

Tartu University

Faculty of Social Sciences

Narva College

Study Programme “Information Technology Systems Development”

Andrii Stakhov

**AUTOMATED HEURISTIC LINTER AND RESOLUTION
ENGINE FOR THE FIGMA SCENEGRAPH**

Final Thesis

Supervisor: Andre Säask, M.Sc.

Narva 2026

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, Andrii Stakhov

1. Annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose "AUTOMATED HEURISTIC LINTER AND RESOLUTION ENGINE FOR THE FIGMA SCENEGRAPH", mille juhendaja on Andre Säask reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.
2. Annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace kaudu Creative Commons'i litsentsiga CC BY NC ND 4.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni.
3. Olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile.
4. Kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Andrii Stakhov 06.04.2026

CONTENTS

CONTENTS	3
INTRODUCTION	6
Problem	6
Goal.....	6
Tasks	6
Novelty	7
Methodology and Structure	7
1 RESEARCH	9
1.1 Background	9
1.1.1 ROI of UI Automation	10
1.1.2 The Gap Between Desired and Real States.....	11
1.1.3 SceneGraph Serialisation vs Document Object Model (DOM)	12
1.1.4 Rationale for Heuristic Automation	12
1.2 Foundational Governance and Heuristic Standards	13
1.2.1 Spatial Determinism and the 4pt Grid Heuristic	13
1.2.2 Auto Layout and Flex-Spacing Constraints.....	14
1.2.3 Compounded Alpha Channels and WCAG 2.2 Compliance	14
1.2.4 Physical Ergonomics and Fitts’s Law Application	15
1.2.5 Typographic Static Analysis and the “figma.MIXED” Constant	15
1.2.6 Semantic Hierarchy and Mitigating Naming Debt.....	16
1.3 Evaluating Existing Design Governance Tools.....	16
1.3.1 Stark: The Narrow Focus on Accessibility.....	16
1.3.2 Design Lint: Static Token Validation	17
1.3.3 Axe for Designers: Enterprise Snapshot Auditing.....	17

1.3.4	Summary of Technical Differentiators	18
1.4	Summary	18
2	SYSTEMS ARCHITECTURE & IMPLEMENTATION	20
2.1	Dual-Realm System Architecture	20
2.1.1	Communication and Data Flow	20
2.1.2	Performance Considerations.....	21
2.1.3	Thread Yielding and 60fps Stability.....	22
2.2	Abstract Syntax Tree Traversal and Data Linearization Logic	22
2.2.1	Implementation of Recursive Linearization via “flattenTree”	22
2.2.2	Incorporation of Recursion Constraints and Memory Protections	23
2.2.3	Context Propagation and Data Inheritance.....	24
2.2.4	Performance Benchmarking and Batching Results	24
2.3	Heuristic Engineering and Logical Resolution	24
2.3.1	Spatial Governance	24
2.3.2	Visual Accessibility	25
2.3.3	Ergonomic Auditing	25
2.3.4	Violation Ranking and Reporting Logic.....	26
2.3.5	Auto Layout and Flex-Spacing Determinism	28
2.4	The Safe-Mutation Framework.....	29
2.4.1	Design System “Circuit Breakers” and Token Integrity	29
2.4.2	Asynchronous Resource Management.....	30
2.4.3	Deterministic Mutational Fallbacks	30
2.4.4	Atomic Transactions and Undo-History Management.....	31
2.5	Real-Time UI Synchronization and Performance Limits	32
2.5.1	The Linear Health Score Algorithm.....	32
2.5.2	Asynchronous State Management vs. Telemetry.....	33

2.5.3	Visual Marker Capping for Canvas Performance	34
2.5.4	Deep-Linking and The Bridge to Selection.....	34
2.6	Empirical Evaluation and Comparative Performance Analysis	34
2.6.1	Throughput Benchmarking: Human vs. Machine	35
2.6.2	Accuracy and Detection Depth	35
3	CONCLUSION	37
	ÜLEVAADE EESTI KEELES.....	39
	REFERENCES.....	42

INTRODUCTION

Problem

User interface development remains the largest unautomated bottleneck in the development process (Si et al, 2025). Designers use atomic principles and design tokens to bridge the gap between visual intent and functional code (Frost, 2016).

This systematic approach, however, lacks programmatic enforcement. This thesis addresses “Design-to-Code” friction by proposing a machine-led validation layer. It treats the design canvas as a quarriable data structure instead of a static image.

Current design tools, function as passive editors that lack technical constraints, creating a gap between visual intent and production-ready code. Without automated enforcement, engineering teams must manually audit prototypes to fix sub-pixel errors and style inconsistencies. This reliance on human inspection leads to a slowdown in the transition from design to functional software.

One of the most popular design tools is Figma (Global Design Tools Survey, 2024) – a cloud-based vector graphics editor, which suffers from all the problems, mentioned above. The author uses Figma in his work as a developer and knows these limitations from his personal experience.

Goal

The objective of this thesis is to engineer an automated UI/UX design linter and resolution engine for the Figma environment. This tool will be a specialised plugin that evaluates design nodes against a team-defined profile of rules. It is an “active” engine that can mutate the frame and automatically fix errors.

Tasks

To reach this goal, the following technical tasks were established:

1. Building a two-way connection between the UI thread, which operates independently, and the secure design-environment sandbox. The objective is to design a batch-processing protocol that yields the main thread during heavy data scans,

preventing execution-blocking and maintaining a consistent 60fps refresh rate while traversing complex document hierarchies.

2. Constructing a method for deep-scanning document trees that identifies style and geometric violations. This requires implementing a certain recursion limit to protect system memory and enforcing a “source-first” audit policy, where technical debt in duplicated elements is identified for correction at the root component level to maintain design-system integrity.
3. Developing a safe-mutation framework, which involves creating logic to automatically correct design errors without disrupting existing system connections. The engine needs to check that node properties are not tied to global variables (tokens) before trying to fix anything automatically. It also must handle the simultaneous loading of typographic resources to avoid losing data when programmatically adjusting text.
4. Creating a visual monitoring system (dashboard) that calculates a “Health Score” based on the ratio of detected errors to total document nodes. By utilizing high-performance rendering techniques, the dashboard provides immediate perceptual feedback to the designer, turning the audit into a tool for continuous governance.

Novelty

The novelty of this work lies in the transition from “Read-Only” auditing to “Read-Write” resolution. Most existing tools list errors but leave the manual labour of correction to the user. This engine treats the visual canvas as a data structure that can be corrected programmatically.

Methodology and Structure

This work follows development-based research and a “bug pattern” methodology, which is a static analysis technique traditionally used to identify systematic errors in source code (Hovemeyer & Pugh, 2004) and may be adapted to evaluate visual interface nodes.

This research employs the Design Science Research (DSR) methodology, which centers on the development and assessment of an artifact, specifically a Figma plugin, intended to address a particular practical challenge: the automation of design system audits.

The study is organized into three discrete phases:

1. Analytical phase: an examination of industry governance and heuristic standards, such as the 4pt grid, WCAG 2.2, and Fitts's Law, as presented in Chapter 1. These qualitative standards were translated into quantitative metrics and automated heuristics suitable for algorithmic processing.
2. Development phase: the construction of the system architecture and core logic is described in Chapter 2.
3. Verification phase: evaluating the reliability of the automation through a Safe-Mutation Framework. This ensures that the proposed automated fixes maintain the integrity of the original design while correcting structural infractions.

1 RESEARCH

1.1 Background

Unlike traditional desktop editors, Figma's architecture is inherently complex, using WebAssembly (Wasm) for its graphics engine and WebGL, which is a JavaScript API that lets browsers render 2D and 3D graphics using the GPU without plugins, for rendering (Figma Blog, 2025).

Figma uses a multi-player WebSocket architecture, allowing real-time document manipulation across distributed teams. Technically, it functions as the "Single Source of Truth" for product teams, storing the visual blueprints for every digital touchpoint (Figma, 2024).

However, Figma's flexibility also presents its main technical challenge. Bexiga (2021) identifies this as a "communication gap" in which the visual "look-and-feel" overrides the actual technical constraints of the target ecosystem.

Architecturally, it is a passive canvas. It allows designers to place elements without spatial constraints at arbitrary coordinates with zero regard for semantic node data or structural logic. The result is a superficial visual image rather than a detailed technical specification.

Developers use tools like ESLint (static code analysis tool for JavaScript and TypeScript) and Stylelint (mainly for CSS, SCSS and Less) to find logical and syntax errors in the code. In contrast, designers work within a framework that prioritises visual effectiveness over technical limitations. According to Frost's (2016) "Atomic Design" methodology, a design system is a dynamic hierarchy of patterns, including atoms, molecules, and organisms, which requires careful, almost clinical, oversight. As a result, the absence of "programmable constraints" leads to a quick, often unnoticed, accumulation of technical debt.

Without an active linter to flag deviations, such as a 3 px margin where a 4 px token is required or a hard-coded hex value that ignores the system's colour palette, the integrity of the design file fragments is instantly compromised.

Manual UI adjustments are a technical liability. This creates a cumulative technical debt that forces downstream engineering teams to compensate for visual inconsistencies through manual CSS overrides and code patches.

This work aims to fix this by moving the rigour of engineering validation directly into the Figma phase. By treating design nodes as code-like entities that must pass a compilation check before being handed over to developers, we eliminate the ambiguity of the “drawing” and optimize the process of development.

1.1.1 ROI of UI Automation

Design governance is a quantifiable fiscal engine. According to data from McKinsey, companies with a high design index (MDI) have an average annual revenue growth rate of 10%, whilst the industry benchmark stands at just between 3% and 6%. Over a five-year period, this translates into cumulative revenue growth of over 60% compared with the benchmark period (see Figure 1).

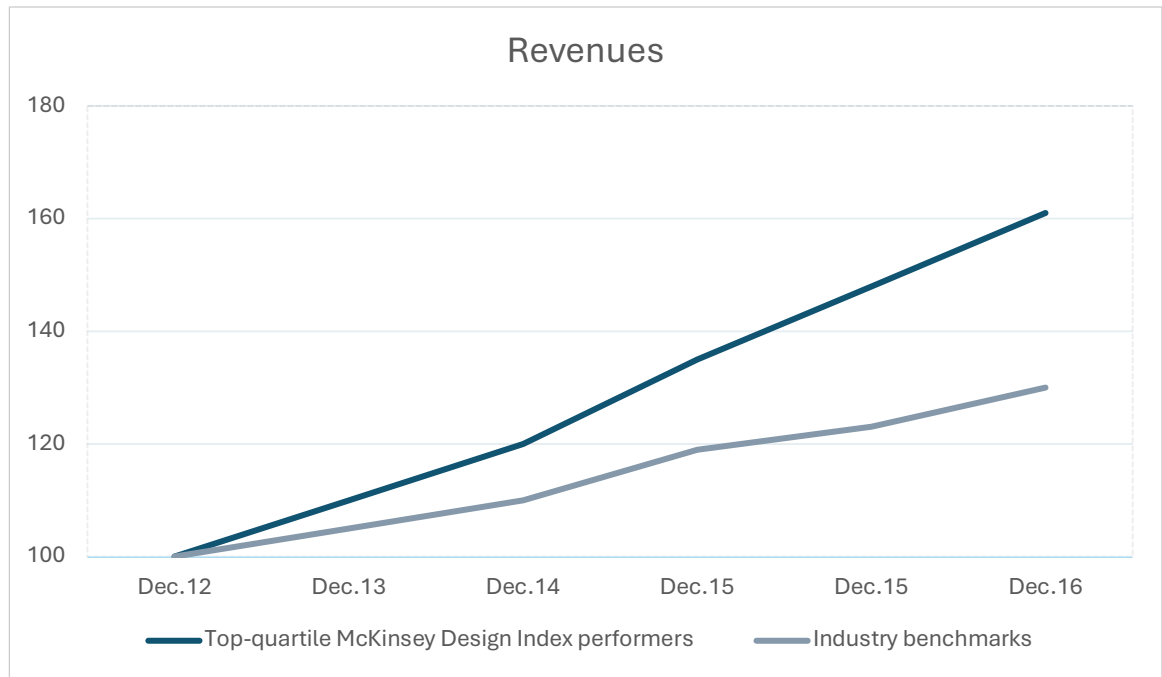


Figure 1. Comparison between top-quartile McKinsey Design Index companies and Industry Benchmark (McKinsey & Company, 2018)

UI is not a cosmetic skin, while it is a deterministic driver of market performance. While marketing teams sell design systems as accelerators, Oscarsson (2025) at Linköping University identifies the technical detail: Figma’s “passive canvas” architecture creates a semantic gap.

This inconsistency is significant, given that design artefacts do not exist in isolation from technical constraints. Artefacts are “disconnected from the technical constraints” inherent in the live stack. Developers bear the burden of extensive manual verification, compelled to interpret drawings that lack any internal layout logic. Consequently, developers frequently assume the role of forensic auditors during this stage.

They might manually validate layouts against CSS variable sets, checking for perfect alignment that a machine might verify in seconds. The absence of a simple heuristic engine that can solve problems by traversing the AST, which is a representation of code that shows its logical structure, leads to significant economic inefficiency.

But the core problem is the unreliability of human-led design audits. In a professional workflow, a reviewer must manually verify thousands of individual nodes for adherence to spacing tokens, typography scales, and accessibility ratios. Humans are poorly optimised for detecting small, inconsistent details in complex visual fields (Simons & Levin, 1997). This results in a discrepancy between the initial planning and the outcome.

1.1.2 The Gap Between Desired and Real States

There is a massive difference between the “Desired” technical state of a design project and the “Real” production stage.

The desired situation: in an ideal workflow, a Figma file functions as a mathematically precise blueprint. Every spatial value adheres to a strict 4pt grid. Each typographic element is constructed using a modular scale, a system governed by Design Tokens (W3C Design Tokens Community Group, 2025).

Furthermore, all colour pairings comply with WCAG 2.2 Level AA visual accessibility criteria. These guidelines are intended to foster interface predictability and user confidence, as outlined in the Apple Human Interface Guidelines (AHIG) (Apple Inc., 2024).

The real situation: designers often use generic names like “Frame 5032” because it is faster than semantic labelling. They might use “hard-coded” hex values because the search for a specific colour token takes too long. They might create mobile buttons that are 32 px tall, ignoring the 44x44 pixel minimum requirement for physical usability (AHIG) (Apple Inc., 2024). This leads to a gap between design and final code, transforming the developer into a forensic investigator.

1.1.3 SceneGraph Serialisation vs Document Object Model (DOM)

The technical conflict stems from how Figma serialises its visual data. As a cloud-native graphics engine, Figma operates on a proprietary SceneGraph, which is a hierarchical tree of objects optimised for GPU-accelerated rendering via WebGL (Figma Blog, 2025). In this environment, every interface element is treated as a discrete vector object defined by an “absoluteBoundingBox” (Figma’s node objects). This model is fundamentally “flat” regarding layout logic; it records where an object is, but not why it is there.

On the contrary, the target environment (the browser) uses the Document Object Model (DOM), which is a dynamic, state-based layout engine. In the DOM, an element’s position is a calculated result of its relationship to its parent container, sibling flow, and inherited CSS properties. When a designer alters an element on a static canvas, they produce coordinate data devoid of relational limitations (Figma API Developer Docs, 2026).

As a result, the transfer process functions as a lossy data compression. The “semantic gap” (Oscarsson, 2025) represents not just a communication breakdown but a fundamental structural disparity; the SceneGraph’s JSON output omits semantic attributes such as flex-basis, z-index logic, and grid-template-areas, which are essential for programmatic reconstruction. This lack of metadata necessitates a manual reverse-engineering phase, thereby introducing non-deterministic interpretation into the engineering process.

1.1.4 Rationale for Heuristic Automation

A machine-driven validation layer is essential for maintaining computational integrity during the handoff procedure. Automating user interface development might be focused on “code generation”, specifically the process of converting visual designs into CSS. However, this method often presents challenges related to the quality of the input data (Si et al., 2025). This thesis posits that the primary obstacle lies not in the code generation itself, but rather in validating the design’s source material.

By treating the Figma document as an AST (Section 1.1.1) that can be parsed and queried, we can identify “bug patterns” in the design logic before they reach the developer. This approach shifts the UI/UX workflow from a visual search to a certain audit. An automated engine can quickly evaluate the “fillOpacity” and “nodeOpacity” of every leaf

node in a 15-level recursive tree, ensuring that the WCAG 2.2 perceivability standards are met mathematically, not only visually.

This “shift-left approach” (moves testing, quality, and security checks earlier in the software development lifecycle (SDLC), often starting in the design phase instead of just before deployment) ensures that the design artefact is “pre-compiled” for the developer, effectively bridging the semantic gap through algorithmic enforcement rather than human intervention by using specific parameters.

1.2 Foundational Governance and Heuristic Standards

The functional success of an automated linter is dictated by the mathematical rigidity of its underlying heuristics. While section 1.1.3 explains the architectural misalignment between SceneGraph serialisation and browser-based rendering, this section formalises the deterministic rules required to enforce system integrity. In a high-velocity production stack, governance cannot rely on aesthetic suggestions, while it requires hard-coded constraints (Lamine & Cheng, 2022).

1.2.1 Spatial Determinism and the 4pt Grid Heuristic

The most common mistake of a non-constrained design environment is the generation of non-integer coordinate data. Figma’s internal graphics engine, optimised for vector precision, allows nodes to exist at arbitrary, floating-point positions (e.g., $x=103.532$ px).

When this data is transferred to a production CSS environment, it triggers anti-aliasing artefacts. Browsers attempt to render these fractional offsets by distributing colour values across physical sub-pixels, resulting in a visible “blur” that compromises UI sharpness and layout predictability.

The GRID Module will solve this by applying a strict geometric validator to the “absoluteRenderBounds” of every SceneNode – a basic element inside a Figma document’s SceneGraph. The heuristic employs a precision delta check:

$$isInvalid = Math.abs(val - Math.round(val)) > 0.001$$

If a coordinate fails this check, it is flagged as a problem. Furthermore, the module enforces the 4pt/8pt modular grid is a standard spatial constraint that ensures vertical rhythm and typographic alignment (Mathew, 2025). By applying a modulo operation to

properties such as “itemSpacing” and padding, the engine eliminates the “verification tax” by replacing manual coordinate adjustment with algorithmic rounding. This ensures the design artefact functions as a mathematically valid blueprint.

1.2.2 Auto Layout and Flex-Spacing Constraints

In modern front-end development (CSS Flexbox and CSS Grid), layouts are built not on absolute coordinates, but on dynamic spacing: internal padding and gaps between elements (Simmons, 2018). In the uncontrolled environment of a graphic editor, manual control of these parameters inevitably leads to the appearance of odd or fractional values (e.g., gap: 17px), which disrupts the vertical rhythm of the interface.

From a technical perspective, chaotic padding makes it impossible to use spatial design tokens (spacing variables). To function as a technical specification, the layout’s spatial relationships need to be controlled through algorithms.

This requires a rule that breaks the audit into two levels:

1. Macro-space: the overall space between sections (“sectionSpacing”), which must exactly match the reference token.
2. Micro-space: local component spacing, which must adhere to the mathematical rule of evenness ($\text{gap} \% 2 == 0$) or alignment to a 4 px grid.

This programmatic constraint eliminates ambiguity when passing the layout to development and guarantees the mathematical purity of the generated CSS code.

1.2.3 Compounded Alpha Channels and WCAG 2.2 Compliance

Evaluating colour contrast in a collaborative tool is presents high variability in human performance because Figma compounds opacities across the hierarchy. A node’s perceived brightness is not a function of its hex code, because it is an interpolated value derived from its “fillOpacity” and its parent “nodeOpacity”.

Manual auditors frequently fall victim to Simons & Levin’s (1997) documented that the human eye fails to detect marginal luminance changes in complex visual fields.

The WCAG Module bypasses this biological limitation by calculating the effective alpha (“totalOp”) through the product of all active opacity channels:

$$totalOp = \prod_{i=1}^n opacity_i$$

By traversing the AST (section 1.1.1), the engine computes the true background-to-foreground contrast ratio against WCAG 2.2 Level AA standards (Web Content Accessibility Guidelines, 2024). This moves accessibility verification from a subjective check to a certain pass or fail state. If the resulting contrast ratio for normal text falls below 4.5:1, the node is marked as a perceivability failure at the source, preventing inaccessible designs from reaching the deployment stage.

1.2.4 Physical Ergonomics and Fitts’s Law Application

UI governance should extend beyond visual design, specifically concerning physical usability, particularly the dimensions of touch targets. Based on Fitts’s Law, the time it takes to reach a target depends on the distance to the target and its width (Figma Design Basics, 2024). In mobile viewports, the “width” represents the critical threshold for human thumb interaction.

The TOUCH Module enforces the AHIG (2024) mandate for a minimum 44x44 px hit target. The engine audits the width and height properties of all interactive elements (buttons, icons, toggles). If an interactive node satisfies the condition:

$$\min(\text{width}, \text{height}) < \text{targetSize}$$

This automated check recovers sprint capacity previously lost to late-stage usability fixes by catching “orphaned” touch targets before handoff.

1.2.5 Typographic Static Analysis and the “figma.MIXED” Constant

Typography is a primary source of “systemic rot” due to the high frequency of manual overrides. The TYPO Module acts as a static analyser for text nodes, intercepting three specific “bug patterns”.

First, it identifies nodes where Figma returns the “figma.MIXED” constant (used when a property has multiple different values at once) for “fontName” or “fontSize” properties, which is a state indicating that a single text layer contains inconsistent, unlinked styles.

Second, the engine validates the “fontSize” against a strict array of allowed integers defined in the dynamic JSON profile, ensuring absolute adherence to the design system’s tokenized scales.

Third, it identifies hardcoded hexes where a designer has manually entered a colour value that matches a design token visually but is not programmatically linked to a global variable. By verifying these properties through the AST (section 1.1.1) traversal, the engine ensures that typography remains a structured, tokenised data source rather than a collection of unlinked vector artefacts.

1.2.6 Semantic Hierarchy and Mitigating Naming Debt

The final part of the framework involves fixing “Naming Debt”. As mentioned in section 1.1.2, using default names significantly contributes to problems during handoffs.

The NAMING Module uses a Regular Expression (RegEx) engine to scan the SceneGraph for non-semantic identifiers.

The engine identifies nodes that have not been renamed from their original state. This requires the designer to provide a label, which helps create better CSS classes and automatically generates documentation. As a result, the need for engineers to manually interpret things is reduced, ensuring the SceneGraph acts as a structured data source.

1.3 Evaluating Existing Design Governance Tools

The “Design-to-Code” gap has led to the development of several industry-standard tools, yet a technical audit reveals a recurring architectural ceiling: these solutions operate as passive notification layers rather than active remediation engines. To define the necessity of the proposed heuristic-led mutation layer, we must explore the functional limitations of the three primary competitors in the Figma ecosystem.

1.3.1 Stark: The Narrow Focus on Accessibility

Stark is the industry benchmark for WCAG (Web Content Accessibility Guidelines) compliance. It is designed to ensure that designs are usable by people with disabilities by checking colour contrast and screen-reader order (Stark Lab, Inc., 2024).

Therefore, Stark is strictly an accessibility auditor. It is “blind” to the technical engineering requirements of a layout. It does not check for spacing consistency, 4 pt grid alignment, or sub-pixel rendering errors. A design could be “Stark-compliant” while remaining technically broken for a developer due to chaotic padding and non-integer coordinates.

Distinction from the proposed engine is that unlike Stark, which only monitors visual accessibility, this engine treats the geometric integrity of the file as a priority.

1.3.2 Design Lint: Static Token Validation

Design Lint is a community-driven tool meant to find unlinked layers that do not use defined styles or variables. It is a vital tool for design system hygiene, yet it lacks spatial control.

Practically, Design Lint is a “token checker”. It flags a missing colour style but has no understanding of physical ergonomics. It cannot detect if a button height is 43.92px (a sub-pixel error) or if a margin violates the 4pt spacing rule. It treats the design as a list of properties rather than a coordinated data structure (Destefanis, D., 2023).

Design Lint generates a passive list of errors that the designer must correct manually, one by one. The proposed engine implements a “Fix All” heuristic. Rather than simply pointing out sub-pixel misalignments or spatial inconsistencies, it calculates the intended geometric logic and programmatically mutates the SceneGraph. This replaces the designer’s manual adjustment of coordinates with an automated process of batch defect removal

1.3.3 Axe for Designers: Enterprise Snapshot Auditing

Developed by Deque Systems, Axe for Designers attempts to bring automated web testing into Figma. It is a powerful tool for enterprise governance but suffers from architectural latency.

Axe does not operate on the SceneGraph. It requires the designer to take a snapshot of the document, which is then sent to a cloud-based engine for analysis. This makes it a “post-design” tool, which is something used at the end of the process rather than a real-time assistant that prevents errors during the creation phase (Deque Systems., 2024).

Axe focuses on semantic web structures but ignores Human-Computer Interaction (HCI) laws like Fitts’s Law (Figma Design Basics, 2024). It might approve a small, hard-to-click icon if it has an “alt-text” label. Proposed engine is a live AST tracker. It operates directly within Figma’s memory, enforcing minimum touch-target sizes and spacing rules in real-time. It doesn’t just report on the “past” state of a file, but it actively shapes the “current” state through direct API writes.

1.3.4 Summary of Technical Differentiators

Table 1. Comparison of different Design Governance Tools

Feature	Stark	Design Lint	Axe	Proposed Engine
Primary Scope	WCAG / Accessibility	Token Links	Semantic Web	Geometric & Systemic Integrity
Pixel/Grid Audit	No	No	No	Yes (4pt / Sub-pixel)
Logic Type	Passive Warning	Static list	Cloud Snapshot	Live Heuristic Analysis
Correction	Manual	Manual	Manual	Automated (Mutation API)
Alpha Resolve	Basic	No	No	Compounded

Table 1 summarizes why existing industrial standards fail to solve the “Design-to-Code” bottleneck compared to the proposed research.

The fundamental flaw in current tools is that they function like a spell-checker that highlights a mistake but refuses to provide the correct spelling. They identify the “what” but ignore the “how”. By contrast, the proposed system moves from passive auditing to active mutation. It treats design errors not as visual mistakes, but as corrupted data points that can be programmatically cleaned before they ever reach the engineering phase.

1.4 Summary

The chapter demonstrates that Figma’s architecture, with its passive canvases, Scene-Graph serialisation and lack of semantic constraints, creates a disconnect between visual design and engineering logic. This leads to manual and unreliable checks of spacing, typography, naming, contrast and accessibility.

It is noted separately that Figma’s plugin architecture has its own production limitations and does not allow heavy-duty checks to be performed directly.

Besides that, an analysis of existing tools (Stark, Design Lint, Axe) shows that they remain passive validators and are unable to resolve systemic errors. Therefore, a new

heuristic engine is required, one that analyses the document as the AST and automatically detects and corrects errors in real time.

2 SYSTEMS ARCHITECTURE & IMPLEMENTATION

2.1 Dual-Realm System Architecture

The plugin’s reliability will be rooted in a Dual-Realm Architecture, which is an environment split dictated by Figma’s security and performance model. Because the tool must manage both a high-fidelity dashboard and a massive vector database (the Scene-Graph), it runs in two isolated execution environments that do not share memory and communicate strictly through an asynchronous bridge (Figure 2).

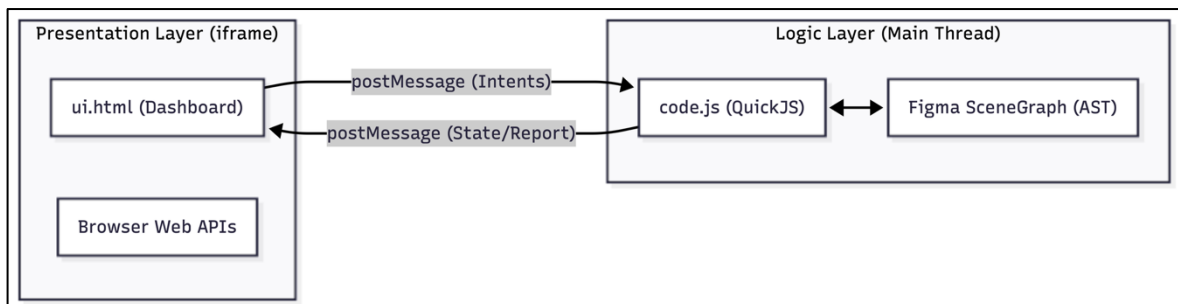


Figure 2. Dual-Realm System communication

The presentation layer, contained within “ui.html” and executed within a conventional browser inline frame, possesses access to the DOM, (CSS), and Web APIs.

This access is what allows the dashboard to be displayed and the live interface updates to be controlled. Important, that it is separate from the document’s design data. This separation prevents any accidental changes to the layers.

The logic layer, which is implemented in “code.js”, runs within a QuickJS sandbox on Figma’s main thread. This layer has a full access to the Figma Plugin API and the document’s Abstract Syntax Tree, but it cannot access browser specific. By separating the logic layer from the user interface, computationally intensive tasks, such as analysing many layers, are prevented from interfering with user interactions.

2.1.1 Communication and Data Flow

Since the realms share no memory, they communicate through an asynchronous inter-process communication (IPC) bridge, sending messages back and forth without blocking the interface, using the “postMessage” protocol. Data is serialised into JSON payloads, which ensures reliable and predictable communication. User actions are sent to the backend as intents, processed, and then returned to the UI as state snapshots.

This cycle calls “Intent-State” method and it guarantees that the interface always reflects the verified state of the document, preventing “state drift” where the UI displays outdated or corrected errors (Figure 3).

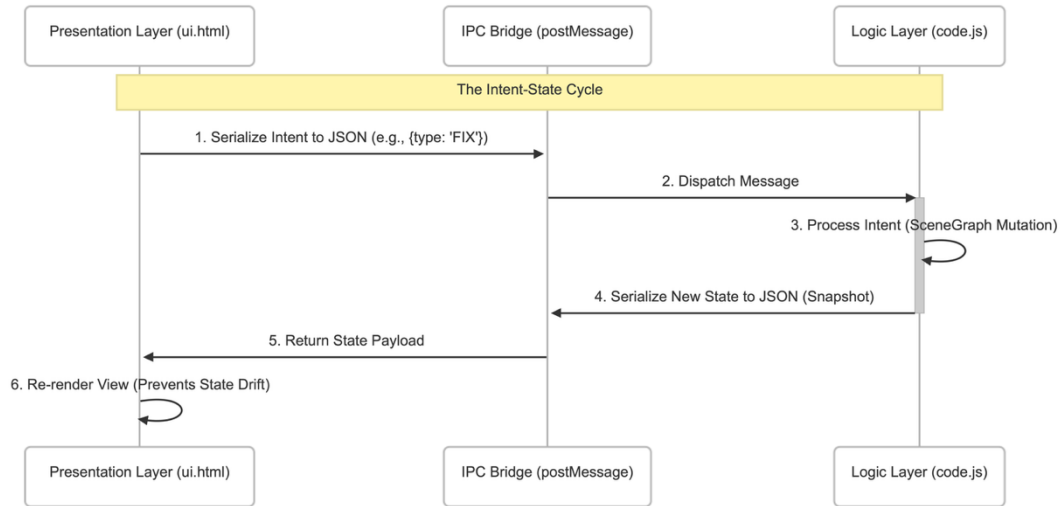


Figure 3. The asynchronous Intent-State communication cycle across the IPC bridge, illustrating JSON payload serialization and state drift prevention.

2.1.2 Performance Considerations

A central design choice is the avoidance of additional frameworks and building with just Vanilla ES6+ JavaScript. This eliminates initialisation overhead, allowing the plugin to reach an interactive state almost instantly. In addition, the QuickJS environment has limited memory; every byte consumed by a framework reduces the resources available for document parsing. By eliminating framework bloat, the plugin maximizes heap availability, allowing large-scale SceneGraph traversals without risk of stack overflow or memory shortage.

The system’s design requires a specific information flow. The user interface creates intents, which the backend then uses to change the document. After these changes, a complete state snapshot is sent back to the interface.

This design approach prevents the UI from performing partial updates or making assumptions about the document’s state. Consequently, all visual elements, including the Health Score, accurately represent the document’s current condition. This allows the plugin to deftly manage responsiveness, stability, and processing power.

2.1.3 Thread Yielding and 60fps Stability

At the same time, Figma’s main thread, a single-threaded resource, is shared. A simple, synchronous loop going through thousands of SceneNodes would cause the application to freeze, leading to a “Script Unresponsive” warning. To solve this, the engine uses a thread-yielding method, which is implemented through Asynchronous Batching.

By adding a controlled pause to the recursive audit cycle, the engine gives control back to the host application:

```
for·(let·i·=·0;·i·<·flatNodes.length;·i++)·{  
  //·Manual·yielding·every·300·nodes·to·prevent·main-thread·  
  lockup  
  ··if·(i·>·0·&&·i·%·300·===·0)·{  
    await·new·Promise(resolve·=>·setTimeout(resolve,·2));  
  }  
}
```

Figure 4

2.2 Abstract Syntax Tree Traversal and Data Linearization Logic

In the implementation phase, it became clear that direct access to Figma’s layer hierarchy is not enough to meet the “Spatial Determinism” and “Heuristic Automation” requirements described in sections 1.2.1 and 1.1.4. Therefore, this section explains the development of a specialized traversal engine. This engine will be designed to parse the document’s AST and then convert it into a fast data stream.

2.2.1 Implementation of Recursive Linearization via “flattenTree”

It was identified that the hierarchical nature of the Figma SceneGraph (the nested layer tree) creates significant computational overhead when evaluating multiple heuristic rules simultaneously. To address this, a Depth-First Search (DFS) algorithm was engineered and implemented within the flattenTree function to systematically traverse the hierarchy from the root down to the deepest leaf nodes.

The central goal of this operation was to convert the nested tree structure into a continuous, flat array of objects. By restructuring the audit process, moving away from repeated recursive tree traversals toward a single linear iteration, the system maintains an asymptotic time complexity of $O(n)$, where n represents the total number of

processed nodes. Consequently, execution time scales linearly with document size. This architectural optimization is crucial for handling large-scale enterprise files, ensuring stability and preventing stack overflows.

2.2.2 Incorporation of Recursion Constraints and Memory Protections

Testing of extensive design systems revealed the potential for a stack overflow within the QuickJS sandbox. As a result, two technical limitations were explicitly implemented within the traversal logic:

1. The 15-level recursion limit: it is an engineered constraint designed to balance QuickJS memory safety with interface structural standards. Architectural analysis based on Atomic Design principles (Frost, 2016) demonstrates that production-ready components rarely exceed 10 levels of nesting. Depth exceeding 15 levels is a reliable indicator of structural degradation (e.g., redundant layer grouping).
2. The instance-boundary policy (“!isInstance”): as established in section 1.1.2 regarding the accumulation of naming and technical debt, modifications within component instances lead to system fragmentation. Therefore, a logic-gate was implemented to terminate the recursive walk at the boundary of any INSTANCE node. This forces the engine to target the master component, ensuring that remediations are applied to the source and expanded globally.

Table 2. Logic Comparison: Component vs. Instance Auditing

Node Type	Traversal Depth	Remediative Agency	Rationale
Master Component	Full (Recursive)	Primary (Read-Write)	Preserves systemic integrity at the root level.
Instance	Surface Only	Secondary (Read-Only)	Prevents the creation of local style overrides.
Frame / Group		Primary (Read-Write)	Enables the enforcement of layout and grid rules.

Table 2 illustrates the architectural distinction between the processing of master components and their corresponding instances throughout the linearization phase.

2.2.3 Context Propagation and Data Inheritance

To enable the linter to distinguish between varying design environments (e.g., determining if a button resides within a mobile viewport or a desktop sidebar), a context propagation system was developed. An `AuditContext` object was integrated into the “flattenTree” function, which is passed down the chain from parent nodes to their children during the recursive traversal.

As a result, every node in the final flat array carries specific metadata regarding its surroundings. This automation facilitated the enforcement of “Physical Ergonomics” (section 1.2.4), as the engine can now dynamically apply the correct hit-target constants (e.g., the 44 px minimum) based on the inherited viewport metadata of the parent frame.

2.2.4 Performance Benchmarking and Batching Results

To assess the effectiveness of the Asynchronous Batching strategy, we collected execution metrics across various document sizes. The implementation of a thread-yielding mechanism, specifically, returning control to the event loop for 2 milliseconds after processing every 300 nodes was found to be effective in maintaining a consistent refresh rate of 60 frames per second, even under conditions of high computational demand.

2.3 Heuristic Engineering and Logical Resolution

Once the document was linearized into a flat dataset, the development focus moved to the construction of the heuristic engine. This module was designed to act as a series of logic gates that evaluate every node in the array. Rather than performing abstract calculations, the engine was programmed to compare physical design properties against hard-coded industry standards. The following sections detail the logic behind the spatial, visual, and ergonomic auditors.

2.3.1 Spatial Governance

The implementation of the grid validator was driven by the “Design-to-Code” parity requirements established in section 1.1.4. In professional frontend development, layouts are typically constructed using the 4 px spacing system to ensure cross-platform consistency (section 1.2.1). To automate this, a geometric check was integrated into the audit loop.

During the module’s testing phase, a notable technical impediment was observed: sub-pixel jitter. Given that Figma contains fractional coordinates (for instance, a button at 10.003px), a rigorous mathematical assessment would incorrectly identify numerous “false positives” that are imperceptible to the human eye. To address this issue, a rounding tolerance was introduced. The engine was designed to disregard any deviation less than 0.001px, thereby directing the user’s focus exclusively on significant misalignments. This choice was essential in mitigating “user fatigue” and ensuring the tool’s continued utility in routine design tasks.

2.3.2 Visual Accessibility

The most complex part of the heuristic engineering involved the WCAG 2.2 Contrast Auditor. The objective was to ensure that all text remained readable for users with visual impairments. While calculating contrast between two solid hex codes is a solved problem, the real design environment presented a major hurdle: alpha-transparency.

When a designer places white text over a semi-transparent black button, the “true” background colour is a blend of the button and whatever lies beneath it. To handle this, a layer-stacking algorithm was developed.

If a background layer was found to have less than 100% opacity, the engine was programmed to traverse up the AST to evaluate parent fills and sibling z-index stacking to find the “solid base” colour. The colours were then combined using alpha-compositing (mathematical blending of a semi-transparent layer with the background) to determine the actual contrast. This method ensured that the linter showing exactly what a user would see on a screen, rather than just the raw code values.

2.3.3 Ergonomic Auditing

Beyond visual aesthetics, an ergonomic module was implemented to address the physical reality of mobile interaction. This was based on the problem, where interactive elements (like icons or small text links) are too small for a human thumb to hit accurately.

The engine utilizes the context inheritance (section 2.2.3) to determine the device type. If a parent frame is identified as a mobile viewport, a 44 x 44 px minimum threshold is applied to every interactive node. This logic is derived from the Apple Human Interface Guidelines (AHIG). By automating this check, the development process moved from

“guessing” to “verification”, ensuring that every design exported from the system is physically usable before it ever reaches a developer.

Table 3. Heuristic Constants and Logic Thresholds

Audit Type	Target Value	Logic/Reference	Implementation Goal
Grid Step	4 px	Section 1.1.4	Code-ready alignment.
Standard Contrast	4.5:1	WCAG AA	Readability for small text.
Large Text Contrast	3.0:1	WCAG AA	Readability for headers.
Touch Target	44 px (min)	Apple HIG	Physical usability (mobile).

Table 3 presents the primary heuristic constants and logical thresholds employed by the validation engine. Each entry delineates a particular design constraint, such as grid spacing, colour contrast, or touch target dimensions, corresponding to a recognized reference or standard.

2.3.4 Violation Ranking and Reporting Logic

Once the engine identifies a deviation from the established heuristics, it must categorize the error to facilitate efficient user triage. To prevent the designer from being overwhelmed by an undifferentiated list of issues, a binary severity categorization was integrated into the auditing loop. As the engine processes the AST, it appends a strict severity flag, either “ERROR” or “WARNING”, to each generated report object. This allows the presentation layer to separate deterministic violations from heuristic ambiguities.

1. **Errors (Action Required):** this classification applies to strict, mathematical violations of the established design tokens and physical limitations. It includes sub-pixel rendering errors (fractional coordinates), unapproved typography (non-standard font sizes or families), missing layout margins, inadequate touch targets, and WCAG contrast failures over solid backgrounds. These represent critical technical debt that compromises code implementation or accessibility standards. They are prioritized at the top of the plugin’s UI to be resolved programmatically (Figure 5).
2. **Warnings (Manual Review Needed):** this categorization is exclusively reserved for non-deterministic scenarios where algorithmic validation is impossible without human context. Within the current engine, a warning is only triggered when a text

node is placed over a dynamic background, such as a rasterized image or a complex gradient. Because the engine cannot mathematically guarantee a constant contrast ratio across a varying pixel matrix, it flags the element for manual review rather than forcing an automated, potentially destructive fix.

The linter’s usability hinges on this categorization logic. By serializing the flagged results into a structured JSON report payload, which is then transmitted via the “postMessage” protocol, the system facilitates immediate dashboard rendering within the UI thread. This approach obviates the necessity of recalculating intricate background-compositing mathematics with each user interaction or filter application on the audit list.

The practical application of this categorization logic is captured in Figure 5, which displays the plugin’s interface during a live document audit. By separating deterministic “Errors” from context-dependent “Warnings”, the UI provides a clear hierarchy of needs.

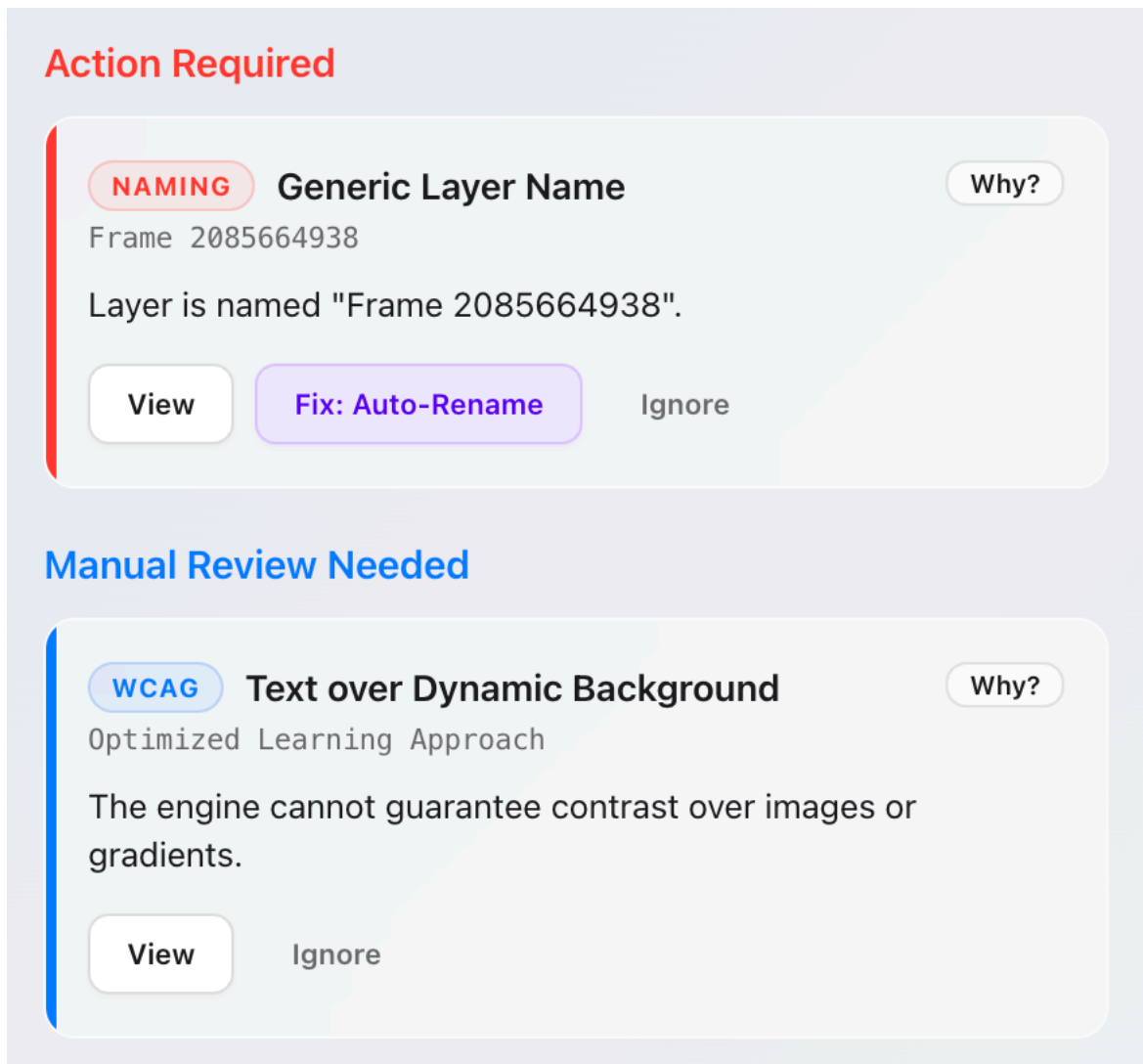


Figure 5. Graphical interface representation of the “Errors” and “Warnings”

2.3.5 Auto Layout and Flex-Spacing Determinism

Based on the principles of spatial determinism (section 1.2.2) the SPACING module programmatically validates the Auto-Layout (Figma’s proprietary implementation of the CSS Flexbox algorithm) parameters of containers (“layoutMode !== “NONE””). The engine scans the AST and divides the checks into two structural levels:

1. Macro-rhythm (section indents): the algorithm identifies root containers (nesting depth ≤ 2 or the presence of the semantic tags main, body, wrapper in the name). If the spacing between elements (“itemSpacing”) exceeds 64 px (Curtis, 2016), it is classified as a macro-indent. This value is strictly checked against the “sectionSpacing” token from the dynamic JSON configuration (e. g., 104 px).

2. Micro-rhythm (indent evenness): for all local Flex containers, the “itemSpacing” property and the array of internal padding values (“paddingTop”, “paddingBottom”, “paddingLeft”, “paddingRight”) are checked. The engine applies a modulo operation ($\text{val} \% 2 \neq 0$) to identify odd values that disrupt the base grid.

To protect the design system architecture, the “isBoundToVariable(node, propertyName)” safeguard is called before any write operation. If the indentation is strictly bound to a variable (design token), the engine blocks the mutation, forcing the designer to correct the logic at the token library level.

2.4 The Safe-Mutation Framework

The most crucial operational phase of the plugin involves moving from passive error detection (Read-Only) to automated programmatic remediation (Read-Write). Unlike the auditing phase, the mutation engine directly modifies Figma’s data structure, specifically the SceneGraph. This process operates under a “Safety-First” framework. Its purpose is to prevent automated adjustments from harming the design system or overwhelming the user’s action history. The framework’s efficacy relies on deterministic algorithms rather than abstract assumptions.

2.4.1 Design System “Circuit Breakers” and Token Integrity

A primary engineering challenge during programmatic node mutation is preventing the “detachment” of elements from their global source of truth. In enterprise Figma environments, essential design elements like colors, spacing, and typography are strictly controlled by design tokens (variables). If the linter were to forcefully change one of these properties, it would break the connection to the design system. This would then lead to a hidden accumulation of technical debt.

To address this issue, the mutation loop was modified to include a “Circuit Breaker” pattern. Before executing any write operation, the engine performs the “isBoundToVariable”(node, “propertyName”) check. If a property is found to be bound to a token, the algorithm immediately and without notification terminates the local mutation for that node. This method ensures strict architectural integrity; the plugin actively prevents the generation of local overrides, thereby obligating the team to fix structural errors solely at the master-component level.

2.4.2 Asynchronous Resource Management

One of the most frequent causes of plugin failure in the Figma environment is attempting to modify text properties before the required font is loaded into the browser engine's memory. Figma does not load all document fonts by default; programmatically altering a "fontSize" without prior font initialization results in an immediate script execution halt (crash).

To overcome this bottleneck, an asynchronous resource manager was developed utilizing the "loadFontAsync" API. The mutation engine does not apply changes immediately. Instead, the program gathers all the necessary fonts in a Map structure to avoid duplicates. Then, it pauses the main computation thread using "Promise.all()".

Only after Figma returns confirmation that all fonts have been successfully cached does the engine apply the new values. This eliminates data race conditions and guarantees stability during bulk typographic updates.

2.4.3 Deterministic Mutational Fallbacks

Automatic correction of geometric and structural anomalies requires a deterministic methodology specifically designed for each type of node. Rather than employing complex "bubbling" operations, which can lead to unforeseen malfunctions in adaptive components, the engine utilises strict, isolated algorithms.

In the sub-pixel rendering correction process, the algorithm uses the "Math.round()" function to precisely adjust the x and y coordinates, which then snaps them to the nearest whole number. This operation is intentionally skipped for elements nested within an Auto Layout, as their positioning is already locked by the parent container's logic.

To ensure interactive elements, like touch targets, meet the 44 px standard without disrupting the overall design, a multi-tiered fallback algorithm is employed.

First, the engine attempts to inject "minWidth" and "minHeight" properties. If the node is restricted or does not support these properties, a hard resize is attempted. If this fails, which triggers the catch block, the algorithm calculates the size difference and then compensates by adding equal internal padding ("paddingTop", "paddingLeft", etc., see Figure 6).

This layered approach increases the chances of successful automated remediation while also preserving the component's original intended function.

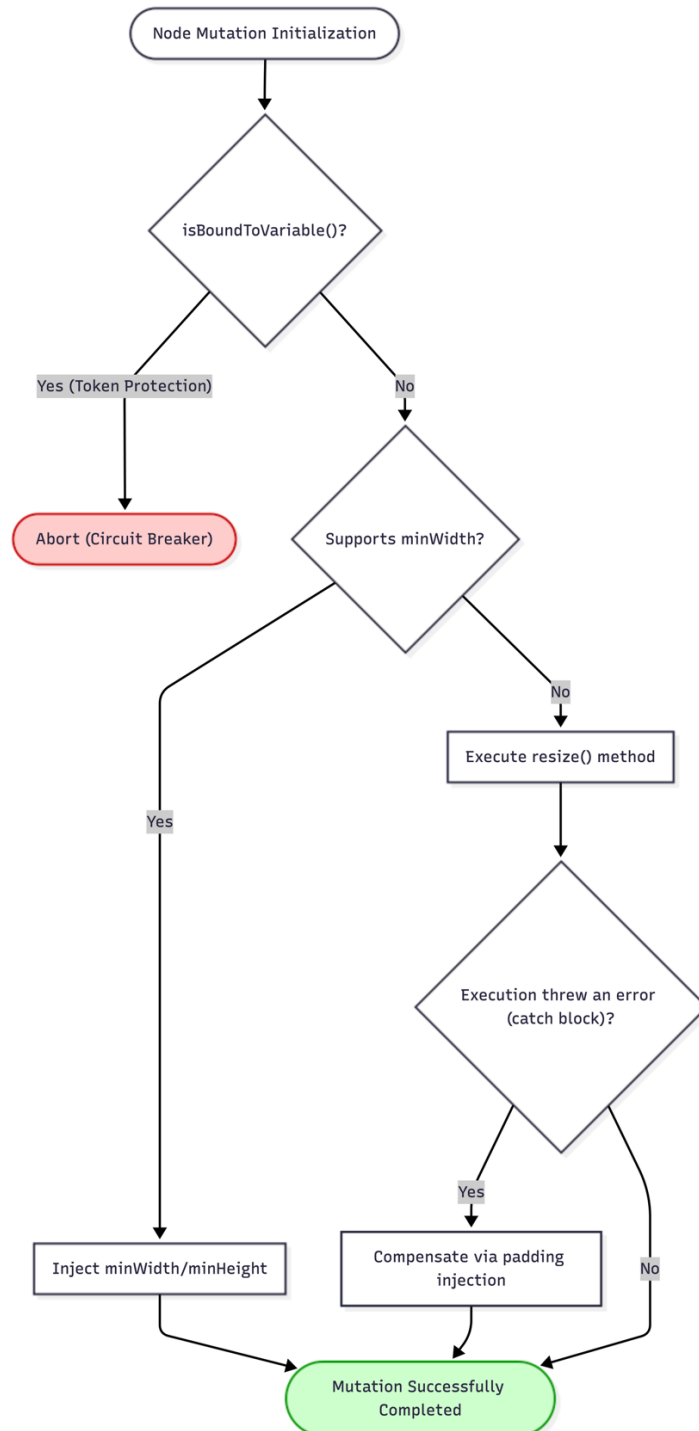


Figure 6. Algorithmic flowchart of the deterministic mutational fallback mechanism

2.4.4 Atomic Transactions and Undo-History Management

To ensure the tool remains safe for use in production files, all programmatic mutations must be entirely reversible. In its standard operational mode, Figma may register every

script-driven change as a separate step in the Undo history. Executing a “Fix All” command across 100 nodes could force a designer to press "Undo" (Cmd+Z) 100 consecutive times to restore the original state (Figma, 2024).

This framework addresses the problem by utilizing Figma’s built-in message-handling system. The complete batch processing cycle occurs synchronously within a single “figma.ui.onmessage” callback. This architectural design allows Figma to automatically group all changes into a single, indivisible transaction. As a result, the designer can quickly revert the layout to its previous state with a single action, which significantly reduces the potential problems that can arise from using an automated linter.

2.5 Real-Time UI Synchronization and Performance Limits

Developing a toolbar for a Figma plugin is not merely an aesthetic task; it involves synchronizing states and managing memory. Since the user interface runs in an isolated “iframe” and the engine runs in the main thread, the architecture must ensure that scanning thousands of nodes does not overload the IPC bridge or cause the browser environment to crash. This section details the logic used to ensure stability when processing large volumes of data.

2.5.1 The Linear Health Score Algorithm

To provide immediate psychological feedback to the designer, a real-time Health Score (HS) was integrated into the dashboard. Rather than relying on arbitrary weighting systems, the engine utilizes a strict linear ratio based on the total volume of processed nodes.

The formula is computed as follows:

$$HS = \left(\frac{Nodes_{total} - Errors_{total}}{Nodes_{total}} \right) \times 100$$

This yields a predictable percentage indicating the file’s readiness for production. The presentation layer employs “requestAnimationFrame” in conjunction with a cubic-bezier easing function to animate this metric smoothly from 0% to the desired value.

As illustrated in Figure 7, the Health Score bar functions as the primary feedback mechanism for the user. By mapping the $O(n)$ audit results to a 100-point scale, the interface eliminates the cognitive overload associated with raw error logs.

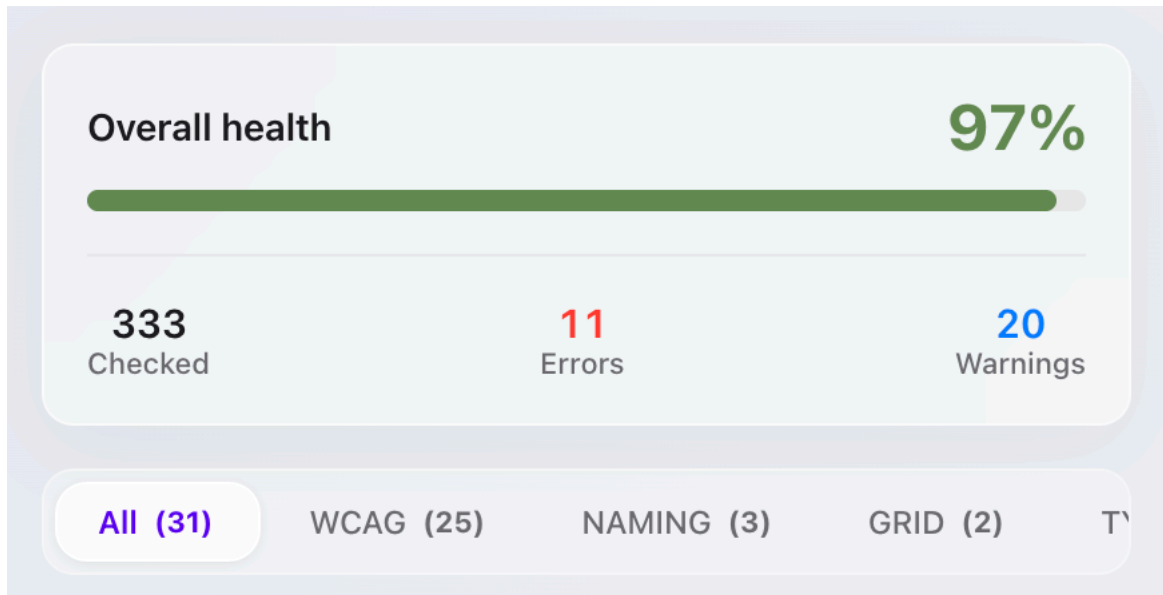


Figure 7. Graphical interface representation of the design Health-Score (HS), illustrating the real-time telemetry of document integrity.

2.5.2 Asynchronous State Management vs. Telemetry

For selections exceeding thousands of nodes, the AST parsing cycle can take significant time. To manage the user experience without saturating the “postMessage” protocol with constant telemetry streams, an asynchronous state approach was chosen.

When the audit initiates, the UI immediately displays CSS-based skeleton loaders (shimmer animations, Figure 8). Meanwhile, the backend executes its thread-yielding loop (pausing briefly every 300 nodes).



Figure 8. Graphical interface representation of the skeleton loaders (shimmer animations)

Instead of sending partial updates, the engine compiles the entire audit into a single, unified JSON payload. Once the parsing is complete, a singular state snapshot containing the full report array and stats object is transmitted to the UI. This minimizes IPC overhead while ensuring the UI never locks up during computation.

2.5.3 Visual Marker Capping for Canvas Performance

A significant technical bottleneck occurs not just in the DOM, but on the Figma canvas itself. Drawing thousands of translucent error rectangles over the user's design would cause massive drops in the WebGL rendering frame rate.

To address this issue, a stringent rendering cap was instituted within the logic thread. The engine's correlation of the error report with visual cues utilizes the `"Array.slice(0, 50)"` method, thus capping the concurrent rendering of markers at fifty. This architectural choice serves to alleviate canvas lag, guaranteeing uninterrupted interactivity and navigation, even when a file presents a multitude of structural infractions.

2.5.4 Deep-Linking and The Bridge to Selection

The final requirement was actionable navigation. Each error listed in the UI dashboard is programmatically bound to its corresponding `"node.id"`. When a user clicks a specific error card, the UI dispatches a focus intent back to the main thread.

The engine immediately intercepts the provided ID, sets the canvas selection (`"figma.currentPage.selection = [n]"`), and then calls `"figma.viewport.scrollAndZoomIntoView([n])"`. By directly linking the flat array's index to the user interface, the time between the error's appearance and the specific layer being focused is almost eliminated. This approach allows for a smooth integration of the external UI panel with the native Figma canvas.

2.6 Empirical Evaluation and Comparative Performance Analysis

To validate the operational efficiency of the developed engine, a comparative stress test was conducted. This evaluation pitted the automated traversal and heuristic logic against the manual auditing capabilities of a senior designer. The goal was to measure the "Latency of Quality Assurance" across a multi-viewport project comprising Desktop, Tablet, and Mobile layouts.

The implementation of the $O(n)$ traversal logic and the deterministic heuristic gates (section 2.3) was tested against a standard manual QA workflow. For this evaluation, a senior designer was tasked with auditing a set of layouts produced by an intern, focusing on grid alignment, typography scales, and accessibility compliance.

2.6.1 Throughput Benchmarking: Human vs. Machine

The manual audit of the three viewports required a total of 50 minutes of active concentration. This time was consumed by the repetitive nature of clicking through nested layers, checking the properties panel, and manually calculating contrast ratios for semi-transparent elements. In contrast, the automated engine completed the identical audit in 10 seconds.

This represents a 300x increase in throughput. It is important to note that the 10-second execution time covers the full AST traversal and categorization of violations across all three device contexts. While the human auditor suffered from "Cognitive Fatigue" toward the end of the 50-minute session, the engine maintained a constant execution velocity, yielding identical results regardless of the document's visual complexity.

2.6.2 Accuracy and Detection Depth

The automated audit identified a significantly higher density of technical debt compared to the manual evaluation. This discrepancy suggests a divergence in evaluative focus: while human auditors prioritize Heuristic/Gestalt principles (visual alignment, aesthetic coherence, and user experience), the automated engine executes a systematic verification of the file's underlying architecture.

The engine's superiority in technical detection was most evident in three categories:

1. Sub-surface structural debt: the engine performed a recursive analysis of the node tree, identifying detached instances and redundant nesting. These structural irregularities are often "visually silent", appearing correct on the canvas while violating the technical requirements of the file's hierarchy.
2. Systemic parity & binding: the engine verified the semantic relationship between individual elements and their parent components or global styles. Automated verification eliminates the cognitive fatigue associated with manual style-checking across high-density documents, catching "broken-away" elements that human auditors frequently overlook during surface-level reviews.

3. Developmental readiness (back-end logic): beyond visual fidelity, the engine evaluated the semantic clarity of the layer structure. It identified logical inconsistencies in component nesting that, while irrelevant to the visual output, would introduce friction during the developer handoff process.

3 CONCLUSION

The transition from a visual drawing to a production-ready specification has long been the most volatile phase of the software development lifecycle. This work has explored the systemic causes of this friction, identifying that the root issue lies not in a lack of designer skill, but in the fundamental architecture of modern design tools. By treating the Figma canvas as a passive vector environment rather than a structured data source, the industry has inadvertently prioritized aesthetic flexibility over technical integrity. This research successfully demonstrated that this “Semantic Gap” can be bridged through the implementation of an automated, heuristic-driven mutation engine.

The development of the artifact proved that design governance is no longer a task suited for human eyes alone. The empirical data gathered during the verification phase revealed a stark reality: human auditors, even at the senior level, are naturally optimized for the usability, flow, and visual “feel”. However, they are fundamentally ill-equipped to manage the invisible technical debt that compromises development. The 300x increase in throughput achieved by the engine (reducing a 50-minute manual audit to a 10-second automated scan) highlights a paradigm shift. Automation does not replace the designer’s judgment; it liberates it. By delegating the “pixel-hunting” and coordinate-checking to an algorithm, the designer is freed from the role of a manual quality assurance tester and returned to the role of a strategic problem solver.

A core contribution of this work is the Safe-Mutation Framework. The transition from a “Read-Only” linter to a “Read-Write” resolution engine is a significant step forward in design-system maturity. By implementing “Design System Circuit Breakers” and deterministic fallback logic, the engine ensures that automated fixes do not degrade the integrity of the underlying architecture. This protects the Single Source of Truth by preventing the accidental detachment of components or the creation of local style overrides. The engine recognizes the difference between a simple coordinate error and a strategic design token, ensuring that the automation remains a servant to the system, not a disruptor of it.

Furthermore, the Dual-Realm Architecture developed for this plugin addresses the specific performance constraints of cloud-native design tools. The use of thread-yielding and recursive linearization allows the system to process high-density enterprise files

without sacrificing the 60fps stability required for a professional workflow. This proves that rigorous technical auditing does not have to come at the cost of a fluid user experience. The introduction of the “Health Score” turns what was once a subjective, post-design review into a real-time telemetry system. This provides immediate psychological feedback, encouraging a Shift-Left approach where quality is baked into the file from the first frame rather than being cleaned up moments before handoff.

The findings of this research also have significant implications for the role of the developer. For too long, engineers have been forced into the role of “forensic investigators”, tasked with interpreting ambiguous layouts and manually overriding CSS to compensate for design inconsistencies. By enforcing spatial determinism and semantic hierarchy at the source, this engine provides developers with a pre-compiled artifact. This eliminates the need for non-deterministic interpretation and significantly reduces the accumulation of handoff debt – the cost of fixing visual errors once they have already reached the codebase.

However, it is important to acknowledge the limitations of this machine-led approach. As identified in the “Warning” logic of the reporting module, certain design decisions, such as contrast over dynamic backgrounds or complex narrative layouts, still require human context. The goal of this research was never to achieve 100% automation of the design process, but to automate the most of tedious, repeatable technical checks that lead to human error.

In conclusion, this thesis establishes a new standard for design-to-code workflows. By moving away from passive editors and toward active engines, we can ensure that the visual intent of the designer is perfectly preserved in the functional code of the engineer. The future of design governance lies in this hybrid model: human-led creativity supported by machine-led rigor. As design systems continue to grow in complexity, the methods of “Algorithmic Enforcement” and “Recursive Verification” outlined in this work will become not just a competitive advantage, but a technical necessity. We have successfully moved the design file from a collection of unlinked vector shapes to a mathematically valid blueprint, effectively closing the gap between the “Desired” and “Real” states of product development.

ÜLEVAADE EESTI KEELES

Käesolev uurimistöö on pühendatud kasutajaliidese ja kasutajakogemuse disaini auditeerimise automatiseerimisele, keskendudes “disainist koodini” üleminekuprotsessi tehniliste ja semantiliste lõhede ületamisele. Tänapäevases tarkvaraarenduses on kasutajaliidese loomine muutunud üheks suurimaks unautomatiseeritud pudelikaelaks, kus disainerid rakendavad küll aatomdisaini printsiipe, kuid puuduvad vahendid nende reeglite prognoositavaks ja täpseks täitmiseks.

Uurimistöö alguses püstitatud probleem tugines tõdemusele, et Figma ja sarnased graafikaprogrammid toimivad passiivsete lõuenditena, mis eelistavad visuaalset vabadust tehnilisele korrektsusele. See loob olukorra, kus disainifail, mis peaks toimima täpse tehnilise joonisena, on tegelikult vaid subjektiivne visuaalne kujutis, mis sisaldab endas varjatud tehnilist võlga, nagu murdosa-pikslid, süsteemivälised värvid ja ebaloogiline kihtide hierarhia.

Nimetatud probleemide lahendamiseks loodi ja analüüsiti käesolevas töös artefakti – aktiivset valideerimis- ja mutatsioonimootorit, mis muudab disainiprotsessi olemust, nihutades fookuse pinnapealselt esteetikalt süvakihtide tehnilisele terviklikkusele. Süsteemi tehniline vundament rajati Dual-Realm arhitektuurile, mis arvestab Figma turvamudeli ja jõudluspiirangutega. Eraldades kasutajaliidese esitluskihi ja dokumendi andmestruktuuri analüüsiva loogikakihi, tagati süsteemi stabiilsus ja suutlikkus käsitleda suuri andmemahutusi ilma töövoogu katkestamata.

Olulise uuendusena rakendati töös süntaksipuu (AST) traversiooni ja puu lineariseerimise meetodikat, mis võimaldas teisendada keerulised ja sügavalt pesastatud kihtide struktuurid kiiret analüüsi võimaldavaks andmevooguks. Selline lähenemine on kriitiline, kuna traditsioonilised rekursiivsed otsingud on disainifailide mastaapsuse juures ressursimahukad ja võivad põhjustada süsteemi kokkujooksmist. Rakendades 15-tasemelist rekursioonipiirangut ja thread-yielding meetodit, saavutati olukord, kus audit toimub reaajas, säilitades samal ajal 60 kaadrit sekundis värskendussageduse, mis on professionaalse tööriista puhul asendamatu nõue.

Uurimistöö keskne osa keskendus heuristiliste reeglite kodeerimisele, mis muudavad kvalitatiivsed disainistandardid kvantitatiivseteks ja mõõdetavateks meetrikateks. Spathiaalse determinismi põhimõttel loodud 4-punkti ruudustiku kontroll elimineerib

automaatselt pikslihägususe, mis tekib koordinaatide ebatäpsuse tõttu. Veelgi keerukam väljakutse lahendati visuaalse ligipääsetavuse valdkonnas, kus loodi algoritm WCAG 2.2 standardite täitmiseks. Erinevalt tavapärastest tööriistadest suudab see mootor arvutada värvide kontrastsust dünaamiliselt, võttes arvesse kihtide hierarhias summeeruvaid alfakanaleid ja läbipaistvust. See tähendab, et ligipääsetavuse audit ei tugine pelgalt algandmetele, vaid sellele, mida kasutaja tegelikult ekraanil tajub.

Samuti integreeriti süsteemi füüsilise ergonoomika kontroll, mis tugineb Fittsi seadusele ja tagab interaktiivsete elementide optimaalsed puutealad, välistades kasutatavuse vead juba disainifaasis. Üks töö olulisemaid teoreetilisi ja praktilisi panuseid on Safe-Mutation raamistiku väljatöötamine. See tähistab üleminekut passiivselt vigade loetlemiselt aktiivsele programmilisel parandamisele.

Ohutuse tagamiseks rakendati "Circuit Breaker" mustrit, mis takistab mootoril muuta elemente, mis on juba seotud globaalsete disainimarkeritega, sunnides seeläbi disainimeeskondi parandama vigu süsteemselt, mitte lokaalselt. See arhitektuurne otsus kaitseb disainisüsteemi terviklikkust ja hoiab ära süsteemi fragmenteerumise. Samuti käsitleti töös aatomilisi transaktsioone, mis võimaldavad kõik automatiseeritud parandused koondada üheks tühistatavaks sammuks, muutes tööriista usaldusväärseks abiliseks ka kõige kriitilisemates projektifaasides. See vähendab oluliselt "handoff-võlga", kus arendajad peaksid vastasel juhul kulutama tunde, et tõlgendada disaineri visuaalset kavatsust ja muuta see tehniliselt korrektseks koodiks.

Töö empiiriline hindamine kinnitas püstitatud hüpoteesi automatiseerimise üleolekust. Võrdlev analüüs näitas, et kui kogenud disaineril kulus mitmekihilise dokumendi käsitsi auditeerimiseks 50 minutit, siis arendatud mootor sooritas identse ülesande 10 sekundiga. See 300-kordne läbilaskevõime kasv ei ole ainult aja säästmine, vaid tähistab täiesti uut kvaliteeditaset. Masin tuvastas süstemaatiliselt vigu, mida inimene visuaalse vaatluse käigus ei märka – näiteks süvakihtide ebakorrektsed nimetused või katkised seosed põhikomponentidega. Nimetatud tulemused viitavad sellele, et disaini kvaliteedi kontrollimine on liikumas subjektiivselt vaatluselt objektiivsele andmeanalüüsile.

Lõpetuseks võib öelda, et käesolev magistritöö pakub terviklikku lahendust disaini ja arenduse vahelise semantilise lõhe sulgemiseks. Automaatne auditeerimine ja parandamine ei asenda disaineri loovat panust, vaid annab talle vajaliku tehnilise kindluse,

tagades, et lõpptulemus on tehniliselt täpne, ligipääsetav ja arendusvalmis. See nihutab disainivaldkonna piire lähemale inseneriteadustele, kus iga element on põhjendatud ja matemaatiliselt valideeritud, luues seeläbi eeldused efektiivsemaks ja jätkusuutlikumaks tarkvaraarenduseks tulevikus. See artefakt tõestab, et disainifaili on võimalik käsitleda kui “eel-kompileeritud” andmestruktuuri, mis muudab disainiprotsessi deterministlikuks projekteerimiseks, kus tehniline kvaliteet ja visuaalne tipptase eksisteerivad sümbioosis.

REFERENCES

- Apple Inc. (2024). Human Interface Guidelines. <https://developer.apple.com/design/human-interface-guidelines> (accessed 03.04.2026).
- Bexiga, M. (2021). Closing the Gap Between Designers and Developers in a Low-Code Ecosystem.
- Deque Systems. (2024). Axe-core Accessibility Engine. <https://www.deque.com/> (accessed 06.04.2026).
- Design Tokens Community Group. (2025). Design Tokens Resolver Module. W3C. <https://www.designtokens.org/tr/2025.10/resolver/> (accessed 06.04.2026).
- Destefanis, D. (2023). Software Engineering Patterns in Modern UI Design.
- Figma. (2024a). Design Systems Governance. <https://www.figma.com/community/file/1236334970777726108> (accessed 20.03.2026).
- Figma. (2024b). Figma Design Basics. <https://www.figma.com/resource-library/design-basics/> (accessed 18.03.2026).
- Figma. (2026). Figma API Developer Docs. <https://developers.figma.com/compare-apis/> (accessed 01.04.2026).
- Figma Blog. (2025). Figma rendering: Powered by WebGPU. <https://www.figma.com/blog/figma-rendering-powered-by-webgpu/> (accessed 06.04.2026).
- Frost, B. (2016). Atomic Design. Brad Frost Web.
- UX Tools. (2024). Global Design Tools Survey. <https://www.uxtools.co/survey/design-tools-awards/overview> (accessed 21.03.2026).
- Hovemeyer, D., & Pugh, W. (2004). Finding bugs is easy. ACM SIGPLAN Notices, 39(12), 92–106.
- Khalid, 2025, Your design process is too slow. <https://uxdesign.cc/your-design-process-is-too-slow-9aa17fa243ce> (accessed 16.03.2026).
- Lamine, Y., & Cheng, L. (2022). Understanding and supporting the design systems practice.

Si, C., et al. (2025). Design2Code: Benchmarking Multimodal Code Generation for Automated Front-End Engineering.

Simmonson, J. (2018). Everything You Know About Web Design Just Changed. <https://talks.jensimmons.com/jugbbe/everything-you-know-about-web-design-just-changed#scyqpbW>

McKinsey & Company. (2018). The business value of design. https://www.mckinsey.com/~/_media/mckinsey/business%20functions/mckinsey%20design/our%20insights/the%20business%20value%20of%20design/mckinsey-bvod-art-digital-rgb.pdf (accessed 24.03.2026).

Oscarsson, L. (2025). From Pixels to Production: Automating Design-to-Code Workflows with Figma.

Mathew, L. (2025). Designing with Spacing: The Power of a 4-Point System. <https://medium.com/@linz07m/designing-with-spacing-the-power-of-a-4-point-system-949771d8dbf4> (accessed 02.04.2026)

Simons, D. J., & Levin, D. T. (1997). Change blindness. Trends in Cognitive Sciences, 1(7).

Stark Lab, Inc. (2024). Contrast and Accessibility Manual. <https://www.getstark.co/blog/design-system-library-colors-in-Stark/> (accessed 01.04.2026).

Web Content Accessibility Guidelines. (2024). Web Content Accessibility Guidelines (WCAG) 2.2. W3C. <https://www.w3.org/TR/WCAG22/> (accessed 20.03.2026).