Making a Design System Machine-Legible

AI-directed design systems tooling — architecture, validation, and the round trip back to design

Current work. Amazon, CORE UX, Devices and Services.

Context

I lead design systems and platform strategy for 1P, 2P, and 3P enablement within CORE UX at Devices and Services. The role runs across two disciplines by design: I manage designers and I manage programs — the processes, the pipeline, and the partnerships across several partner organizations.

In practice that means owning the Amazon Developer B2B experience for app subscription submissions on Alexa devices, running the Fire TV component pipeline and the toolkit our teams build in, and building the operational tooling those teams depend on. It is a two-sided problem in the same shape as the marketplace work I owned previously on the Amazon Appstore: first-party teams, second-party partners, and third-party manufacturers all need the same system to hold, while each needs to bend it differently.

That last constraint is what pushed me into the work below. When third parties customize your components, you cannot review every downstream implementation. The system has to encode its own rules, or it does not hold.

The problem

AI did not remove the bottleneck in design production. It moved it.

Generating a first version — a screen, a component, a page of markup — is now nearly free. Reconciling that version back into a real system is not. So teams add prototyping capacity and end up with more artifacts requiring refinement, in code and in design files, than they had before. The visible symptom is velocity that does not convert: more output, the same shipping rate, and a rework queue that grows.

The common response is to buy more generation. That makes the problem worse, because generation was never the constraint.

The actual constraint is that design systems are not legible to a model. A design file carries geometry, style, and layer structure. It does not carry ontology — what a thing is, what it is for, what it may and may not do. Most tooling in this space treats the design file as the source of truth and tries to extract semantics that were never encoded there. That is why it plateaus in the same place every time: it can produce something plausible, and it cannot produce something correct.

The inversion

The design file supplies appearance. A separate specification layer supplies meaning.

Once you accept that, the work stops being prompt engineering and becomes systems architecture. I have been building that specification layer, and it has five parts:

Intent encoded in the token, not the value. A token named for its appearance tells a model nothing about where it is allowed to appear. A token named for its role tells it exactly that. Most systems encode appearance and leave intent in the heads of the people who maintain the library — which works fine until the consumer is a machine, or a third party you will never meet.

A composition grammar. Which components may nest inside which, what a valid slot is, which combinations are forbidden. This is the part that lives almost entirely as tacit knowledge in two or three maintainers. Externalizing it is most of the work and nearly all of the value.

A closed vocabulary instead of negative instruction. Telling a model not to embellish competes against everything it has ever seen; it fills open space with the most statistically common pattern available. Enumerating the allowed components, props, token names, and composition rules — and rejecting anything outside the enumeration — changes the behavior without persuasion. You stop asking and start constraining.

Executable definitions of failure. The quality bar has to be something the system can apply, not something a person applies at review. That means canonical accepted and rejected examples, each rejection carrying its reason, and a validator that runs. This is the piece that turns a reviewer's judgment into infrastructure — and it is the precondition for a system that can correct its own output.

Stable identity across the round trip. Without an ID that survives design file to code and back, "bidirectional" is regeneration wearing a costume. The test is not whether something returns. It is whether it returns as a bound component instance with its layout structure and constraints intact — something a designer can keep working in, rather than a flattened pile of frames.

What I built and what it proved

Working proofs, built through prompt- and IDE-driven development, currently cover:

- Ingesting design pages and frames, and identifying components within them
- Pulling in and validating design tokens against the specification
- Translating design files into code, or into structured markdown for further ideation in either direction
- Exporting code, interface guideline documentation, and component libraries
- Validators and instructional layers that check output against the rules and explain what failed

- A round trip that returns generated components to a layered, editable design file — the piece most tooling in this category does not attempt

The technical finding worth carrying forward: **the model was never the limiting factor.** Quality tracked almost entirely with the precision of the specification. Every failure I chased turned out to be an ambiguity in the system I was describing, not a deficiency in the tool describing it. Which means the work is design leadership — deciding what the system means — wearing a technical costume.

The second finding: **the intelligence belongs in the specification layer, not in a tuned model.** A rules corpus is portable across model generations, versionable, and auditable. It improves every time the underlying models improve. A fine-tuned model is locked to a vendor and a version, and it goes stale.

What I would build next

With a team rather than the margins of a role:

1. **Take one component family fully to production standard** — spec, golden set, validator, round trip — rather than broad partial coverage. A complete narrow slice is demonstrable; a broad partial one is not.
2. **Formalize the golden set** as a regression suite, so quality is enforced continuously rather than reviewed periodically.
3. **Close the self-correction loop** — reject, return the reason, retry against it — which is only possible once failure is executable.
4. **Instrument it.** Rework rate, time from generated to merged, percentage of output passing validation on first attempt. If the argument is productivity, the argument needs numbers.

Why this is a leadership problem

Wiring a design tool to a model is a weekend. Specifying a system precisely enough that a model cannot produce a wrong result requires knowing what the system means, what the teams consuming it actually need, where the exceptions are legitimate, and where the quality bar is real versus habitual.

That knowledge does not come from the tooling. It comes from having built design systems from scratch, from having managed the teams that live with the output, and from having sat between design and engineering long enough to know which arguments are worth encoding and which are worth dropping.

Seventeen years, design teams of up to twenty, systems built from nothing at Goldman Sachs and net-new developer portals at JPMorgan Chase. This is the same work. The consumer changed.

Written from my own method and architecture. No proprietary system content, code, or interface material is included.