24

font-weight-400
font-weight-500
font-weight-700
```

What Primitives Should NOT Do

Primitives should never express **intent** or **context**. The moment a primitive name includes words like "action," "primary," "hover," or "button," it has crossed into semantic territory — and you've lost the clean separation that makes the architecture work.

❌ Avoid	✅ Prefer
`color-brand-primary`	`color-blue-500`
`color-action-blue`	`color-blue-600`
`space-button-padding`	`space-16`
`font-size-heading`	`font-size-24`

The test: *could this name remain accurate even if every tier-2 and tier-3 token that references it changed completely?* If yes, it's a good primitive name. If the name would become misleading, it has meaning baked in where it doesn't belong.

Tier 2: Semantic / Alias Tokens

What They Are

Semantic tokens are where **meaning enters the system**. They don't store raw values directly — they reference a primitive token and assign it a **role** within your design language.

This is the tier that turns `color-blue-500` into `color-action-primary`. The value doesn't change. The *meaning* does.

Think of semantic tokens as a **translation layer**. They translate raw palette decisions into design language decisions.

Why This Tier Is the Most Important

Semantic tokens are what make your system:

- **Themeable** — swap a theme by changing which primitive each semantic token points to, without touching a single component

- **Readable** — designers and developers can understand intent from the name alone
- **Resilient** — if your brand color changes from blue to teal, you update one primitive reference in the semantic tier, and every component that consumes `color-action-primary` updates automatically

Without semantic tokens, your tier-3 component tokens would reference primitives directly. That means every component would be tightly coupled to specific values — and every rebrand, mode switch, or theme variant would require hunting through every component token individually.

The semantic tier is the **abstraction** that makes scale possible.

The Four Semantic Naming Domains

The workshop focuses on four specific areas where semantic naming requires deliberate decisions. These are the places where naming conventions break down most often — and where getting them right pays the largest dividends.

Domain 1: Scales

Scales are ordered progressions of a value — spacing, sizing, border radius, font size, line height, z-index. Primitives store the raw scale values. Semantic tokens give those values **relational meaning**.

The challenge with scales is choosing names that express position in the progression without locking you into specific values.

Option A: Numeric scales

```
space-100
space-200
space-300
space-400
space-500
```

Numeric scales are flexible and easy to extend (you can insert `space-150` between `space-100` and `space-200`). They're especially common for color scales where the full range needs to be accessible.

Option B: T-shirt sizes

```
space-xs
space-sm
space-md
space-lg
space-xl
space-2xl
space-3xl
```

T-shirt sizes are more readable and more natural for teams without an engineering background. Their limitation is that the scale is harder to extend in the middle without resorting to awkward names.

Option C: Semantic role names

```
space-component-gap
space-component-padding
space-section-gap
space-page-margin
```

Role-based scale names move away from expressing position in a progression and instead express **where** the space is used. This is highly readable but more opinionated — and it requires more decisions upfront.

> *"**Practical guidance:** Most systems benefit from a combination. Use numeric or t-shirt scales for primitives and fine-grained control. Use role-based names at the semantic tier for the handful of spacing decisions that recur throughout the system (component internal padding, section gaps, layout margins)."*

Domain 2: Emphasis

Emphasis tokens express **hierarchy of importance** within a category. They answer the question: *among all the possible values for this property, which one should command the most attention?*

The most common emphasis vocabulary:

```
primary     → highest visual weight; the most important instance
secondary   → supporting; visible but not dominant
tertiary    → minimal; present but de-emphasized
subtle      → very low contrast; background-level
```

Applied to color:

```
color-text-primary          → main body text; high contrast
color-text-secondary        → supporting text; medium contrast
color-text-tertiary         → placeholder, captions; lower contrast
color-text-subtle           → nearly invisible; disabled or de-emphasized
text
```

Applied to surfaces:

```
color-surface-primary       → main background
color-surface-secondary     → card or elevated surface
color-surface-tertiary      → nested container
color-surface-subtle        → barely-there background tint
```

The naming trap to avoid: Don't use emphasis vocabulary if you're not building a true hierarchy. If your system only needs two levels, use `default` and `subtle` — not `primary` and `secondary` — because those imply a third and fourth level that doesn't exist.

> *"**Key principle:** Emphasis names are **relative**, not absolute. `primary` doesn't mean a specific value. It means "the most prominent choice in this context." As soon as someone asks "which hex is primary?" — that's a signal they haven't internalized the tier structure yet."*

Domain 3: Hierarchy

Hierarchy tokens express **structural relationships** — not visual weight, but **organizational level**. Where emphasis describes "how important," hierarchy describes "what level."

This shows up most clearly in typography:

```
font-size-display          → largest; for hero or headline moments
font-size-heading-1        → primary section heading
font-size-heading-2        → sub-section heading
font-size-heading-3        → tertiary heading
font-size-body             → standard reading size
font-size-caption          → smaller supporting text
font-size-label            → UI label text (forms, buttons)
font-size-overline         → small, uppercase category markers
```

And in layout/elevation:

```
shadow-level-0             → flat; no elevation
shadow-level-1             → subtle lift; cards
shadow-level-2             → floating elements; dropdowns
shadow-level-3             → modals; popovers
shadow-level-4             → top-level overlays; toasts
```

The distinction between hierarchy and emphasis:

Emphasis (`primary` , `secondary`) describes visual weight within the same structural level. Hierarchy (`heading-1` , `heading-2` , `body`) describes the level itself.

A heading-2 and a heading-3 are at different **hierarchy** levels. A primary heading and a secondary heading at the *same* level have different **emphasis**.

> *"Think of it this way: hierarchy is the **skeleton** (structure), emphasis is the **volume** (how loud each part speaks)."*

Domain 4: Conditional States

State tokens express how a token's value changes based on **interaction or system condition**. They're the semantic layer's answer to "what happens when a user touches this?"

The standard state vocabulary:

```
default         → resting state; no interaction
hover           → pointer is over the element
active          → element is being pressed/clicked
focus           → element has input focus
focus-visible   → focus ring visible (keyboard navigation only)
selected        → element is in a chosen/on state
checked         → boolean on-state for checkboxes and radios
disabled        → element cannot be interacted with
loading         → async action is pending
error           → validation failure
warning         → advisory, non-blocking issue
success         → positive confirmation
readonly        → visible but not editable
```

State tokens typically modify a base semantic token. Applied examples:

```
color-action-primary-default
color-action-primary-hover
color-action-primary-active
color-action-primary-disabled
color-action-primary-focus
```

Why states belong at the semantic tier (not just the component tier):

You might be tempted to only name states at the component level (`button-background-hover`). But if semantic tokens don't carry state, you lose the ability to express *shared behavioral rules* across components.

Consider: your system's action color should lighten on hover everywhere it appears — buttons, links, icon buttons, chips. If that hover behavior lives only in component tokens, you'll define it separately for each component. If it lives at the semantic tier as `color-action-primary-hover` , every component can reference the same rule.

State at the semantic tier = **shared interaction language**.

Tier 3: Component-Specific Tokens

What They Are

Component tokens are the **final application layer**. They scope a semantic decision to a specific component part in a specific state.

This is the tier you worked with extensively in Layer 2. The naming formula you learned there — `[component]-[part]-[property]-[state]` — is the tier-3 pattern.

```
button-container-background-default
button-container-background-hover
button-container-background-disabled
button-label-text-default
button-label-text-disabled
button-icon-foreground-default
```

What Component Tokens Reference

In a well-structured system, component tokens don't reference primitive tokens directly. They reference **semantic tokens**.

```
button-container-background-default → color-action-primary-default
button-container-background-hover   → color-action-primary-hover
button-label-text-default           → color-text-on-action-primary
```

This chain — component → semantic → primitive — is what gives you:

- Theming (swap the semantic-to-primitive mapping, and components change automatically)
- Readability (every component token tells you its *role*, not its *value*)
- Resilience (no component token breaks when primitive values change)

When to Create Component Tokens (and When Not To)

Not every component needs its own token set. Component tokens earn their existence when:

1. **The component has a genuinely unique visual behavior** that isn't captured by a semantic token
2. **The component needs to be independently themeable** within a larger system
3. **Multiple design decisions converge in one place** and the component token acts as a useful coordination point

If a component can simply consume semantic tokens directly without ambiguity, adding a layer of component tokens is overhead without benefit.

> *"**Rule of thumb:** Start by consuming semantic tokens in components. Add component-scoped tokens only when you encounter a decision that can't be expressed cleanly by an existing semantic token."*

The Full Chain in Practice

Here's the three-tier structure end-to-end for a single token decision — a primary button's background color on hover:

```
Tier 1 (Primitive):
  color-blue-600 = #1D4ED8

Tier 2 (Semantic):
  color-action-primary-hover → color-blue-600

Tier 3 (Component):
  button-container-background-hover → color-action-primary-hover
```

What each tier contributes:

- The **primitive** stores the raw value and nothing else
- The **semantic** token declares that this blue, at this shade, is what "action primary on hover" means in this system
- The **component** token declares that the button container's background, on hover, should follow that action-primary-hover rule

If the brand color changes from blue to violet, you update `color-action-primary-hover` to point to `color-violet-600` instead — and the button updates automatically, along with every other component that references `color-action-primary-hover`.

That is the payoff of the three-tier architecture.

Common Mistakes in the Three-Tier Structure

Mistake 1: Skipping Tier 2

The most common structural failure. Components reference primitives directly:

```
button-container-background-hover → color-blue-600   ×
```

This works until you need to change anything — a rebrand, a dark mode, a new product theme. Then you're updating component tokens individually instead of flipping one semantic mapping.

Mistake 2: Putting Semantic Meaning in Tier 1

```
color-action-primary = #1D4ED8   ✗  (this is a semantic name at the primitive tier)
```

Primitives that encode meaning lose the separation of concerns that makes the system scalable. You can't reuse `color-action-primary` as a primitive because it's already opinionated.

Mistake 3: Creating Redundant Tier 3 Tokens

```
button-primary-container-background-default → color-action-primary-default
button-secondary-container-background-default → color-action-secondary-default
```

If `primary` and `secondary` variants are already handled by semantic tokens, you don't need to split them again at the component tier. Consume the semantic token with a variant prop or modifier instead.

Mistake 4: Inconsistent Tier Depth

Some components have three tiers; others have two; others have one. The system becomes unpredictable because the rules for when to go deep and when not to are undocumented.

> *"This is why documentation matters — not as an afterthought, but as the mechanism that communicates the rules of the system to everyone using it."*

What You Now Have

With all three tiers in place, you have a complete architecture:

- **Tier 1** gives you a controlled palette of raw values
- **Tier 2** gives you a shared vocabulary of meaning — the design language your whole system speaks
- **Tier 3** gives you precise, anatomy-driven names that apply those decisions at the component level

The system is now **navigable**. A designer can trace a component token back to its semantic meaning and its primitive value. An engineer can understand a token's purpose without asking. A new team member can orient themselves in the system by reading names — not by asking for a guided tour.

Where This Leads

Layer 4 — Documentation and Workflow builds on this architecture to answer the question everyone eventually runs into: *how do we keep all of this legible and current as the system grows?*

The three-tier structure gives you the *what*. Documentation gives you the *how* and the *why* — and it's what turns naming decisions from tribal knowledge into durable, transferable system knowledge.

Layer 4 — Naming Templates

The Blank Page Problem

Every naming decision in a design system is a small act of design. And like all design work, it's easy to stall when facing a blank page.

Most teams default to one of two failure modes when there's no template to work from:

Failure Mode A: Name by gut. Someone types a name that sounds right to them in the moment. It's never questioned because reviewing token names feels pedantic. Six months later, nobody can explain why it was named that way — including the person who named it.

Failure Mode B: Name by committee. The team schedules a meeting to debate the name. Everyone brings different mental models. The discussion circles without resolution. Eventually someone's preference wins by attrition, not by logic. The decision is unmemorable because it wasn't documented.

Both failures share a root cause: the team is trying to make a **naming decision** without a **naming system**. They're answering the question "what should we call this?" before answering "what are the rules for how we name things?"

A naming template solves this by front-loading the system design. Once the template exists, naming decisions become a matter of filling in coordinates — not inventing logic from scratch.

What a Naming Template Actually Is

A naming template is not a style guide. It's not a list of approved words. It's not a spreadsheet of existing token names.

A naming template is a **set of structured questions** that force the right thinking before a name is chosen. It captures the *reasoning* behind a name, not just the name itself.

This matters because names without reasoning are fragile. When the reasoning is documented alongside the name, future team members can:

- Understand why a token was named that way
- Apply the same logic to new tokens consistently
- Challenge and revise names with evidence, not opinion
- Onboard to the system without a guided tour

The Six Questions Every Token Name Should Answer

The workshop framework organizes naming documentation around six questions. These map to the classic **who, what, where, when, why, how** structure — applied specifically to design token decisions.

Who

Who is this token for?

This isn't about individual people. It's about audience tier: is this token intended to be consumed directly by designers, by engineers, by both, or by neither (i.e., it's an internal alias that powers other tokens)?

The answer shapes the name's vocabulary. A primitive token (`color-blue-500`) doesn't need to speak to designers — it's infrastructure. A semantic token (`color-action-primary`) needs to be readable by both designers and engineers. A component token (`button-container-background-hover`) is primarily an engineering contract, though it should still be legible to a designer reading through it.

What

What design decision does this token represent?

This is the most important question — and the one most often skipped.

A token isn't just a value. It's a **decision**. And every decision has a reason. Documenting the decision means documenting:

- What property is being controlled
- What role or meaning that property carries
- What it's *not* (the alternatives that were considered and rejected)

For example: `color-feedback-error` doesn't just mean "red." It represents the decision that *this specific shade of red is the one your system uses to signal an error state across all components*. That decision has consequences: it means no component should use a different red for error states; it means this token should be used consistently wherever errors are communicated.

If you can't articulate the decision, the name is premature.

Where

Where is this token used — global, contextual, or component-scoped?

Tier placement answers this question, but it's worth making explicit in documentation:

- **Global** — applies everywhere in the system (tier 1 primitives, most tier 2 semantics)
- **Contextual** — applies within a specific theme, mode, or brand variant
- **Component-scoped** — applies only within a specific component (tier 3)

Documenting scope prevents tokens from being misused. When a component token is mistaken for a semantic token and used in a different component, the system develops invisible coupling that's hard to untangle later.

When

When does this token apply — which state or condition activates it?

For stateless tokens, this is straightforward: `color-surface-primary` applies in the default, resting state of any primary surface.

For stateful tokens, this requires precision: what exactly triggers the `hover` state? Does it apply on touch devices? What about `focus-visible` vs `focus`? Is `disabled` a visual state, an interactive state, or both?

Documenting "when" prevents teams from applying state tokens inconsistently — for instance, applying a `hover` color to an element that should be showing a `selected` state.

Why

Why was this name chosen over alternatives?

This is the most overlooked question — and the one that pays off most over time.

Every naming decision has alternatives. Some are obviously worse. Some are genuinely close calls. Documenting why the chosen name won (and what the runner-ups were) serves two purposes:

1. It prevents the same debate from being relitigated every time a new team member encounters the name
2. It creates an audit trail that makes the naming system **learnable** — not just usable

A short note is enough:

> *""We chose `emphasis` over `weight` because `weight` is already used in typography contexts. We chose `primary/secondary/tertiary` over `1/2/3` because the labels are more readable in code review and less likely to be confused with scale indices.""*

That reasoning takes thirty seconds to write and saves hours of future debate.

How

How does this token connect to adjacent tokens in the system?

No token lives in isolation. Documenting the relationships makes the system navigable:

- What primitive does this semantic token reference?

- What semantic token does this component token reference?
- What other tokens belong to the same family?
- Is this token part of a scale, an emphasis set, or a state progression?

This is particularly important at the semantic tier. `color-action-primary` only makes full sense when you can also see `color-action-secondary`, `color-action-subtle`, and `color-action-primary-hover`. The token is a node in a network — the documentation should make that network visible.

The Template in Practice

Here's what a completed template entry looks like. This is not a spreadsheet row — it's a documentation record that captures the full reasoning:

Token: `color-action-primary-default`

Who: Designers and engineers. This is a semantic token intended to be consumed directly in component tokens and, where no component token exists, directly in components.

What: The primary action color in its default (resting) state. This is the color used for interactive elements that initiate a primary action — buttons, links, actionable icons. It is *not* used for feedback states (errors, success), brand decoration, or neutral surfaces.

Where: Global. Applies across all components and contexts unless overridden by a theme or mode variant.

When: Default state — no interaction has occurred. Replaced by `color-action-primary-hover` on hover, `color-action-primary-active` on press, `color-action-primary-disabled` when the element is non-interactive.

Why: `action` was chosen over `interactive` because it more precisely describes the category (things that *do* something, not just things the user can interact with). `primary` was chosen over `brand` because action hierarchy is independent of brand expression — a brand color might not always be the action color across all themes.

Connects to:

- References primitive: `color-blue-600`
- Sibling tokens: `color-action-primary-hover`, `color-action-primary-active`, `color-action-primary-disabled`, `color-action-primary-focus`
- Semantic siblings: `color-action-secondary-default`, `color-action-subtle-default`
- Consumed by: `button-container-background-default`, `link-text-default`, `icon-button-foreground-default`

This level of documentation isn't necessary for every token — but it *is* necessary for the foundational decisions that govern large swaths of the system. The tokens at the top of the hierarchy — the ones everything else builds on — deserve this treatment.

For more routine tokens, a lighter version works:

Token: `button-container-background-hover` **References:** `color-action-primary-hover` **State:** Hover — cursor is over the button container **Note:** Follows action-primary progression; does not reference primitive directly.

Naming Templates for Each Tier

The six questions apply everywhere, but the emphasis shifts depending on which tier you're working in.

Tier 1 Template (Primitives)

The key decisions at this tier are about **scale structure** and **completeness** — do you have all the values you need, are they organized into coherent families, and are the scale steps meaningful?

For each primitive family, document:

- Scale structure: what steps exist and why (why 9 steps in the color scale? why these spacing values?)
- Naming convention: numeric (100–900), t-shirt (xs–3xl), or other
- Gaps intentionally left: values that don't exist and why they don't need to

Tier 2 Template (Semantics)

The key decisions at this tier are about **meaning** and **relationships** — what role does this token play, and how does it relate to other semantic tokens?

For each semantic category, document:

- The vocabulary in use (e.g., `primary / secondary / tertiary / subtle` for emphasis)
- The state progressions in use (e.g., `default / hover / active / disabled / focus`)
- The scope: which themes or modes might override these, and how

---

Tier 3 Template (Component Tokens)

The key decisions at this tier are about **anatomy** and **application** — which parts of this component have tokens, and why?

For each component, document:

- The anatomy map (from Layer 2) — every part that has a token
- The property/state matrix — which states apply to which properties
- Anything *not* tokenized, and why (not every property needs a token)
- Which semantic tokens each component token references

Turning Templates Into Workflow

A template that lives in a document nobody opens is no better than no template at all. The goal is to embed the template into the workflow — the moment when naming decisions are actually made.

Three practical approaches:

Approach 1: Name as you build. When a designer creates a new component or token in Figma, they fill in the template in the same session. It's not a separate documentation task — it's part of the definition of "done."

Approach 2: Name in pre-build planning. Before a component is built, the team fills in the template as a planning exercise. The naming decisions are made collaboratively, *before* the component exists, so the build phase starts with a clear contract.

Approach 3: Name in documentation sprints. For existing systems without templates, schedule focused sessions to apply the framework retroactively. Start with the tokens that cause the most confusion or questions — those are the ones where missing documentation has the highest cost.

> *"Whatever workflow you choose: documentation written at the time of decision is always more accurate than documentation written after the fact. Memory fades. Reasoning reconstructed weeks later is reasoning invented, not recalled."*

What You Now Have

With naming templates in place, you have:

- A **repeatable process** for making naming decisions that doesn't start from scratch
- **Documented reasoning** that survives team changes, onboarding, and time
- A **shared language** for discussing naming decisions that isn't dependent on any one person's intuition
- The raw material for the next layer: a documentation workflow that makes these templates accessible and usable by the whole cross-functional team

---

Where This Leads

Layer 5 — Documentation Workflows takes everything captured in naming templates and asks the harder question: *where does this documentation live, how does it stay current, and how does it reach the people who need it — including LLMs?*

Layer 5 — Documentation Workflows

The Documentation Trap

Most design system documentation gets written in one of two moments:

Too early — before the system is real, as aspirational guidelines that don't reflect how the system actually works once it's built.

Too late — as a cleanup effort after the fact, when the people who made the decisions have moved on, the reasoning has been forgotten, and documentation becomes archaeology rather than knowledge transfer.

Both produce the same result: documentation that doesn't match reality, that nobody trusts, and that quietly stops being used.

The workshop's approach is different. It treats documentation not as a deliverable that gets produced once, but as a **workflow** — a continuous practice woven into the work itself. The goal isn't a documentation site. It's a documentation habit.

Why "Outside the Design Tool" Matters

One of the workshop's specific emphases is documenting naming decisions *outside* design tools. This deserves unpacking, because it runs counter to how most designers work.

Figma is where design happens. It feels natural to document decisions there — in component descriptions, in annotation frames, in property panel notes. And some of that is genuinely useful.

But design tools have structural limitations as documentation systems:

- **Access is gatekept.** Engineers, product managers, and marketing often don't have Figma access — or don't know how to navigate it if they do.
- **Search is limited.** Finding a specific decision buried in a component's description requires knowing where to look.
- **History is opaque.** Figma doesn't make it easy to understand *why* something changed, only *that* it changed.
- **LLMs can't read it.** A naming convention documented in a Figma frame is invisible to the AI tools increasingly being used alongside design systems work.

Documentation that only lives in the design tool is documentation that only serves designers. The workshop's principle is that naming decisions affect the whole cross-functional team — so the documentation should be accessible to the whole cross-functional team.

The Two Types of Documentation a Naming System Needs

Before getting into workflows, it helps to distinguish two different things that often get conflated under the word "documentation."

Type 1: Decision Records

Why did we make this choice?

Decision records capture the **reasoning** behind a naming decision at a specific point in time. They're the output of the naming template from Layer 4. They answer the who, what, where, when, why, and how.

Decision records don't change when the system evolves — they're a historical record. If a token gets renamed, you add a new decision record explaining why, rather than editing the old one.

This is sometimes called an **Architecture Decision Record (ADR)** in engineering contexts. The concept translates directly to design systems work.

Type 2: Reference Documentation

What does this name mean, and how do I use it?

Reference documentation is the living, current-state description of your naming system. It's what a new team member reads to understand how the system works today. It's what an engineer consults when they're unsure whether to use `color-action-primary` or `color-brand-primary`. It's what you give an LLM as context before asking it to help name a new token.

Reference documentation *does* change as the system evolves — it reflects current reality.

> *"**The key distinction:** Decision records explain history. Reference documentation explains the present. A healthy naming system needs both, and they live in different places with different maintenance rhythms."*

The Documentation Stack

A practical documentation workflow uses a small stack of tools and formats, each doing a specific job. Here's how the layers work together:

Layer A: The Source of Truth Document

A single, non-design-tool document — typically a wiki page, Notion doc, or markdown file in a repository — that contains:

- The **naming conventions** in use (the vocabulary, the axes, the patterns)
- The **tier structure** and what each tier is responsible for
- The **category glossary** (what `action` means vs. `brand` vs. `feedback`, etc.)
- The **state vocabulary** and when each state applies
- The **component type definitions** (simple, sophisticated, compound, composed)

This document is the **onboarding document** for anyone new to the system. It's also the context document you give to an LLM when you want it to help with naming work.

It should be short enough to read in one sitting. If it runs to thirty pages, it's too long — that's a sign it's trying to do too much. The source of truth document describes the *rules*; it doesn't catalog every token.

Layer B: The Token Catalog

A structured record of every token in the system, including:

- Token name
- Tier (primitive / semantic / component)

- Current value or reference
- Status (stable / experimental / deprecated)
- Brief description of purpose

This is the **lookup resource** — not for understanding the system, but for finding a specific token quickly. It can live in a spreadsheet, a documentation site, a JSON file with comments, or a dedicated token management tool.

The catalog is high-maintenance by nature. The most practical way to keep it current is to generate it from the token files themselves where possible, rather than maintaining it manually in parallel.

Layer C: Decision Records

A running log of significant naming decisions — the template records from Layer 4, organized chronologically or by category.

These don't need to be elaborate. A decision record can be as simple as:

```
Date: [date]
Token: color-feedback-warning-default
Decision: Named 'warning' rather than 'caution' or 'alert' to align
with engineering convention in our codebase. 'Alert' was rejected
because it conflicts with the alert component name.
References: color-yellow-500
Status: Stable
```

The format matters less than the habit. A folder of simple text files, a section in a wiki, a Notion database, a GitHub discussion thread — any of these work. The goal is that the reasoning is *somewhere findable*, not that it's in any particular tool.

Layer D: In-Tool Annotations

Descriptions, notes, and labels that live *inside* Figma (or your design tool of choice) and provide immediate context at the point of use.

These are not the source of truth — they're a **signpost** pointing to the source of truth. A good in-tool annotation might say:

> *“"Primary action color. Default state. References: color-action-primary-default. Full documentation: [link]."”*

Short, specific, and linked. Not a self-contained essay, but not empty either.

In-tool annotations are what designers encounter first. Their job is to confirm that the designer is using the right token and point them toward deeper documentation if they need it.

The Documentation Workflow in Practice

A workflow isn't a structure — it's a sequence of actions that happen at specific moments. Here's what that looks like concretely.

Moment 1: Creating a New Token

When a new token is created:

1. **Fill in the naming template** (the six questions from Layer 4) before finalizing the name
2. **Add the token to the catalog** with its tier, reference, and a one-line description
3. **Write a decision record** if the naming decision was non-obvious (if anyone on the team would ask "why is it called that?", document it)
4. **Add an in-tool annotation** in the design file pointing to the catalog entry

This takes five to ten minutes per token. It feels like overhead until the second time someone asks you to explain a name you documented this way — and the answer is already written.

Moment 2: Renaming or Deprecating a Token

Renames and deprecations are where undocumented systems fall apart. Without a workflow, a rename becomes a quiet change that breaks things downstream with no explanation.

When a token is renamed:

1. **Write a decision record** for the deprecation — why the old name is being retired, what replaces it
2. **Update the catalog** — mark the old name as deprecated, add the new name as stable
3. **Update the source of truth document** if the rename reflects a convention change
4. **Update in-tool annotations** to reflect the new name
5. **Communicate the change** — a brief note in whatever channel the team uses for system updates

The decision record for a rename is often the most valuable one in the system. "We renamed X to Y because..." is exactly the kind of institutional knowledge that disappears without documentation.

Moment 3: Onboarding a New Team Member

With a documentation stack in place, onboarding to the naming system has a clear sequence:

1. Read the source of truth document (the rules)
2. Browse the token catalog (the current state)
3. Skim recent decision records (the recent history and reasoning)

This takes an afternoon rather than weeks of tribal knowledge transfer. The documentation *is* the guided tour.

Moment 4: Asking an LLM for Help

This moment is increasingly common — and it's where documentation quality has a direct, measurable impact on output quality.

An LLM asked to "help name this new token" without context will produce names that sound plausible but don't follow your system's conventions. An LLM given your source of truth document as context will produce names that fit your vocabulary, your tier structure, and your established patterns.

The practical workflow:

1. Paste (or reference) the source of truth document as context
2. Describe the token you need to name: its tier, its role, its relationships
3. Ask the LLM to propose names following the established conventions
4. Review proposals against your naming template questions
5. Document the chosen name as you would any other

The LLM becomes a **naming accelerator**, not a naming authority. It proposes; you evaluate. The documentation is what makes the proposals good.

What Makes Documentation Stay Current

The hardest part of any documentation workflow isn't starting — it's maintaining. Documentation drifts because the work of keeping it current competes with the work of building the system.

A few principles that help:

Make documentation the path of least resistance. If the workflow for adding a token requires touching a separate documentation tool, it won't happen consistently. The closer the documentation step is to the naming step, the more likely it is to occur. Templates embedded in the tools people are already using work better than templates that require switching contexts.

Assign ownership, not just responsibility. "Everyone is responsible for documentation" means nobody is. A named person or rotation for documentation review — even if it's just a monthly fifteen-minute pass through the catalog — is more effective than diffuse accountability.

Let the catalog be generated, not written. Manual token catalogs go stale. Wherever possible, generate the catalog from token files, with descriptions pulled from structured comments in the token data. The catalog becomes current automatically when tokens change.

Treat stale documentation as a system failure, not a people failure. When documentation falls behind, it's usually because the workflow made it too hard, not because people didn't care. The fix is a better workflow, not more reminders.

Visual Mapping as Documentation

The workshop introduces visual mapping techniques as a documentation tool — not for final reference documentation, but for the **thinking stage** that precedes it.

A visual map is a diagram that shows:

- The relationships between token tiers
- The family structure within a semantic category
- The state progressions for a given token set
- The anatomy of a component mapped to its token names

Visual maps serve two purposes:

For the team making decisions: A diagram forces you to see gaps, inconsistencies, and awkward relationships that aren't visible in a list. It's easier to spot a missing state or an asymmetric family when the tokens are laid out spatially.

For communicating decisions to others: A diagram communicates structure in a way that a list or table can't. Showing a stakeholder the three-tier structure as a connected diagram lands differently than describing it in prose.

The visual map isn't the documentation itself — it's a **thinking tool** that produces documentation. It answers "does this structure make sense?" before you commit to it in a token file or a catalog entry.

The Minimum Viable Documentation Workflow

For teams starting from nothing, or teams that feel the full stack is too much to take on at once, here is the smallest version that still provides meaningful value:

Step 1: Create a single shared document with your naming conventions — the vocabulary, the tier structure, the category definitions. One to three pages. Share it with the whole team.

Step 2: When naming a new token that isn't obviously covered by existing conventions, write two sentences: what decision this token represents, and why it's named the way it is. Put them in the document.

Step 3: Add a one-line description to every token in your design tool. Even just: "Primary action color, default state."

That's it to start. Three steps, all lightweight. The full stack grows from this foundation as the system — and the team's documentation habits — mature.

What You Now Have

With a documentation workflow in place, the naming system becomes:

- **Accessible** — not locked in one person's head or one tool's interface
- **Durable** — reasoning survives team changes, reorgs, and time
- **Teachable** — new team members can learn the system from documentation, not just from observation
- **LLM-compatible** — the source of truth document becomes a context payload that makes AI assistance genuinely useful
- **Maintainable** — because documentation is a workflow, not a one-time event

---

Where This Leads

Layer 6 — LLM-Assisted Workflows takes the documentation infrastructure you've just built and puts it to active use — showing how to construct prompts, workflows, and feedback loops that use AI tools to accelerate naming decisions, generate documentation drafts, and validate consistency across a growing system.

Layer 6 — LLM-Assisted Workflows

Why This Layer Exists

Every layer up to this point has been building toward something that works without AI assistance. The systems thinking, the anatomy mapping, the three-tier structure, the naming templates, the documentation workflows — all of it is designed to be human-operable, human-maintainable, and human-legible.

That foundation is not incidental to this layer. It's the prerequisite for it.

LLMs are pattern-matching engines. Give one a well-structured naming system with documented conventions, and it will recognize the patterns and extend them accurately. Give one an undocumented, inconsistent system, and it will invent plausible-sounding names that fit nobody's conventions — because there are no conventions to fit.

The investment in Layers 1 through 5 is what makes Layer 6 possible. The documentation you built in Layer 5 is the context that makes LLM assistance reliable rather than random.

> *"**The fundamental principle:** An LLM is a naming accelerator, not a naming authority. It proposes; you evaluate. The quality of its proposals is a direct function of the quality of the context you give it."*

What LLMs Are Good At in Naming Work

Before getting into specific workflows, it helps to be clear-eyed about where LLMs genuinely help — and where they don't.

Where LLMs Add Real Value

Pattern extension. Given an established set of token names, an LLM can generate new names that follow the same patterns accurately and quickly. This is where they save the most time — not inventing conventions, but applying existing ones at scale.

Consistency checking. An LLM can scan a token list and identify names that violate established conventions — inconsistent terminology, missing states in a progression, tier mismatches, naming patterns that don't match the documented vocabulary.

Generating alternatives. When a name choice is genuinely ambiguous, an LLM can generate a set of options with reasoning for each — essentially doing the first pass of the "why" question from the naming template. This surfaces considerations you might not have thought of.

Documentation drafting. Given a token name and its context, an LLM can produce a first draft of the decision record — the who, what, where, when, why, how structure from Layer 4. The draft will need editing, but it's faster to edit than to write from scratch.

Vocabulary arbitration. When a team is debating between two naming approaches, an LLM can analyze both against established conventions and articulate the tradeoffs — sometimes more neutrally than any team member who has become attached to a position.

Where LLMs Fall Short

Organizational context. An LLM doesn't know your team's history, your political constraints, your technical debt, or the internal debates that shaped your current conventions. It can't know that `action` was chosen over `interactive` because of a specific engineer's strong objection eighteen months ago.

Value judgments. When two options are genuinely equal by the rules of your system, an LLM can't make the call. It can tell you the options are equivalent and why — but the decision still belongs to a human.

Maintaining state across sessions. Most LLMs don't remember previous conversations. Every session starts fresh. This means the context that makes them useful has to be re-supplied every time — which is exactly why a well-maintained source of truth document is so important.

Catching subtle semantic drift. An LLM might not notice that you've been slowly using `primary` to mean two slightly different things across two different parts of your system — because the drift is subtle enough that it looks consistent at the pattern level. Human review is still necessary.

The Context Payload

The single most important concept in LLM-assisted naming workflows is the **context payload** — the structured documentation you provide at the start of a session to orient the LLM to your system.

A context payload is not a vague description. It's a specific, structured document that contains everything the LLM needs to extend your naming system accurately.

What Goes in a Context Payload

1. Naming conventions summary
The core vocabulary of your system: what axes you use, what each axis means, and how they combine. This comes directly from your source of truth document.

```
Our token names follow this structure:
- Tier 1 (primitives): [category]-[family]-[scale]
  e.g., color-blue-500, space-16, font-size-14
- Tier 2 (semantic): [category]-[role]-[emphasis]-[state]
  e.g., color-action-primary-default
- Tier 3 (component): [component]-[part]-[property]-[state]
  e.g., button-container-background-hover
```

2. Vocabulary definitions
The specific terms your system uses and what they mean. Don't assume the LLM knows what you mean by `subtle` or `surface` — define it explicitly.

```
Emphasis vocabulary: primary (highest weight), secondary (supporting),
tertiary (minimal), subtle (de-emphasized, near-invisible)

Role vocabulary:
- action: interactive elements that initiate an action
- feedback: system responses (error, warning, success, info)
- brand: decorative or identity-driven color
- neutral: backgrounds, borders, non-semantic surfaces
```

3. State vocabulary
The states your system uses and the precise conditions under which each applies.

4. A representative sample
Ten to twenty existing token names from your system, across tiers, that demonstrate the conventions in practice. This gives the LLM concrete examples to pattern-match from — not just abstract rules.

5. Explicit constraints
Things the LLM should *not* do — conventions you've rejected, vocabulary you've deliberately avoided, patterns that look plausible but violate your system.

```
Do not use:
- 'interactive' (we use 'action')
- 'color' as a role name (too generic)
- numeric suffixes on semantic tokens (-1, -2, -3)
- camelCase (we use kebab-case throughout)
```

Core LLM Workflows

Workflow 1: Naming a New Token

When to use it: You need to name a token that fits into an established tier and category, and you want to move quickly without sacrificing consistency.

The prompt structure:

```
Context: [paste your context payload]

Task: I need to name a new [tier] token with the following characteristics:
- Category: [color / space / typography / etc.]
- Role: [what this token does in the system]
- Tier: [primitive / semantic / component]
- Relationships: [what it references, what references it]
- State: [if applicable]

Please propose three name options following our naming conventions,
with a brief explanation of each. Then recommend one and explain why.
```

What to do with the output:
Treat it as a shortlist, not a decision. Run each proposal through the six questions from the naming template. The LLM's recommendation is a starting point — your evaluation is the decision.

Workflow 2: Consistency Audit

When to use it: You have an existing set of tokens — a new component's token set, or a section of your semantic tier — and you want to check it against your conventions before committing.

The prompt structure:

```
Context: [paste your context payload]

Task: Please audit the following token names for consistency with
our naming conventions. For each issue found, explain:
1. What convention is being violated
2. What the name should be instead
3. Why the current name is problematic

Token list:
[paste token names]
```

What to do with the output:
Review each flagged issue. Some will be genuine violations; others will be edge cases where your conventions are ambiguous. The ambiguous ones are the most valuable — they reveal gaps in your documented conventions that need to be closed.

Workflow 3: Generating a Token Family

When to use it: You're building out a new semantic category and need to generate the full set of tokens — all emphasis levels, all states — before you start designing components that consume them.

The prompt structure:

```
Context: [paste your context payload]

Task: I'm creating a new semantic token family for [role/category].
Based on our conventions, generate the complete token set including:
- All emphasis levels we use (primary, secondary, tertiary, subtle)
- All states we use (default, hover, active, disabled, focus)
- Any additional variants specific to this category

Format the output as a structured list grouped by emphasis level.
Flag any places where you're uncertain whether a token is needed.
```

What to do with the output:
The LLM will generate a comprehensive set — likely more tokens than you immediately need. Treat this as a menu, not a mandate. Remove tokens that don't have a use case yet. Add any the LLM missed. The value is having the complete logical space laid out so you can make deliberate inclusion/exclusion decisions rather than discovering gaps later when a component needs something that doesn't exist.

Workflow 4: Documentation Drafting

When to use it: You've made a naming decision and need to write the decision record. You have the reasoning in your head but haven't written it down.

The prompt structure:

```
Context: [paste your context payload]

Task: Help me write a decision record for the following token:

Token name: [name]
What it represents: [brief description]
Alternatives considered: [other names that were discussed]
Why this name was chosen: [your reasoning, even if rough]
References: [what primitive it points to, what references it]

Please format this as a complete decision record following our
six-question template (who, what, where, when, why, how).
Fill in anything I haven't specified based on our conventions,
and flag anything you're not sure about.
```

What to do with the output:
Edit the draft — the LLM will fill in some sections more confidently than others. Verify that the "why" section accurately reflects your reasoning rather than a plausible-sounding reconstruction. Add any organizational context the LLM couldn't know.

Workflow 5: Naming Arbitration

When to use it: Your team is stuck between two or more naming approaches and needs a structured way to evaluate the tradeoffs without the discussion becoming political.

The prompt structure:

```
Context: [paste your context payload]

Task: We're debating between the following names for a [tier] token
that [describe the token's role]:

Option A: [name]
Option B: [name]
Option C: [name] (if applicable)

For each option, evaluate:
1. How well it follows our documented conventions
2. How well it communicates the token's role to designers
3. How well it communicates the token's role to engineers
4. How well it scales to related tokens we'll need to add
5. Any potential for confusion with existing names

Then give a recommendation with your reasoning.
```

What to do with the output:
Use the analysis to structure your team discussion, not to end it. The LLM's recommendation carries no authority — but its structured evaluation of the tradeoffs can move a stuck conversation forward by giving the team a shared framework to respond to rather than competing intuitions to defend.

Workflow 6: Onboarding Documentation Generation

When to use it: You need to create or update the source of truth document — the conventions summary a new team member or LLM session would read to understand your system.

The prompt structure:

```
I'm going to give you a sample of token names from our design system.
Based on these, please reverse-engineer and document:
1. The naming conventions in use (structure, axes, vocabulary)
2. The tier structure and what each tier is responsible for
3. Any vocabulary definitions that can be inferred
4. Any conventions you can't determine from the sample alone
   (flag these explicitly as gaps to fill)

Token sample:
[paste 30–50 representative token names across all tiers]
```

What to do with the output:
This is particularly useful for existing systems that grew without explicit conventions. The LLM's reverse-engineered documentation gives you a starting draft of your source of truth document — and the gaps it flags are exactly where undocumented tribal knowledge lives. Those gaps are your documentation backlog.

Building a Feedback Loop

The most sophisticated LLM workflow isn't a single prompt — it's a **feedback loop** that improves the system over time.

The loop works like this:

1. Use LLM to name or audit.
Run one of the workflows above. Collect the output.

2. Evaluate against your conventions.
Where the LLM got it right, that's evidence your conventions are well-documented. Where it got it wrong or uncertain, that's evidence of a gap.

3. Update your documentation.
Close the gaps the LLM exposed. Add definitions it couldn't infer. Make explicit the conventions it had to guess at.

4. Test with the updated context.
Run the same prompt with your updated context payload. Did the output improve? If so, your documentation is better — not just for the LLM, but for every human who reads it too.

5. Repeat.
Every naming session leaves the documentation slightly better than it found it. Over time, the context payload becomes a precise, comprehensive description of your system that produces consistent results with minimal correction.

> *"The feedback loop is the mechanism by which LLM-assisted workflows make your human system better — not just faster. Every gap the LLM exposes is a gap that would have confused a new team member too."*

Practical Guardrails

Working with LLMs in naming workflows requires a few explicit guardrails to prevent the most common failure modes.

Always supply context, every session.
LLMs don't carry memory between sessions. A session without a context payload produces generic output. Make supplying the context payload a non-negotiable first step — treat it like opening the right file before you start working.

Never accept a name without running it through the template.
The LLM's proposal is a candidate, not a decision. The six questions from Layer 4 are your evaluation filter. A name that sounds right but can't answer "why was this chosen over alternatives?" isn't ready to enter the system.

Flag LLM-generated names in documentation.
When a decision record documents a name that was generated or significantly shaped by an LLM, note it. This isn't a quality judgment — it's a transparency practice that helps future team members understand the provenance of decisions.

Treat confident errors as the most dangerous output.
LLMs produce plausible-sounding names even when they're wrong. A name that sounds exactly right but violates a subtle convention is harder to catch than one that's obviously off. Consistency audits are the check — don't skip them because the names look good.

What You Now Have

With LLM-assisted workflows in place, your naming system has:

- A **scalable process** for generating names that fit your conventions, even as your system grows faster than any individual can manually keep up with
- A **consistency mechanism** that catches violations before they compound into systemic confusion
- A **documentation accelerator** that reduces the friction between making a decision and recording it
- A **feedback loop** that uses AI assistance to improve the human documentation — not just to produce output faster

And underneath all of it: the foundation of Layers 1 through 5, without which none of this layer would work.

The Complete Picture

You've now moved through the full arc of the workshop framework:

Layer 1 gave you the mental model — names as rules, names as coordinates, systems thinking applied to the problem of naming.

Layer 2 gave you the vocabulary — component anatomy, part names, the formula that turns structure into consistent token names.

Layer 3 gave you the architecture — the three-tier structure and the four semantic naming domains where decisions concentrate.

Layer 4 gave you the process — naming templates that capture reasoning, not just names.

Layer 5 gave you the infrastructure — documentation workflows that make reasoning accessible, durable, and current.

Layer 6 gave you the acceleration — LLM workflows that extend your system at speed without sacrificing the consistency the earlier layers built.

These layers don't have to be implemented all at once. But they do build on each other. A team that skips to Layer 6 without Layers 1 through 5 will get fast, confident, inconsistent output. A team that builds all six — even at minimum viable scale — has a naming system that can outlast any individual contributor and grow without accumulating confusion.

What Comes Next

With all six layers in place, the natural next step is making this material **actionable** — moving from understanding the framework to practicing it in hands-on exercises that simulate the real decisions your team will face.

That's where templates, worksheets, and workshop activities come in: tools that let you apply the framework to concrete scenarios before using it on your live system.

Ready to build those tools together?