ISSN: 2096-3246
Volume 58, Issue 01, 2026

DESIGN TOKENS AND CONTRACT-FIRST SYSTEMS FOR CROSS-PLATFORM UI CONSISTENCY

Harshit Sunilkumar Vora
Independent Researcher, USA

Abstract
The development of cross-platform UI at scale requires not only style guides, design patterns, and design language systems but also a design system architecture that enforces principled contracts between visual decisions, behavioral expectations, and accessibility semantics. Design tokens encode visual intention in a portable, platform-agnostic format, enabling design decisions to be semantically mapped to platform representations without re-encoding at each surface. Component contracts abstract stable interaction semantics, accessibility expectations, state transitions, and error handling behaviors that must remain consistent across device classes and rendering backends. Together, these architectural mechanisms address the fundamental sources of interface drift—behavioral divergence, semantic discontinuity, and undetected accessibility regressions—all of which compound across platform-driven release cycles. Interaction correctness and cross-platform consistency are treated as first-class reliability properties, not post-hoc quality concerns. Token standardization and tool-agnostic interchange formats motivate treating tokens as versioned public interfaces subject to the same governance disciplines applied to APIs. Breaking change detection, managed escape hatches, and continuous drift detection pipelines maintain interface integrity as teams scale and surface proliferation increases. Quality is measured through metrics including token coverage, component reuse rates, contract coverage, and assertion parity across platform targets.

Keywords: Design Tokens, Component Contracts, Cross-Platform Consistency, Accessibility Semantics, Interface Drift Detection

1. Introduction
Multi-platform UI engineering is an engineering challenge in part due to the requirements that a feature run on a range of input modalities, rendering engines, accessibility APIs, and release cycles for different device categories (desktop, TV, mobile, and wearables). This creates drift, the divergence of the experience across these categories as each minor update is made in look, behavior, and how it conveys meaning to users and assistive technologies. This drift is not just about visual consistency but behavioral and semantic as well, and it compounds over time as independent teams ship changes across surfaces without a shared contract for what consistency means.
The extent of the problem is well documented in accessibility literature and practice. A large analysis of home pages found that nearly all of the home pages contained at least one detectable accessibility failure, with an average of more than fifty failures for each of the pages under analysis [1]. Missing input labels and low color contrast are the most common failures. They are also the most persistent from year to year, and the industry is aware of and often complains about these failures. They are not implementation issues though; they are architectural issues. Since these are re-encoded by each team without a common source of truth, interfaces that contain strategies such as conditional rendering or state transitions are at a higher risk of semantic divergence.

The industry has also identified design systems as a solution to the issue of fragmentation. The majority of in-house product developers have been found to be maintaining a design system, moving from experimental development tools to operational infrastructure [2]. Adoption alone does not prevent drift. A design system without versioned contracts for visual decisions and interaction semantics provides consistency of appearance but not consistency of behavior. It is behavioral churn that raises the barriers for users dependent on keyboard navigation, a remote control, or assistive technologies.
It is more than simply refactoring shared assets or exchanging code snippets. It is principled architecture where

UI design and behavior decisions have versioned, verifiable contracts between UX designers and engineers. Design tokens offer a way to express visual intent as data that can be resolved to platform-native representations. So a semantic decision such as base action color, heading scale, or duration of motion can be used directly without recording intent at each target. Component contracts provide semantics, specifying the focus order, keyboard handling, pointer interaction rules, state transitions, and accessibility roles and relationships. These contracts allow different implementations of the platform to render the same component in different ways while remaining semantically identical from a user perspective and to formal verification tools.

The standardization ecosystem has caught up with this architecture. A community group specification for the design token interchange format — a technology-agnostic representation of token data, contextual variants, and resolution behavior — was published by the W3C Design Tokens Community Group in late 2025 [3]. Along with the drift detection tooling interoperability and governance that shared interchange formats enable, standardized data provides a compelling case for treating tokens as a versioned public interface instead of a build-time convenience.

Within cross-platform interoperability, publishing the community group specification like this lays the groundwork for a path forward for token standardization work that allows systems at scale to maintain visual and token consistency across tools and platforms [6], implemented at each stage of the pipeline through mechanical portability, drift detection, and visual contract auditing.

This, combined with the component contracts and versioned tokens, makes versioning of a cross-platform interface system possible, along with the ability to take measurements about quality, helping to prevent drift, and treating the parity of classes of devices as an active engineering property, rather than a goal achieved only via migration.

2. Design Tokens as Portable Visual Contracts

2.1 What Tokens Represent

Design tokens express values representing design decisions (such as color, typography, spacing, motion, shape, and elevation) as platform-agnostic abstractions. This does not mean centralizing the values used but rather the semantics that are applied to them, even if the platform's native primitives for implementing semantics may differ considerably from those of other platforms. In the end, this token is not just a hex color but also the fact that it is a primary interactive or critical user interface state or a disabled state. That decision may then be refactored into a CSS custom property, platform constant, or theme variable without the team needing to interpret the meaning of that value in context.

At scale, the difference between value and meaning becomes architectural; if visual decisions are localized or concrete, as when color constants are hardcoded, spacing constants are duplicated, or typography definitions are customized for each platform, there are as many representations of the same intent as there are surfaces. Instead, changing the main brand color, for example, would require finding every surface that uses the tokens and changing it, without a mechanical way to ensure that. Because intent is stored and transformed in one place, there is only one source of truth, and changes become reproducible.

This breadth is reflected in the tokens practiced by design systems. A survey of design practitioners found that, among the domains of tokens used in design systems, color tokens were the most common and used by nearly all of the maintainers of design systems; typography tokens were the next most common [2]. While not as universal, most respondents reported some use of spacing systems. The inconsistent patterns observed across platforms reveal an underlying high degree of variability that motivates the encoding of systematic spacing patterns [2]. The frequency of token systems reveals that they are more than mere theoretical constructs; they are operational infrastructure within product organizations.

2.2 Token Layering and Composition

The token layer system separates raw values from semantic intentions, arranged in a consistent hierarchy. Global tokens are defined at the primitives level, such as an entire color palette, a numeric spacing ramp, or a type scale. These are the building blocks of the visual language and are intentionally vaguer about their

semantic meaning. Semantic tokens alias these primitives to intent-based names, mapping a palette item to its intent, e.g., an interactive primary color, a surface's background, a destructive action, etc. Component tokens then bind those semantic tokens to pieces and states of components, such as the button label's intent, the border of an alert, or the outline of a focused input.

This approach reduces coupling at each interface in the three-layer model. Consumers of downstream components depend on semantic intent; changing all the raw color values in a palette (a full palette refresh) will cause any downstream component bindings to continue to work as long as semantic relationships are kept. Likewise, if a rebranding exercise mandates changing the color for the primary action, developers can just change the alias relationship between the semantic token and its global primitive without changing the definition for local usages.

Contextual resolution for theming, density variants, and accessibility modes (high-contrast and reduced-motion modes) defines the context within which token values can be resolved. Accessibility conformance studies have shown how semantic correctness of interface outputs must be mechanically verified rather than inferred from structural completeness, which reinforces that contextual resolution must also provide visually consistent and semantically verifiable outputs across all supported variants [5]. In addition to treating contextual modifiers as part of a chain of ordered inputs, the resolver also executes fallback chains, gracefully degrading partially specified variants to their nearest defined ancestor.

The governance impact of layering is just as powerful: enforcing token taxonomy boundaries means encoding every layout rule, each responsive breakpoint, and each conditional spacing fallback into the system, creating another large contract to maintain, making the system expensive to change and difficult to reason about. The goal for tokens is to have stable primitives that have semantic aliases. Layout and responsiveness are different architecture concerns. In design systems practice, an overload of tokens or failure to maintain the token boundary is the most common cause of system fragility, as teams can no longer tell stable semantic contracts apart from implementation-level constants [4]. Keeping the boundary allows the token contract to be readable , and the transform pipeline to be tractable.

Token Layer	Coupling Risk	Evolvability
Global Primitive Tokens	High	Low
Semantic Alias Tokens	Medium	Medium
Component Binding Tokens	Low	High

Table 1: Token Architecture Layer Coupling Risk by Layer Type [3], [4]

2.3 The Transformation Pipeline

Tokens only take effect if translation to platform-native outputs is pre-engineered in a reproducible, validated, and testable pipeline and not just an automated export process. The first step in the pipeline is to validate that the tokens satisfy the schema defined by the stable interchange format specification. This provides an understanding of how token files are structured, how to specify token aliases, and how to encode contextual variants [3]. The token validation process prevents malformed definitions and ambiguous token definitions from propagating into the platform outputs, catching invalid specifications as early as possible.

Once the graph is validated, a normalization phase removes alias chains, detects circular references, and builds a complete dependency model. The normalized graph is processed by the contextual resolver, which creates a set of resolved semantic snapshots for all combinations of theme, density, and accessibility variants. The snapshots are the most important intermediate artifacts of the pipeline. They capture what each semantic token resolves to for each supported context in a common format that is agnostic to any specific platform. Snapshots from a pipeline run are then diffed and can be used to detect semantic drift by reporting the semantic values that changed, what contexts the change is under, and whether the change is intentional.

The pipeline then fans into a series of platform emission stages, generating CSS custom properties and design specification files for the web, platform token artifacts for mobile, and platform-specific constrained outputs for television and wearable targets where rendering capabilities vary widely. Emission stages express the

semantic intent of the source token in the idiomatic way for their target platform, but the source of truth for what a token means is the resolved snapshot, not the platform output. This separation avoids problems where optimizations or constraints from one component may inadvertently affect the semantic contract.

The motivation behind the public release of the stable specification from a cross-platform interoperability standpoint is to explicitly frame the token standardization work in terms of allowing systems at scale to maintain visual and token consistency across tools and platforms [6], implemented at each stage of the pipeline through mechanical portability, drift detection, and visual contract auditing.

3. Component Contracts: Encoding Stable Interaction Semantics

3.1 The Contract Surface

In software components, the specification of a component contract includes public properties, intents, events, named states, and accessibility semantics (roles, names, relationships, and announcements). A component contract is the surface area between platform implementations and developers of products that build on top of it, as well as the mechanism for achieving or compromising cross-platform consistency. Making accessibility semantics first-class contract artifacts is not just an accessibility issue: it is the reality of today's web, where assistive technologies consume semantic structures that cannot be discovered by visual analysis of the webpage.

The rate of semantic failure in the production interface space is an important motivator for requiring enforcement at the contract level. Large-scale audits of production web interfaces have found that missing form input labels are among the leading forms of failures in terms of prevalence. More than one-third of all form inputs in the sample were found to be missing accessible labeling [1]. The architectural problem with this approach is that if labeling rules are determined by implementation decisions rather than a component contract, then the team will not be consistent in following them in the absence of a mechanical way to detect or prevent the problem from reaching users. A component contract for accessible name computation should specify how to compute a label, when to use fallback relationships, and whether visible labels must be presented.

This audit also found poor contrast in text in the majority of sampled pages, despite multiple years of increased awareness in the industry [1]. These results suggest that manual approaches to accessibility cannot keep up with the pace of interface design and development today. Contract-first systems express visual and semantic contracts at the edge of each component and use automatic checks to turn compliance into a structural quality rather than an after-the-fact assessment.

The visual contract includes the five dimensions of the contact surface and the component part and state token bindings to ensure that color, typography, spacing, and motion decisions are made by tapping into the common token system. The interaction contract specifies keyboard, pointer, tab focus, and activation behavior across devices and operating systems. The accessibility semantics contract provides roles, accessible names, accessibility relationships, and announcement behavior for dynamically updated elements that are meant to be accessible. The state contract describes named states and their behavioral and semantic relations. The error and recovery contract describes error detection, error presentation, and error recovery and the relation between error messages raised by the system and the inputs that triggered them.

The specification of a stable interchange format reinforces the ability to use visual contracts. A tool-agnostic, versioned format for token bindings can be machine-checked at the boundary of a component [3]. This allows conformance to a visual contract to be asserted as part of a pipeline build process, reducing the need to marshal correct information at the designer-developer hand-off. These five dimensions define what it means for a component to be correct, regardless of how it may be laid out in a particular platform.

It is often the case that implementation-defined semantics are less visible. For example, two identical components on a web page and on a television display may expose assistive technology to a different role or relationship structure or not expose change announcements when their states change. Since these changes are outside the range of sighted users or visual regression tests, they may persist across many releases. For contract-first systems, this class of divergence is visible because semantic obligations are expressed instead and checked by testing snapshots of the accessibility tree rather than visually inspecting.

3.2 Behavioral and State Contracts

Well-specified behavioral contracts around focus order, focus visibility, keyboard interaction, pointer interaction, and error presentation are an important property of platform implementations to independently check for conformance. Since many components appear visually consistent on different platforms without necessarily behaving consistently under keyboard navigation, remote control, or switch access, cross-platform inconsistency in these contracts is detrimental to accessibility. A pointer-triggered button or a modal dialog that operates only with visual focus but not with keyboard focus constitutes a complete interaction failure for no-pointer users, but the failure is not reflected in the user interface.

Auditing data on public interfaces shows the behavioral effects of uncontrolled semantic implementation. In a study of composite widget design patterns, some pages implementing navigation menu patterns introduced barriers to interaction by ignoring required keyboard interactions and missing relationships in the underlying document markup [1]. The data is especially revealing because navigation menus are one of the most widely deployed composite widgets and show that even well-studied, prominent patterns like this are improperly re-implemented when behavioral obligations are scattered. For example, a behavioral contract for a navigation menu component states the entire keyboard interaction model (arrow key navigation, escape key dismissal, home and end key behavior, and focus for open and close) as an interface property, not just a documentation recommendation.

In practice, without a stable behavioral contract, teams diverge in their local use of the common patterns of a design system, making inconsistent systems across surfaces. This is not due to laziness or negligence; the lack of a mechanical enforcement boundary means that design systems with only documentation of behavioral expectations make locally reasonable design decisions that together result in an inconsistent system. Contract-first architecture obviates this by making behavioral obligations part of the interface of the component and versioning and verifying them as visual and semantic contracts.

Error presentation is a behavioral contract dimension in that user control is affected when trying to recover from an error. If errors are visually rendered without exposing an error state to assistive technology, then assistive technology users are dimensionally unaware of the error state. Behavioral contracts define how errors are exposed through restrictions on the relationship between the input and the error message, the timing of error announcements, the availability of recovery actions, and the focus after errors are exposed. All these constraints can be verified by inspecting the accessibility tree or through interaction tracing, enabling automated regression detection instead of manual audits. Because this audit shows over a third of form inputs are unlabeled, error association rules must be encoded as contract obligations rather than implementation guidelines [1].

State contracts define named states of loading, empty, error, partial data, offline, and recovery and the expected user-facing behavior and semantic responsibilities of the developer when transitioning to and exiting those states. A cross-platform engineering challenge of state contracts is that the named states can have different platform-specific idioms, even with the same name. For example, the loading state for a web surface could be skeleton placeholders, for a television surface a spinning icon due to few layout options, or for a wearable surface a simple progress indicator with no room for anything else. All of these are valid implementations under a contract-first approach as long as they fulfill the same semantic requirements. Loading states should communicate progress to assistive technologies, avoid exposing interactive controls that are in a loading state, and reach a stable state with an announcement once the loading has completed.

As an interface gets larger and possibly more complicated, the number of named states and paths through these states may grow quickly. A study looking at interface complexity over time found that a home page's number of elements increased by a meaningful amount over a period of six years, consistent with this trend [1]. With an increasing number of elements in terms of state-dependent rendering, dynamic regions, and platform differences in state transition behaviors, the complexity of such configurations increases. Contract-first systems solve this complexity by explicitly stating what the state behavior is for all possible transitions at component boundaries in terms of state machines or transition tables in so-called contract artifacts. This allows state-based tests and snapshot coverage tests to be written rather than requiring the implementation team to understand what every state and state transition signifies.

One advantage of governance is that state contracts can be specified up front. This means that platforms can implement the state specification directly rather than have to infer it from implementations. A newly named state, a change in the behavior of a transition, or an extension of semantic obligations can all be expressed in terms of a contract delta. This considerably contributes to the traceability, reviewability, and communicability of state evolution across the teams consuming shared components , and reduces the effort of coordination that is typically required in large cross-platform systems [4].

4. Versioning and Governance

Token systems and component contracts are public interfaces in the same sense that software libraries and APIs are: they are stable surfaces that downstream components can depend on, and changing them has real costs. These properties imply that versioning is a first-class engineering discipline. It cannot simply be adopted after the decision is made to make a breaking change. Not versioning interface contracts is not a neutral omission; it is a debt that amasses quietly until the breaking change flows through all dependent surfaces without warning and forces uncoordinated remediation across every consumer team simultaneously.

However, this concern is not entirely misplaced: in a study of all the package releases in the largest software package index, one-third contained at least one breaking change. Furthermore, an important proportion of these breakages did not result in a version increment, as required by semantic versioning [7]. This is important for the field of token and contract governance because it means that although versioning schemes such as semantic versioning are compatible with breakage, breakage discipline is unlikely to be observed in communities that do not have automatic tooling to support token systems, and component contracts are particularly vulnerable given the lack of tooling maturity in the customary software package ecosystem. If automatic compatibility checks are not a part of the release pipeline, breaking changes (e.g., semantic token alias changes or interaction invariants) make their way silently to consumers, resulting in drift that is hard to detect and expensive to repair.

In general, breaking changes in a token and contract system can be separated into several based on what kind of governance they require. For example, renaming a semantic token is a breaking change because all downstream bindings that use the token will no longer work after the rename. Changing an interaction invariant, such as when keyboard interaction is enabled on a component, what focus management strategy to use for an open dialog, or when to announce dynamic state changes, are breaking changes. Target implementations that satisfy the previous interaction invariant would suddenly differ in a way that is not reflected in the new contract. Changing the accessibility technology exposure of a component (e.g., changing a role, removing a relationship, or changing the algorithm used for computing the accessible name) is a breaking change since assistive technology users trained for the previous semantic structure of a component would experience the new behavior without indication that there is a change. All these three categories of breaking changes require versioning, migration documentation, and a deprecation period during which consuming teams can update their service contracts before the previous one is withdrawn.

Change Type	Governance Overhead	Migration Required	Pipeline Enforcement
Semantic Token Rename	High	Yes	Yes
Interaction Invariant Change	High	Yes	Yes
Accessibility Exposure Change	High	Yes	Yes
New Optional Token Alias	Low	No	No
New Component State	Low	No	Partial
New Optional Contract Property	Low	No	No

Table 2: Breaking Change Categories and Governance Overhead [7], [8]

Additive changes (new optional token aliases, new component states, and new optional properties on existing contracts) have a lighter governance burden since they do not break existing implementations. Governance should distinguish between additive and breaking changes when reviewing contributions to the protocol and be able to automatically classify proposed changes to route them through the appropriate approval process based on their classification, e.g., automated compatibility checks. Further characterization of the downstream impact of breaking changes suggested that breaking changes are more likely to contribute to a slowdown of package upgrades by the dependent packages or manifest failures in dependent systems. This reinforces the systemic impact of their frequency [8]. Token and contract governance systems that minimize unnecessary breaking changes (for instance, through deprecation cycles, aliasing strategies, and additive extension patterns) directly reduce the coordination overhead incurred on consuming teams.

The key is achieving reasonable trade-offs between stability and change at scale. For example, the friction from governance settings where every addition of a token has to go to the full committee or all contracts everywhere have to respect the absolute worst-case backward compatibility is why teams use their own settings. Teams will always fork when maintaining the shared system becomes more expensive than maintaining their fork. These unmanaged forks are the primary failure mode of large design systems: they have the same fragmentation the system was built to prevent, except now they are invisible in the metrics of the shared system and will only be noticed in hindsight. Research into design system practices explicitly identifies this tension. When teams have shared goals, the most common reason that they diverge from shared components is friction related to contributing to and extending components [4]. Governance that does not provide legitimate means of flexibly extending components may drive divergence underground.

The best architectural strategy in this case is to create managed escape hatches. Scoped, temporary overrides that you document; signal as an intent to evolve the contract; and that clients know will be fixed at the end of the scope. A managed escape hatch is not carte blanche to ignore the contract and how it was designed. It is a way to explicitly preserve the contract's intent in a case where it doesn't yet cover a legitimate use case. Given no time limit, the escape hatches start to become permanent forks, and without documenting those changes, they turn into invisible technical debt. Escape hatches, not fed back into the contract evolution process, create an increasingly large gap between what the contract says and what production actually implements.

Some successful governance models share key properties: their ownership boundaries define a core team that maintains the stability of the underlying contract and contributor teams that build the surface. The governance proposal defines contribution paths that are better suited to the extent of the change (e.g., avoiding the overhead of a full governance review for a low-risk additive contribution while still applying a rigorous review for a breaking change). They publish an explicit and versioned changelog that highlights breaking changes, deprecations, and additive extensions so consumers know exactly how to plan their migrations. They also treat reuse metrics as a governance health signal: if reuse rates decrease, that means the governance process is putting more friction in than it is taking out, and one should investigate escape hatch usage patterns as a signal for whether and how to evolve the contract.

Evidence from surveys of design systems practice suggests that sustained use of shared system components, defined as sourcing more than half of the work product from the shared system, is one condition correlated with broader indicators of success [2]. There is also empirical evidence that sustained rates of component reuse may correlate with a measurable reduction of work spent on redundant implementation, as well as shorter onboarding times enabled via shared behavioral contracts between team members [9].

Health Signal	Healthy Indicator	Warning Indicator
Component Reuse Rate	More than half of work product	Declining trend
Escape Hatch Usage	Time-bounded and documented	Undocumented and permanent
Breaking Change Frequency	Low with automated detection	High without signaling
Contribution Pathway Friction	Proportionate to change scope	Uniform high overhead
Changelog Transparency	Versioned and explicit	Absent or informal

Table 3: Governance Model Health Signals and Indicators [2], [4], [9]

5. Evaluation and Drift Detection

A contract-first approach to evaluation means that performance is measured rather than a matter of ideology. This is more important when you note that the quality of a design system is often subjective: does it feel consistent? Are designers and engineers happy? Does the visual language feel consistent across all surfaces? While these signals can be useful, they do not scale, are not sensitive to semantic regressions, and do not offer any falsifiable evidence to support architectural decisions related to governance investment. A measurement-based framework replaces subjective judgment with observable, reproducible metrics that directly measure the properties that the system is trying to enforce in the contract.

5.1 Token Coverage

Token coverage indicates the proportion of colors, type spacing, motion durations, and elevation levels used in the design, as represented in tokens, versus ad hoc local constants hard-coded in components. High token coverage means visual decisions are flowing through the contract system and can be centrally governed, drift detected, and evolved. Broadly speaking, a low token coverage rate means that important portions of visual decision-making are done locally and outside of the contract environment in an ungoverned manner.

Two levels of granularity are used to record token coverage at the interface surface. Overall coverage is the percentage of all token-based values in the interface surface. Another way to evaluate a system's power is the system's semantic token utilization, the proportion of token uses that refer to semantic aliases. A system can have a high level of token coverage but a low semantic token. This means that the system has many values but not many meaningful values. These composed components, when linked to their tokens at paint time rather than design intent, cannot expect to benefit from a semantic redesign propagating through the system (i.e., changing which primitive a semantic alias points to). Design systems' case studies indicate that systems whose design composition is more than eighty-five percent shared tokens and components decrease design and implementation times on a cycle by twenty to fifty percent depending on the intricacy of the work being done [10]. These numbers should not be overgeneralized, but they do suggest relating token coverage targets to measurable productivity outcomes, rather than arbitrary thresholds.

To measure token coverage, build pipeline static analysis tooling is used to parse style outputs for web applications to understand if their values belong to a token or were defined locally. For platform-native targets, this requirement means using platform-specific tooling to examine style application at the component level. The pipeline must report coverage breakdowns per token domain, such as color coverage, typography coverage, or spacing coverage. Domain-level breakdowns indicate which aspects of visual decision-making are most exposed to ad hoc drift and are therefore most in need of governance.

5.2 Component Reuse Rate and Contract Coverage

In survey research of design systems practice, the threshold of utilizing more than half of the work product shared with the design system is cited as an indicator of design system success [2]. Research on the adoption of design systems finds that persistent reuse reduces the time to delivery of a feature and lowers the risk of cross-platform behavioral divergence when behavioral interpretation is not repeated at scale [9].

Contract coverage measures the proportion of components in the shared system that have explicit, machine-readable specifications for behavior, state transitions, and accessibility semantics. A component without a contract specification cannot be automatically verified for conformance, cannot be used as a reference implementation for new platform ports, and cannot be diffed for breaking changes in a structured way. Contract coverage is therefore a prerequisite for the drift detection pipeline: components that lack explicit contracts are invisible to automated semantic regression checks, and their behavior can diverge across platforms without triggering any pipeline signal.

Achieving high contract coverage requires investment in contract-authoring tools and processes. Component contracts must be expressed in formats that are both human-readable for governance review and machine-readable for automated verification. Research into design systems practice demonstrates that teams face a persistent tension between maintaining stable shared contracts and accommodating the flexibility demands of

individual product surfaces and that resolving this tension requires explicit contribution pathways rather than informal conventions [4]. For design system components, this translates into a concrete tooling requirement: contract specifications should be stored as versioned artifacts alongside component implementations, linked to automated test suites that verify conformance on every build.

Evaluation Metric	Measurement Method	Pipeline Automatable
Token Coverage	Static style output analysis	Yes
Semantic Token Utilization	Alias reference ratio	Yes
Component Reuse Rate	Screen composition audit	Partial
Contract Coverage	Specification artifact presence	Yes
Parity Coverage	Cross-platform semantic equivalence	Partial
Accessibility Regression Rate	Accessibility tree diff	Yes

Table 4: Evaluation Metrics and Measurement Feasibility in Contract-First Systems [2], [9], [10]

5.3 Parity Coverage and Drift Detection Pipeline

Parity coverage is the ratio of parity states of components across all platforms and variants of contexts that are supported. Parity is defined as equivalent outcome instead of equivalent implementation. For example, to achieve parity in the accessibility tree structure, interaction behavior, and state machine semantics, controls might leverage different rendering implementations, animation patterns, and layout idioms across platforms. Token parity is a necessary condition for parity. The same semantic token must resolve to the same semantic token value in the platform outputs. However, token parity by itself isn't sufficient, as it only guarantees fidelity in visual intent.

The drift detection runs on each of these artifact types in a continuous pipeline. Resolved token snapshots hold the complete set of semantic token values for each combination of a supported theme, supported density, and supported accessibility variant at the time of the pipeline run. Comparing token snapshots between pipeline runs can be used to determine if token value/context changes were part of an intentional contract change or drift. Accessibility tree snapshots capture the semantic tree exposed to accessibility tooling for all components in named states. Diffing these trees between versions detects semantic regressions, including changes in roles, names, and relationships or announcement behavior regressions not detected by visual regression tools. Interaction traces enumerate in order the focus moves, activation actions, and state transitions that result from running the canonical interaction scripts on the components. By diffing these over two different platforms, behavioral divergence with respect to the canonical interaction can be detected, such as out-of-order focus moves, keyboard activation, or state transitions.

The WebAIM Million report provides a further motivation for finding semantic drift as an active process rather than a discrete event: despite the number of usages of accessibility attributes steadily increasing in the sampled population over four years, the rate of finding errors remained high [1]. This pattern shows that adding semantic markup without validating correctness promotes worse accessibility outcomes and is correlated with increases in errors as the interface becomes more complex. A drift detection pipeline that validates semantic correctness on every build responds to this pattern to treat semantic quality as a continuous characteristic of an interface under regression rather than an audit snapshot.

Large-scale empirical studies of software package evolution also support the case for continuous drift detection. Studies of breaking change propagation in large package ecosystems showed that undetected behavioral divergence amasses much faster than can be detected by human audits, making them unable to catch regressions before they can affect downstream consumers [8]. For cross components, this finding means the pipeline must enforce drift detection at the cadence of the release pipeline, not the manual review cycle. Each pipeline run should archive a report of the detected deltas classified as confirmed breaking changes and needing governance review, deltas that are within the delta variance threshold, and new contractual violations that need to be remediated prior to release. Because this means drift is actionable, quality can be enforced in

line with the development process as opposed to being a separate audit step in the process.
Published experience reports of design system implementations also show that systems with automated conformance checking incur fewer semantic regressions in production and detect divergence across platforms faster compared to systems with a manual review stage [10]. This is a return on investment that pays dividends not only in quality outcomes: catching contract violations at build time rather than in production avoids the cost of investigating and coordinating responses to semantic regressions across multiple platform surfaces. On the cross-platform side, interface quality can be seen as a property of the contract, which is a continuously engineered outcome that grows steadily in response to investments and does not degrade unexpectedly due to independent platform releases [11].

Conclusion

Achieving cross-platform interface consistency at scale is an engineering problem, one that visual guidelines can only partly address. Design tokens enable a portable, evolvable representation of visual intent. They support visual decision-making as a versioned and mechanically transformable contract, rather than an implementation activity repeated on each platform. This makes visual decisions auditable, drift detectable at build time, and large-scale visual evolution more tractable without requiring manual coordination across every platform surface. Component contracts express cross-platform interaction semantics and state behavior. They define correctness conditions for a component, regardless of how it is rendered. Platform implementations are free to differ in their layout algorithm and rendering model as long as they remain semantically equivalent from the perspective of the author, assistive technologies, and automated verification pipelines. Alongside versioned tokens, component contracts move practice from informal convention toward verifiable, measurable interface engineering: quality becomes a property of the pipeline rather than an audit finding; breaking changes are more visible at the point of authorship rather than the point of downstream failure; and drift risk can be reduced before it reaches consumers via resolved token snapshots, accessibility tree snapshots, and interaction traces. This approach adds overhead and is most justified when surface-level variability and team scale are significant. The expected benefit is that drift detectability, quality measurability, and cross-platform consistency are sustained more continuously, rather than degrading between periodic reconciliation efforts after semantic drift has already reached production.

References

[1] WebAIM, "The WebAIM Million 2025 Report," WebAIM, 2025. [Online]. Available: https://webaim.org/projects/million/
[2] Sparkbox, "Design Systems Survey 2019," Sparkbox, 2019. [Online]. Available: https://designsystemsurvey.sparkbox.com/2019
[3] Design Tokens Community Group, "Design Tokens Format Module 2025.10," W3C Community Group Report, Oct. 2025. [Online]. Available: https://www.designtokens.org/TR/2025.10/format/
[4] Yassine Lamine and Jinghui Cheng, "Understanding and Supporting the Design Systems Practice," arXiv preprint arXiv:2205.10713, May 2022. [Online]. Available: https://arxiv.org/abs/2205.10713
[5] Markel Vigo, et al., "Benchmarking Web Accessibility Evaluation Tools," ACM W4A Conference. Available: https://dl.acm.org/doi/10.1145/2461121.2461124
[6] Kaelig Deloumeau, "Design Tokens Specification Reaches First Stable Version," W3C, Oct. 28, 2025. [Online]. Available: https://www.w3.org/community/design-tokens/2025/10/28/design-tokens-specification-reaches-first-stable-version/
[7] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning versus breaking changes: A study of the Maven repository," in Journal of Systems and Software Volume 129, July 2017, Pages 140-158. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0164121216300243
[8] L. Ochoa et al., "Breaking bad? Semantic versioning and impact of breaking changes in Maven Central," Empirical Software Engineering, vol. 27, no. 3, p. 65, 2021. [Online]. Available: https://arxiv.org/abs/2110.07889

[9] Cody Zuschlag, "Building Design Systems of Systems", in Nearform. Available: https://nearform.com/insights/building-a-design-system-of-systems/
[10] Caitlin Lee, "Building a Design System that Breathes with Headspace," Figma, 2021. [Online]. Available: https://www.figma.com/blog/building-a-design-system-that-breathes-with-headspace/
[11] ISO/IEC, "ISO/IEC 25010:2023 Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—Product Quality Model," ISO, 2023. [Online]. Available: https://cdn.standards.iteh.ai/samples