The Figma Workspace Bible

A Complete Systems Manual for AI-Assisted Rapid Design-to-Production Workflows

Version 1.0 | May 2026

Table of Contents

Part 5: Bidirectional Sync Architecture

5.1 Pipeline Overview
5.2 Five Sync Pipelines
5.3 MCP Integration
5.4 Conflict Resolution

Part 6: WordPress Plugin Architecture

6.1 Plugin Positioning
6.2 Core Capabilities
6.3 Technical Implementation
6.4 AI Foundry Integration

Part 7: Agnostic AI Skill Specification

7.1 Skill Contract Format
7.2 20 Core Skills Catalog
7.3 Implementation Adapters

Part 8: Lean/Six Sigma Process Optimization

8.1 Lean Principles for Design
8.2 Six Sigma Quality Metrics
8.3 Waste Elimination Framework
8.4 Process Improvement Cycle

Part 9: Visual Review Mechanisms

9.1 Multi-Viewport Preview
9.2 Visual Regression Testing
9.3 Accessibility Auditing
9.4 Review Workflow Integration

Part 10: Adversarial Review

10.1 Schema-First Justification
10.2 DTCG Standard Adoption
10.3 Figma-Wins-Default Rationale
10.4 MCP vs Custom Integration

Appendices

A. Glossary with Citations
B. Viewport Specification Tables

C. Print Specifications Reference
D. Next Steps Implementation Plan

Part 1: Industry-Standard Primer

1.1 Essential Design Systems Terminology

This section establishes shared vocabulary by defining every major term used in design systems work, progressing from foundational concepts to advanced implementation patterns.

Design Tokens

Definition: Platform-agnostic name-value pairs with semantic meaning that represent the atomic design decisions in a design system.

Design tokens are **not** CSS variables. As documented by the design systems community: "A design token is a platform-agnostic name-value pair with semantic meaning. CSS variables live in CSS. A design token lives in a format that can compile into CSS, Swift, Kotlin, Tailwind config, or whatever your stack needs" (DEV Community, 2024).

Key distinction: Design tokens are the **source of truth** stored in neutral formats like JSON following the W3C Design Tokens Community Group (DTCG) format. CSS variables are one **output format** generated from design tokens.

W3C DTCG Format: Published October 28, 2025, as the first stable specification (version 2025.10), the DTCG format provides a standardized JSON-based structure for token exchange. The specification defines:

- File format: JSON with `.tokens` or `.tokens.json` extensions
- Media type: `application/design-tokens+json`
- Reserved properties using `$` prefix: `$value`, `$type`, `$description`, `$extensions`
- Token types: color, dimension, font family, font weight, duration, cubic bézier, number
- Composite types: stroke style, border, transition, shadow, gradient, typography

Citation: W3C Design Tokens Community Group, "Design Tokens Format Module," https://www.designtokens.org/tr/2025.10/format/ (October 28, 2025)

Token Tiers: Primitive, Semantic, Component

The industry standard three-tier token architecture, established by Material Design 3 and adopted across major design systems, organizes tokens by abstraction level:

1. Primitive Tokens (also called "global" or "reference" tokens): Raw values with no semantic context. Examples:

- `color-blue-500: #3B82F6`
- `spacing-4: 16px`
- `font-weight-bold: 700`

Material Design 3 calls these **reference tokens**: "The most foundational layer, representing the complete palette of available style options" (Material Design, 2024).

2. Semantic Tokens (also called "system" tokens): Context-aware mappings that reference primitive tokens. Examples:

- `color-text-primary` → references `color-gray-900`
- `color-action-primary` → references `color-blue-500`
- `spacing-section-gap` → references `spacing-8`

Material Design 3 describes these as **system tokens**: "This semantic layer makes design decisions by assigning roles to the reference tokens" (Material Design, 2024).

3. Component Tokens: Specific applications to UI components. Examples:

- `button-primary-background` → references `color-action-primary`
- `card-padding` → references `spacing-section-gap`
- `input-border-radius` → references `radius-medium`

Critical principle: "Primitive tokens should only ever be referenced when mapping them to a semantic token. When codebases skip this layer and reference primitives directly, changes become painful" (Figma Help Center, 2023).

Citations:

- Material Design, "Design tokens – Material Design 3," https://m3.material.io/foundations/design-tokens/overview (2024)
- Figma Help Center, "Update 1: Tokens, variables, and styles," https://help.figma.com/hc/en-us/articles/18490793776023 (2023)

Atomic Design Methodology

Created by Brad Frost and published in his book *Atomic Design* (2016), this methodology provides a mental model for thinking about interfaces as both cohesive wholes and collections of parts.

Five stages:

1. Atoms: "UI elements that can't be broken down any further and serve as the elemental building blocks of an interface." Examples include form labels, inputs, buttons, and basic HTML elements.

2. Molecules: "Collections of atoms that form relatively simple UI components." Example: A search form molecule combines a label atom, input atom, and button atom.

3. Organisms: "Relatively complex components that form discrete sections of an interface." Example: A header organism might include a logo, navigation menu, and search form.

4. Templates: "Page-level objects that place components into a layout and articulate the design's underlying content structure." Templates focus on content structure rather than final content, following Mark Boulton's principle: "You can create good experiences without knowing the content. What you can't do is create good experiences without knowing your content structure."

5. Pages: "Specific instances of templates that show what a UI looks like with real representative content in place."

Key insight: "Atomic design is not a linear process, but rather a mental model to help us think of our user interfaces as both a cohesive whole and a collection of parts at the same time" (Frost, 2016).

Citation: Brad Frost, *Atomic Design*, Chapter 2: "Atomic Design Methodology," https://atomicdesign.bradfrost.com/chapter-2/ (2016)

Component-Driven Design

The practice of building user interfaces from modular, reusable components that maintain a single responsibility, can be composed together, and facilitate consistent experiences across products. This approach accelerated with component-based frameworks like React (2013) and Vue.js (2014), which made component reusability and composition fundamental to development workflows.

Component Variants vs Instances

Figma-specific terminology with precise meanings:

Main Component: Defines the properties and structure of a reusable component. Identified by four purple diamonds in the corner.

Variants: "Allow you to group and organize similar components into a single container. This simplifies your component library and makes it easier for everyone to find what they need" (Figma Help Center, 2023). Variants represent different states, sizes, or modes of the same conceptual component grouped within a component set.

Instances: "A copy of the component you can reuse in your designs. Instances are linked to the main component and receive any updates made to the component" (Figma Help Center, 2023).

Overrides: Modifications applied to instances, such as text content changes, color adjustments, or nested instance swaps. Figma preserves overrides when swapping between variants.

Citation: Figma Help Center, "Guide to components in Figma," https://help.figma.com/hc/en-us/articles/360038662654 (2023)

Design-to-Code vs Visual Export

Design-to-code approaches focus on extracting design intent, structure, and tokens to generate semantic code that developers can maintain. This produces component structures, design system references, and development-ready specifications.

Visual export approaches attempt to convert visual designs directly into pixel-perfect code, often producing brittle, non-semantic HTML/CSS that's difficult to maintain. These tools typically generate absolute positioning, inline styles, and non-responsive layouts.

This document advocates for schema-first design-to-code compilation over visual export. The reasoning appears in Part 10: Adversarial Review.

Design System Maturity Models

Design systems evolve through predictable stages. The four-stage model from Sparkbox/Iress (documented by UXPin, 2024) describes this journey:

Stage 1 - Build: Initial creation phase involving problem identification, technology selection, and first release with basic components and styles.

Stage 2 - Adopt: Growing usage requiring education, evangelization, and establishing contribution processes.

Stage 3 - Expand: Scaling phase adding more components, multi-platform support, and community building.

Stage 4 - Evolve: Mature system with continuous improvement, established governance, and future-facing features.

Critical insight: "It took Iress eight years to get from stage one to four. Which goes back to the point we made at the beginning—'developing a design system is a marathon, not a sprint!'" (UXPin, 2024)

Citation: UXPin, "Design System Maturity – How to Improve Your Design System," https://www.uxpin.com/studio/blog/design-system-maturity-how-to-improve-your-design-system/ (2024)

Auto-Layout, Constraints, and Responsive Design

Auto-layout (Figma-specific): A layout system that automatically resizes and repositions child elements based on padding, spacing, and alignment settings. Frames with auto-layout enabled behave like CSS flexbox containers.

Key properties:

- Direction: Horizontal, vertical, or grid
- Resizing: Fixed, Hug contents, Fill container
- Spacing and padding: Numeric values creating consistent gaps
- Alignment and distribution

Constraints (Figma-specific): Define how layers scale and reposition relative to their parent frame when the frame resizes. Applied to frames nested within regular (non-auto-layout) frames.

When to use: "Auto Layout: For components that change size based on content (buttons, cards, lists). Constraints: For components that change size based on environment (input fields, responsive layouts)" (Figma Help Center, 2023).

Citation: Figma Help Center, "Guide to auto layout," https://help.figma.com/hc/en-us/articles/360040451373-Guide-to-auto-layout (2023)

Breakpoints and Fluid Typography

Breakpoints: Viewport width thresholds where layout changes occur. Common frameworks use different breakpoint strategies:

- Bootstrap 5: 576px, 768px, 992px, 1200px, 1400px
- Tailwind CSS: 640px, 768px, 1024px, 1280px, 1536px
- Material Design: 600dp, 840dp, 1240dp, 1440dp

Fluid typography: Text sizing that scales smoothly between minimum and maximum values based on viewport width, using CSS `clamp()` or viewport units (vw, vh). This creates continuously responsive type that adapts without discrete jumps at breakpoints.

Example CSS:

```css
font-size: clamp(1rem, 0.75rem + 1vw, 2rem);
/* min: 1rem, preferred: 0.75rem + 1vw, max: 2rem */
```

CSS Methodologies: BEM, ITCSS, CUBE CSS

BEM (Block Element Modifier): Naming convention created by Yandex using double underscore for elements (`block__element`) and double dash for modifiers (`block--modifier`). Provides semantic, flat-specificity class names that prevent naming collisions.

Official syntax: https://getbem.com/naming/

ITCSS (Inverted Triangle CSS): Created by Harry Roberts, organizes CSS by specificity from generic to specific in seven layers: Settings, Tools, Generic, Elements, Objects, Components, Utilities. Each layer increases in specificity and decreases in reach.

CUBE CSS: Created by Andy Bell, organizes CSS in four layers (Composition, Utility, Block, Exception) that embrace the cascade rather than fighting it. Uses data attributes for exceptions instead of modifier classes.

Official site: https://cube.fyi/

1.2 Figma Product Ecosystem

Understanding when to use each Figma product ensures efficient workflows and appropriate tool selection for each task type.

Figma Design

Purpose: Primary tool for creating, sharing, and testing designs for websites, apps, and digital products.

When to use:

- UI/UX design work for web, mobile, desktop applications
- Design system creation and component libraries
- High-fidelity mockups and prototypes
- Interactive prototyping with Smart Animate
- Design handoff to developers via Dev Mode

Key features: Vector editing, auto-layout, components and variants, design tokens via Variables, constraints, prototyping, branching, library publishing.

Citation: Figma Help Center, "What is Figma?" https://help.figma.com/hc/en-us/articles/14563969806359-What-is-Figma (2024)

Dev Mode

Purpose: Translates designs into code-ready specifications.

When to use:

- Design-to-development handoff
- Inspecting component specifications
- Accessing code snippets and CSS values
- Reviewing Code Connect implementations
- Tracking design changes during development

Key features: Automatic spec generation, code snippets in multiple languages, design token identification, component annotations, comparison mode, plugin ecosystem.

Best practice: Designers work in Design mode; developers work in Dev Mode. This separation optimizes the interface for each role's workflow.

FigJam

Purpose: Digital whiteboard for brainstorming, meetings, diagrams, planning, and research.

When to use:

- Team collaboration sessions
- User journey mapping
- Information architecture planning
- Design sprints and workshops
- Mood board creation
- Strategy alignment
- Stakeholder workshops

Key features: Infinite canvas, stamps and stickers, voting and timer widgets, audio conversation, templates, AI-powered diagramming.

Distinction: FigJam is for **divergent thinking and exploration**. Figma Design is for **convergent execution and specification**.

Figma Slides (Beta)

Purpose: Create interactive slide presentations that stay synced with design work.

When to use:

- Design reviews with stakeholders
- Client presentations
- Team presentations where design updates should auto-sync
- Embedding live prototypes in presentations

Advantage over PowerPoint/Keynote: Slides link directly to Figma designs, so presentations automatically reflect the latest design work without manual updates.

Figma Make (Beta)

Purpose: AI-powered prompt-to-code tool for rapid prototyping.

When to use:

- Validating ideas with interactive prototypes quickly
- Exploring multiple design directions rapidly
- Generating production-ready code from designs
- Early-stage concept exploration

Note: As of May 2026, Figma Make remains in beta with evolving capabilities.

Figma Sites (Beta)

Purpose: Design and publish responsive websites directly from Figma without code.

When to use:

- Publishing portfolio sites
- Creating landing pages
- Building simple marketing sites
- Rapid website deployment from designs

Limitation: Not intended for complex web applications or sites requiring custom backend logic.

Figma Buzz (Beta)

Purpose: Create on-brand visual assets and promotional graphics at scale using AI.

When to use:

- Marketing teams producing social media assets
- Scaling brand asset creation
- Maintaining brand consistency across high-volume content
- Template-based asset generation

Target audience: Marketing and content teams rather than product designers.

1.3 The Figma Object Model

Understanding Figma's hierarchy is essential for organizing workspaces, structuring libraries, and building effective sync pipelines.

Complete Hierarchy

Organization → Team → Project → File → Page → Frame → Section → Component → Component Set → Variant → Instance → Node

Organization

Top-level entity representing a company or enterprise account. Organizations contain multiple teams, manage billing, enforce security policies, and control access at the highest level.

Use case: Large companies with multiple product lines or agencies managing multiple clients.

Team

Group of people collaborating on related projects. Teams have shared billing, members with roles (admin, editor, viewer), and contain projects.

Recommended structure for agencies: One team per major client or per internal functional area (Design System team, Client Projects team, Marketing team).

Project

Folder-like container for organizing related files. Projects live within teams and provide one level of file organization.

Recommended structure: Organize projects by product area, design system version, or initiative. Examples: "Core Design System," "Website 2026," "Mobile App v2."

File

The primary work document containing pages. Files have version history, can be published as libraries, support branching, and have individual sharing permissions.

File types (covered in detail in Part 2.5):

- Design system files
- Application/product files
- Marketing/content files
- Prototype/exploration files

Page

Canvas within a file. Pages appear in the left sidebar and provide organizational structure within files. Each file can contain unlimited pages.

Common pattern (documented in Part 2.3):

- 📐 Cover
- 🧱 Components
- 📚 Styles
- 📋 Documentation
- 🚧 WIP
- 🗄️ Archive

Citation: Figma community best practices, documented across multiple design system implementations (2020-2024)

Frame

Container object with clipping, constraints, and layout properties. Frames can nest within each other, serve as artboards/screens, and support auto-layout when enabled.

Key distinction from groups: Frames have backgrounds, clip content by default, and support auto-layout. Groups simply organize layers without layout behavior.

Section

High-level organizational tool visible in the layers panel that groups related frames on the canvas. Sections help structure large files and make components within the same section appear as "related" in the assets panel.

When to use: Organize major areas of large files, group component families, separate page regions in design files.

Component

Reusable design element with a main definition and instances. Components can have properties, variants, and nested instances. Published components become available across files via libraries.

Main component (identified by four purple diamonds): The source of truth. Changes propagate to all instances.

Component Set

Container grouping variants of a component. Component sets organize related components with different properties (size, state, theme) into a unified interface.

Example: A button component set might include variants for:

- Size: Small, Medium, Large
- State: Default, Hover, Active, Disabled
- Type: Primary, Secondary, Tertiary

Creation: Use slash naming (`Button/Primary/Default`) and Figma automatically suggests combining into a variant component set, or manually use "Combine as variants."

Variant

Individual configuration within a component set. Variants share the same base structure but differ by property values.

Properties: Custom-named axes like "Size," "State," "Theme" with enumerated values.

Instance

Copy of a component used in designs. Instances maintain a link to the main component and receive updates when the main component changes.

Overrides: Text, colors, nested instances, and properties can be overridden on instances. Figma preserves overrides when the main component updates.

Node

Generic term for any object in the Figma file hierarchy. Everything in Figma is a node: frames, shapes, text, components, instances, groups. The Plugin API exposes all elements as nodes with specific types.

Plugin API documentation: https://www.figma.com/plugin-docs/

REST API documentation: https://developers.figma.com/docs/rest-api/

Part 2: Workspace Organization Methodology

2.1 Team/Project/File Structure

For a solo operator running an AI Foundry-style agency with multiple clients and a shared design system, clear organizational structure prevents chaos and enables automation.

Recommended Team Structure

Option A: Single Team with Project-Based Organization

```
None
Team: [Agency Name] (e.g., "AI Foundry")
├── Project: Design System Core
├── Project: Client A - Website
├── Project: Client A - Mobile App
├── Project: Client B - Website
├── Project: Client B - Mobile App
├── Project: Internal - Marketing
└── Project: Exploration / R&D
```

Advantages:

- Simple billing
- All work accessible to solo operator
- Easy cross-project reference
- Lower organizational overhead

Disadvantages:

- All clients see same team in sharing UI
- Less granular access control if hiring contractors

Option B: Team Per Major Client

```
None
Organization: [Agency Name]
├── Team: AI Foundry Core
│   ├── Project: Design System
│   └── Project: Internal Tools
├── Team: Client A
│   ├── Project: Website 2026
```

```
│   ├── Project: Mobile App
│   └── Project: Marketing Assets
└── Team: Client B
    ├── Project: E-commerce Platform
    └── Project: Admin Dashboard
```

Advantages:

- Clean client separation
- Better access control for contractors
- Professional client presentation
- Clearer billing separation

Disadvantages:

- More complex to manage
- Library sharing across teams requires publishing
- Higher organizational overhead

Recommendation for solo operator: Start with Option A. Migrate to Option B when hiring contractors or when client count exceeds five.

Project Organization Principles

By product/platform:

- Keep website designs separate from mobile app designs
- Separate marketing materials from product work
- Isolate explorations from production work

By lifecycle:

- Active projects vs archived projects
- Current version vs deprecated versions

By design system tier:

- Core primitives (colors, typography, spacing)
- Base components (buttons, inputs, cards)
- Composite patterns (forms, navigation, layouts)
- Application-specific implementations

File Organization Within Projects

Design System Project Structure:

```
Design System Core/
├── 🎨 Tokens & Foundations
├── 🧱 Base Components
├── 🏗️ Composite Patterns
├── 📱 Mobile Components
├── 🖥️ Web Components
├── 📊 Data Visualization
├── 📝 Content Components
├── 🚀 Icons & Illustrations
└── 📚 Documentation Site
```

Application Project Structure:

```
Client A - Website/
├── 🎯 Project Brief & Strategy
├── 🗺️ Information Architecture
├── 🎨 Visual Design Exploration
├── 📐 Design System (local extensions)
├── 🖼️ Screens - Marketing
├── 🖼️ Screens - Product
├── 🖼️ Screens - Account
├── 🧩 Patterns Library
├── 📱 Responsive Breakpoints
└── ✅ Production Ready
```

2.2 Naming Conventions and Version Control

File Naming Convention

Pattern: `[Category] [Project/Product Name] - [Specific Area] [Version]`

Examples:

- `[DS] AI Foundry Core - Buttons v2.1`
- `[Web] HomoPlasticus - Homepage Redesign`
- `[Mobile] ClientApp - Onboarding Flow v1.0`
- `[Marketing] Q2 Campaign - Social Assets`
- `[Prototype] Feature X - Interactive Demo`

Prefixes:

- `[DS]` = Design System
- `[Web]` = Website/Web Application
- `[Mobile]` = Mobile Application (iOS/Android)
- `[Marketing]` = Marketing/Content
- `[Prototype]` = Exploration/Prototype
- `[Print]` = Print Materials
- `[Brand]` = Brand/Identity Work

Version numbering: Use semantic versioning (major.minor.patch) for design systems and released components. Omit versions for exploratory or work-in-progress files.

Component Naming Convention

Use Figma's **slash-separated naming** to create hierarchical organization in the assets panel:

Pattern: `Category/Component/Variant/State`

Examples:

- `Button/Primary/Default`
- `Button/Primary/Hover`
- `Button/Secondary/Disabled`
- `Input/Text/Empty`
- `Input/Text/Filled`
- `Input/Text/Error`
- `Card/Product/Horizontal`
- `Card/Product/Vertical`
- `Icon/Navigation/Menu`
- `Icon/Social/Facebook`

Best practices:

- Use title case for readability
- Be consistent with terminology
- Group related components under the same top-level category
- Use semantic state names (Default, Hover, Active, Disabled, Loading, Error, Success)
- Avoid abbreviations unless universally understood

Citation: Figma Help Center, "Name and organize components," https://help.figma.com/hc/en-us/articles/360038663994 (2023)

Figma Branching Strategy

Figma branching creates isolated environments for design exploration without affecting the main file. This enables safe experimentation and structured review workflows.

When to create branches:

- Feature development requiring multiple designers
- Design system updates that need review before publishing
- Experimental variations needing stakeholder evaluation
- Breaking changes that require parallel development

Branch naming:

- `feature/button-redesign`
- `experiment/new-color-palette`
- `fix/accessibility-contrast`
- `update/component-library-v3`

Workflow:

1. Designer creates branch from main file
2. Makes changes in isolated branch
3. Requests review with comparison view
4. Team reviews visual diffs
5. After approval, merge to main
6. Branch becomes part of version history

Best practices:

- Keep branches short-lived (merge within 1-2 weeks)
- One branch per meaningful unit of work
- Use comparison view to review all changes before merging
- Document merge rationale in version history
- Delete branches after merging to prevent clutter

Citation: Figma Help Center, "Guide to branching," https://help.figma.com/hc/en-us/articles/360063144053-Guide-to-branching (2023)

Library Publishing and Versioning

Publishing strategy:

- Publish only finalized, reviewed components
- Include component descriptions explaining usage
- Document breaking changes in version notes
- Notify teams before publishing major updates
- Test in sandbox environment before publishing to production

Version management:

- Use Figma's version history with descriptive names
- Name major milestones: "v2.0 - Component Restructure"
- Tag releases corresponding to code releases
- Document deprecations clearly

Version history naming pattern:

```
None
v3.0 - Major Update: New Token System
v2.5 - Added 12 New Components
v2.4.1 - Fixed Button States Accessibility
v2.4 - Updated Typography Scale
```

2.3 Page Organization Within Files

Industry practice has converged on a standard page structure that separates documentation, finished work, work-in-progress, and archived content.

Standard Page Pattern

📐 **Cover Page**:

- File title, version, last updated date
- Quick links to key pages
- Change log summary
- Component inventory/table of contents
- Usage guidelines summary

🧱 **Components Page** (for design system files):

- All published components organized by category
- Use sections to group related components
- Maintain consistent spacing and alignment
- Include annotations for complex components

📚 **Styles Page**:

- Color palette with tokens
- Typography specimens
- Spacing scale visualization
- Elevation/shadow system
- Border radius scale
- Icon grid and sizing

📋 **Documentation Page**:

- Usage guidelines
- Do's and don'ts
- Accessibility notes
- Code examples (as text or linked)
- When to use each component

🚧 **WIP (Work In Progress) Page**:

- Active explorations
- Designs under review
- Unfinished components
- Ideas being tested

🗄️ **Archive Page**:

- Deprecated components (not deleted, for reference)
- Old versions
- Rejected explorations
- Historical context

Application file page structure (websites/apps):

```
None
📐 Cover & Index
🗺️ User Flows & Architecture
🎨 Style Guide (local tokens/extensions)
🖥️ Desktop Screens
📱 Mobile Screens
⚙️ Component Library (app-specific)
📄 Content Templates
🔄 State Variations (loading, error, empty, success)
🚧 WIP
```

```
🗄️ Archive
```

Within-Page Organization

Layer naming:

- Use descriptive, semantic names that explain purpose
- Avoid generic names like "Rectangle 1" or "Group 42"
- Include state in name when showing multiple states: "Button - Hover"
- Use prefixes for layer types when helpful: `[Frame]`, `[Component]`, `[Section]`

Grouping discipline:

- Use sections for high-level organization
- Use frames for layouts and components
- Use groups only for simple temporary organization
- Avoid deep nesting (keep hierarchy under 5 levels)

Spacing and alignment:

- Maintain consistent spacing between component examples (e.g., 80px)
- Align components to a grid
- Use sections to create visual separation between component families

2.4 Component Organization Architecture

Base Components → Variants → Instances → Overrides Flow

1. Create base component: Define the simplest, most common version first. This becomes the main component.

2. Add variants: Convert to component set and define properties (Size, State, Type). Create variants for each combination.

3. Use instances: Place instances in application designs. Instances automatically stay synchronized with main component updates.

4. Apply overrides: Override instance-specific content (text, nested instances, fills) while maintaining structure and behavior.

When to create new variants vs using overrides:

- **New variant**: Structural or behavioral differences that many instances need
- **Override**: Content-specific changes unique to one or few instances

Instance swapping best practice: When a nested component has variants, users can swap to different variants through the properties panel. This enables composition patterns like buttons with different icon options.

2.5 File Type Strategy

Design System Files

Purpose: Single source of truth for shared primitives, components, and patterns.

Structure:

- Published as library
- Strictly version controlled
- Changes require review process
- Documentation integrated into file
- Comprehensive variants for all use cases

Content:

- Design tokens (via Variables)
- Base UI components
- Composite patterns
- Typography system
- Color system
- Spacing system
- Icons and illustrations

Access control: Editors limited to design system team; viewers for all other designers.

Application/Product Files

Purpose: Specific product designs consuming design system components.

Structure:

- Links to design system library
- Application-specific components (not in shared library)
- Screen designs organized by feature area
- User flows and annotations
- Responsive breakpoint variations

Relationship to design system: Imports components from library; rarely publishes back unless contributing improvements.

Marketing/Content Files

Purpose: Marketing campaigns, social media assets, presentation materials.

Structure:

- Brand assets from design system
- Campaign-specific visuals
- Social media templates
- Print materials
- Ad variations

Characteristics: Often higher iteration velocity, more visual variation, less strict adherence to component library.

Prototype/Exploration Files

Purpose: Experimentation, concept testing, R&D.

Structure:

- Loose, flexible organization
- Multiple directions explored simultaneously
- Interactive prototypes for user testing
- Concept validation before committing to production

Lifecycle: Most explorations archived or deleted after decision made. Successful explorations migrate to application files or contribute to design system.

2.6 Industry Reference Analysis

Learning from established design systems reveals patterns worth adopting and common pitfalls to avoid.

Shopify Polaris

What it does well: Most complete public design system with exceptional content guidelines.

Borrowable patterns:

- Integrated content guidelines alongside component specs

- Voice, tone, grammar, error message guidance
- Well-documented token system with clear semantic naming
- Over 90 components with comprehensive variant coverage
- Pattern documentation (not just components)

Token structure: Primitive → Semantic → Component layers clearly demonstrated.

Citation: Shopify Polaris, "Design," https://polaris-react.shopify.com/design (2024)

IBM Carbon Design System

What it does well: Framework-agnostic and accessibility-first approach.

Borrowable patterns:

- Support for React, Angular, Vue, Svelte, Web Components
- Strong accessibility documentation at component level
- Open-source with active community contribution
- Clear contribution model

Key insight: Build design system outputs for multiple frameworks rather than coupling to single technology.

Citation: Backlight.dev, "Best design system examples," https://backlight.dev/mastery/best-design-system-examples (2024)

Atlassian Design System

What it does well: End-to-end design language with comprehensive resources.

Borrowable patterns:

- Extensive Figma libraries
- Logo and illustration libraries
- Pattern documentation beyond components
- Clear guidelines for when to use each component

Key insight: Design systems extend beyond UI components to include illustrations, icons, logos, and content patterns.

Material Design 3

What it does well: Comprehensive token architecture and dynamic theming.

Borrowable patterns:

- Three-tier token architecture (reference → system → component)
- Semantic token remapping for light/dark/dynamic themes
- Comprehensive documentation of design principles
- Clear responsive breakpoint system

Citation: Material Design, "Design tokens – Material Design 3," https://m3.material.io/foundations/design-tokens/overview (2024)

Apple Human Interface Guidelines

What it does well: Platform-specific guidance within unified system.

Borrowable patterns:

- Core principles (clarity, deference, depth)
- Platform-specific specifications while maintaining consistency
- Official Figma UI kits for all platforms
- Integration of accessibility guidance throughout (not separate section)

Citation: Apple Developer, "Human Interface Guidelines," https://developer.apple.com/design/human-interface-guidelines/ (2024)

GOV.UK Design System

What it does well: Governance model and contribution process.

Borrowable patterns:

- Open contribution model with working group approval
- Strong governance with dedicated maintenance team
- WCAG 2.0 AA as non-negotiable baseline
- Emphasis on learning from other teams' research
- Clear documentation of when to use each pattern
- Principle: "Be consistent, not uniform" (allows appropriate variation)

Key insight: "The Design System brings together patterns and code found across government. It provides styles, components and patterns to help teams create user-centred digital services" (GOV.UK, 2018).

Citation: Government Digital Service, "Introducing the GOV.UK Design System," https://gds.blog.gov.uk/2018/06/22/introducing-the-gov-uk-design-system/ (2018)

Part 3: Multi-Viewport, Multi-Platform, Multi-Language Methodology

3.1 Comprehensive Viewport Matrix

Complete specifications appear in Appendix B. This section establishes methodology for working across all target platforms.

Mobile Viewport Strategy

iPhone Specifications (2026 current lineup):

- iPhone SE: 375×667 CSS pixels, 2x DPR, 326 PPI
- iPhone 14/15/16: 393×852 CSS pixels, 3x DPR, 460 PPI
- iPhone Pro models: 59pt top safe area (Dynamic Island), 34pt bottom (home indicator)

Key considerations:

- Design in CSS pixels (pt), not physical pixels
- Account for safe areas (notch, Dynamic Island, home indicator)
- Provide @2x and @3x assets
- Test portrait and landscape orientations

Android Device Matrix: Material Design window size classes provide device-agnostic breakpoints:

- Compact (<600dp): Single-column layouts
- Medium (600-839dp): Single or two-column layouts
- Expanded (≥840dp): Multi-column, master-detail patterns

Citation: Material Design, "Applying layout," https://m3.material.io/foundations/layout/applying-layout (2024)

Tablet Viewport Strategy

iPad Lineup:

- iPad mini (6th gen): 744×1133 CSS pixels
- iPad (10th gen): 810×1080 CSS pixels
- iPad Air/Pro 11": 834×1194 CSS pixels
- iPad Air/Pro 13": 1024×1366 CSS pixels

Design considerations:

- 4:3 aspect ratio (approximately)
- No notch (design to edge of safe areas)
- 24pt status bar height
- Landscape orientation common (design for both)

Android Tablets: Follow Material Design breakpoints:

- Small tablets (7-8"): 600-839dp width
- Standard tablets (10"): 840-1239dp width
- Large tablets (12"+): ≥1240dp width

Desktop/Web Breakpoint Strategy

Industry-standard breakpoints (synthesized from Bootstrap, Tailwind, Material Design):

Breakpoint	Min Width	Typical Device	Container Width
Mobile	<640px	Phones	100%
Small	640px	Large phones	640px
Medium	768px	Tablets	768px
Large	1024px	Laptops	1024px
X-Large	1280px	Desktops	1280px
2X-Large	1536px	Large displays	1440px

Recommended design widths:

- Mobile: 375px (iPhone), 393px (modern Android)
- Tablet: 768px, 834px, 1024px
- Desktop: 1280px, 1440px, 1920px

Fluid strategy: Design mobile-first with flexible grids. Set max-width containers (typically 1280-1440px) to prevent excessive line lengths on ultrawide monitors.

Citations:

- Bootstrap, "Breakpoints," https://getbootstrap.com/docs/5.3/layout/breakpoints/ (2024)

- Tailwind CSS, "Responsive Design," https://tailwindcss.com/docs/responsive-design (2024)

Print Specifications Summary

Business Cards:

- US: 3.5"×2" (trim), add 0.125" bleed
- EU: 85mm×55mm (trim), add 3mm bleed
- Safe area: 0.125" inside trim

Posters:

- 18"×24", 24"×36" (common US sizes)
- A3, A2, A1, A0 (ISO sizes)
- Bleed: 0.125" (small format) to 0.5" (large format)

Billboards:

- Bulletin: 14'×48' (design at ½"=1' scale @ 300 DPI = 12.5 PPI actual)
- Vinyl includes 7" per edge (3" bleed + 4" pockets)

Digital Ad Units (IAB Standards):

- Medium Rectangle: 300×250px
- Leaderboard: 728×90px
- Billboard: 970×250px
- Half Page: 300×600px
- Mobile Banner: 320×50px

Complete specifications in Appendix B.

Citations:

- IAB, "New Ad Portfolio," https://www.iab.com/newadportfolio/ (2017)
- IDEAlliance, "GRACoL Standards," https://www.idealliance.org/gracol/ (2024)

Social Media Specifications (2026 Current)

Instagram:

- Feed (Portrait): 1080×1350 (4:5)
- Feed (Square): 1080×1080 (1:1)
- Stories/Reels: 1080×1920 (9:16)

- Safe zone: 14% from top/bottom edges

Facebook:

- Feed: 1200×1200 (square), 1200×630 (landscape)
- Stories: 1080×1920
- Cover: 851×315 (desktop), 640×360 (mobile)

LinkedIn:

- Feed: 1200×1200 (square preferred)
- Link Preview: 1200×627
- Cover (personal): 1584×396

TikTok/Reels/Shorts:

- Video: 1080×1920 (9:16)
- Safe zone: 120px from top, 250px from bottom

X (Twitter):

- In-stream Photo: 1200×675 (16:9)
- Header: 1500×500 (3:1)

Complete specifications in Appendix B.

3.2 Adaptive Component Architecture

Building components that respond gracefully to viewport changes requires methodical application of Figma's layout features and deliberate variant strategies.

Responsive vs Adaptive vs Fluid Components

Responsive components: Single component that adapts continuously using auto-layout, constraints, and relative sizing. Uses Figma's Hug, Fill, and Min/Max width properties.

Adaptive components: Multiple variants for specific breakpoints (Mobile, Tablet, Desktop). Component structure may differ significantly between variants.

Fluid components: Continuously scale using percentage-based or viewport-relative sizing without discrete breakpoints.

When to use each:

- **Responsive**: Buttons, cards, simple layouts where structure remains consistent
- **Adaptive**: Navigation, complex layouts where structure changes significantly
- **Fluid**: Typography, spacing, simple containers

Auto-Layout Methodology for Responsive Design

Core technique: Build components using nested auto-layout frames that resize intelligently.

Step-by-step methodology:

1. Start with content: Place text and icons first. Apply auto-layout to wrap them with padding.

2. Set resizing behavior:

- Text: Fixed or Fill container (with max-width for readability)
- Icons: Fixed
- Containers: Fill parent or Hug contents
- Spacers: Fill to create flexible gaps

3. Use Min/Max constraints: Set minimum and maximum widths to prevent extremes.
Example: Button with min-width: 120px, max-width: 400px

4. Nest auto-layouts: Horizontal auto-layout inside vertical auto-layout creates complex responsive behavior.

5. Test at multiple widths: Drag frame edges to verify behavior at different sizes.

Example: Responsive Card Component:

```
None
Card [Frame, Auto-layout vertical, Hug contents]
├─ Image [Fixed 16:9 aspect, Fill width]
├─ Content [Auto-layout vertical, Padding 24px, Gap 16px, Hug]
|  ├─ Title [Fill width, Fixed height or Hug]
|  ├─ Description [Fill width, Fixed height or Hug]
|  └─ Actions [Auto-layout horizontal, Gap 12px, Hug]
|     ├─ Button Primary
|     └─ Button Secondary
```

Key properties:

- Card: Padding 0 (image bleeds), Hug contents vertically
- Image: Fill horizontal, maintains aspect ratio

- Content: Padding 24px, Gap 16px between children
- Actions: Horizontal alignment, Hug contents

This card now:

- Adapts to any width (Fill horizontal behavior)
- Maintains proportions (aspect ratio locked)
- Spaces content consistently (gap tokens)
- Stacks or flows buttons based on available width

Variant Strategy for Breakpoint-Specific Behavior

When structure changes significantly across breakpoints, create variants rather than trying to make a single component flex.

Example: Navigation Component:

Mobile variant (≤768px):

```
None
Navigation/Mobile
├─ Logo [Fixed]
├─ Spacer [Fill]
└─ Menu Button [Fixed]
```

Desktop variant (≥769px):

```
None
Navigation/Desktop
├─ Logo [Fixed]
├─ Nav Links [Auto-layout horizontal, Fill]
│  ├─ Link 1
│  ├─ Link 2
│  ├─ Link 3
│  └─ Link 4
├─ Spacer [Fill]
└─ User Menu [Fixed]
```

Implementation: Use boolean property or breakpoint property in variant.

- Property: `Breakpoint` with values `Mobile`, `Tablet`, `Desktop`
- Instance swapping in designs based on frame width

Device Frame Pattern

Technique: Create frame templates representing specific devices. Nest adaptive components inside these frames for realistic preview.

Device frame structure:

```
None
iPhone 15 Pro Frame [375×812 + safe areas]
├─ Status Bar [Frame, 59pt tall, simulates Dynamic Island]
├─ Content Area [Fill vertical, Fill horizontal]
│  └─ [Place designs here]
└─ Home Indicator [34pt tall, bottom]
```

Benefit: Designers see exactly how components appear with device chrome (status bars, notches, home indicators) without manually adding/removing these elements.

Library approach: Publish device frame components. Designers use instances and swap in their content.

Scalar Testing for Layout Response

Methodology: Systematically test components at different widths to verify responsive behavior.

Test widths:

- 320px (minimum mobile)
- 375px (iPhone SE, small mobile)
- 393px (modern Android)
- 768px (tablet portrait)
- 1024px (tablet landscape, small desktop)
- 1280px (standard desktop)
- 1440px (large desktop)
- 1920px (full HD)

Process:

1. Place component instance on canvas
2. Resize instance to each test width
3. Verify: content doesn't overflow, spacing remains appropriate, text remains readable
4. Document any breakpoints where layout should change
5. Create variants if needed

3.3 Multi-Language Support Methodology

Designing for multiple languages (English, Spanish, German, Arabic, Hebrew, Chinese, Japanese) requires planning for text expansion, directionality changes, and glyph variations.

Figma Variables for String Management

Figma Variables (introduced June 2023) enable string substitution for multi-language design without duplicating frames.

Implementation:

1. Create string variables:

```
None
Collection: UI Strings
├─ Mode: English
│  ├─ nav.home = "Home"
│  ├─ nav.about = "About"
│  ├─ button.submit = "Submit"
│  └─ button.cancel = "Cancel"
├─ Mode: Spanish
│  ├─ nav.home = "Inicio"
│  ├─ nav.about = "Acerca de"
│  ├─ button.submit = "Enviar"
│  └─ button.cancel = "Cancelar"
└─ Mode: German
   ├─ nav.home = "Startseite"
   ├─ nav.about = "Über"
   ├─ button.submit = "Absenden"
   └─ button.cancel = "Abbrechen"
```

2. Apply variables to text: Bind text layers to variables. When mode changes, all text updates instantly.

3. Toggle modes: Switch modes in right panel to preview all languages without duplicating frames.

Benefit: Design once, test all languages. Changes to layout automatically apply to all language versions.

Citation: Figma Help Center, "Update 1: Tokens, variables, and styles," https://help.figma.com/hc/en-us/articles/18490793776023 (2023)

Text Expansion Considerations

Different languages require different amounts of space for the same content.

Text expansion factors (Microsoft i18n guidelines):

- **German**: 30% longer than English on average
- **French**: 15-20% longer than English
- **Spanish**: 20-25% longer than English
- **Italian**: 25-30% longer than English
- **Portuguese**: 20-25% longer than English

Asian languages (Chinese, Japanese, Korean):

- Often **shorter in character count** but require larger font sizes for readability
- Minimum 16px for body text (vs 14px for Latin scripts)
- Line height should be 1.5× or greater (tighter leading causes character overlap)

Design implications:

- Test layouts with longest-language content (often German)
- Use flexible layouts (Fill, Hug) rather than fixed widths
- Set max-width limits to prevent awkward line lengths
- Minimum button width should accommodate expanded text
- Avoid all-caps (not appropriate in all languages)

Pseudo-localization technique: Replace English text with expanded placeholder text to stress-test layouts. Example: "Submit" → "Süßmït" → "[!!! Süßmït !!!]" (simulates 30% expansion + indicates untranslated strings)

Citations:

- Microsoft, "Internationalization best practices," developer.microsoft.com
- W3C, "Internationalization techniques," https://www.w3.org/International/ (2024)

RTL (Right-to-Left) Layout Considerations

Arabic, Hebrew, and Persian use RTL text direction, requiring mirrored layouts.

What mirrors:

- Overall layout direction (left sidebar becomes right sidebar)
- Reading order (tabs progress right-to-left)

- Icon direction for directional elements (back/forward arrows flip)
- Text alignment (left-aligned becomes right-aligned)

What doesn't mirror:

- Media controls (play buttons remain pointing right globally)
- Clocks and circular progress (clockwise universal)
- Numbers (Arabic numerals 1-2-3 read left-to-right even in RTL context)
- Non-directional icons (search, settings, user remain unchanged)
- Logos and brand elements

Figma RTL workflow:

1. Design LTR version first
2. Duplicate frame for RTL variant
3. Reverse auto-layout direction (Left → Right becomes Right → Left)
4. Flip directional icons manually
5. Change text alignment (left → right)
6. Test with actual RTL text (not just reversed English)

Auto-layout direction inheritance: Parent auto-layout direction affects nested children, making bulk RTL conversion faster.

Best practice: Create RTL variants as part of design system component sets. Example: `Navigation/Desktop/LTR` and `Navigation/Desktop/RTL`

CJK (Chinese, Japanese, Korean) Considerations

Typography specifics:

- Larger font sizes needed (minimum 16px vs 14px for Latin)
- Increased line height (1.5-1.7× vs 1.4× for Latin)
- Full-width punctuation takes more horizontal space
- Vertical text direction option (traditional for Japanese, Chinese)

Glyph width: CJK characters are square-ish, affecting layout density.

Font fallback stack:

```
CSS
font-family: -apple-system, BlinkMacSystemFont,
             "Segoe UI", "Helvetica Neue",
             "PingFang SC", "Hiragino Sans", "Noto Sans CJK JP",
```

```
            sans-serif;
```

Web font considerations:

- CJK fonts are large (5-15MB vs <200KB for Latin)
- Subset fonts or use system fonts
- Variable fonts emerging for CJK (reduces file size)

Figma workflow: Use Variables with modes to switch between Latin, Chinese, Japanese, Korean test content. Verify:

- Characters don't clip
- Line breaks occur at appropriate places
- Mixed Latin/CJK content aligns properly

3.4 Animation and Dynamic Content Prototyping

Figma Smart Animate

Smart Animate automatically creates transitions between frames by matching layer names and interpolating property changes.

Supported transitions:

- Position (X/Y coordinates)
- Size (width/height)
- Rotation
- Opacity
- Color (fills, strokes)
- Corner radius
- Effects (blurs, shadows)

Setup:

1. Create Frame A with initial state
2. Duplicate to Frame B
3. Ensure matching layer names (critical!)
4. Modify properties in Frame B
5. Connect with prototype arrow
6. Set interaction: Smart Animate
7. Adjust duration (100-800ms) and easing

Easing options:

- Ease in: Slow start, fast finish (entering animations)
- Ease out: Fast start, slow finish (exiting animations)
- Ease in/out: Slow start and finish (most natural)
- Linear: Constant speed (use sparingly)

Best practices:

- 100-300ms: Micro-interactions (button hover, tooltip appear)
- 300-500ms: Screen transitions, modal open/close
- >500ms: Large-scale animations, perceived as slow
- Match layer names exactly (case-sensitive)
- Avoid animating too many properties simultaneously

Citation: Figma Help Center, "Prototype animations," https://help.figma.com/hc/en-us/articles/360040522373-Prototype-animations (2023)

Advanced Prototyping: Variables-Driven Animation

Figma's Variables can drive prototype interactions beyond static frame-to-frame transitions.

Use case: Toggle switch with state variable:

1. Create boolean variable: `isEnabled`
2. Build component with two states bound to variable
3. Set prototype action: On click → Set variable `isEnabled` to `NOT isEnabled`
4. Component visually updates based on variable value

Use case: Multi-step form with progress:

1. Create number variable: `currentStep`
2. Build conditional logic showing/hiding form sections based on step
3. Button actions increment/decrement step variable

Benefit: Simulate app logic without writing code. Test complex flows with branching, state persistence, and conditional visibility.

When to Use Figma Prototypes vs External Tools

Use Figma prototyping when:

- Communicating interaction intent to developers

- Testing user flows with stakeholders
- Demonstrating component state changes
- Creating clickable demos for presentations
- Prototyping simple animations and transitions

Use external tools (Framer, Principle, ProtoPie, Origami) when:

- Need code-level control over interactions
- Require physics-based animations
- Testing complex gestures (multi-touch, drag, swipe)
- Need to connect to real APIs/data
- Building prototypes for user research requiring high fidelity

Lottie Integration for Production Animations

Lottie (created by Airbnb) is a JSON-based animation format that renders After Effects animations in real-time across web and mobile platforms.

Advantages:

- 600% lighter than GIFs
- Vector-based (scales infinitely)
- Interactive and controllable
- Cross-platform (web, iOS, Android, React Native)

Workflow: Figma → Lottie:

1. Design animation frames in Figma
2. Export via LottieFiles Figma plugin
3. Or: Export frames, recreate in After Effects, export via Bodymovin
4. Optimize JSON file (reduce keyframes, simplify paths)
5. Implement with lottie-web, lottie-react, etc.

LottieFiles Figma Plugin: Converts Figma animations directly to Lottie JSON format without After Effects.

Use cases:

- Loading spinners
- Success/error confirmations
- Onboarding illustrations
- Icon micro-interactions
- Empty state illustrations

Implementation:

```HTML
<script
src="https://unpkg.com/@lottiefiles/lottie-player@latest/dist/lottie-player.js"></
script>
<lottie-player src="animation.json" background="transparent" speed="1" loop
autoplay></lottie-player>
```

Control options: Play, pause, stop, seek, reverse, control speed, trigger on scroll/hover/click.

Citation: LottieFiles, "Platform," https://lottiefiles.com/platform (2024)

Part 4: The Design Control Schema (DCS)

4.1 Purpose and Design Principles

The Design Control Schema serves as the **neutral intermediate representation** between Figma design files and all deployment targets (WordPress, headless CMSs, static sites, mobile apps). This schema-first approach enables a compiler architecture rather than a brittle visual export system.

Why Schema-First Beats Visual Export

Visual export tools (Figma-to-HTML converters) attempt to replicate pixel-perfect layouts by generating absolute positioning, inline styles, and non-semantic markup. This produces code that:

- Breaks when viewport changes
- Cannot be maintained by developers
- Ignores semantic HTML
- Duplicates styling instead of referencing design tokens
- Fails to map to component architectures

Schema-first compilation extracts design **intent** and **structure**, then generates appropriate implementations for each target:

- Gutenberg blocks with semantic markup
- React/Vue components referencing design tokens
- Headless CMS schemas with field validation

- Mobile native components (SwiftUI, Jetpack Compose)

The DCS captures:

- Component identity and namespace
- Token references (not hardcoded values)
- Variant structures
- Content slot definitions
- Responsive rules
- Accessibility requirements
- CMS field mappings

The DCS does NOT capture:

- Absolute pixel positions
- Inline style values
- Frame coordinates
- Visual appearance details not expressible through tokens

W3C DTCG Alignment

The Design Control Schema aligns with the W3C Design Tokens Community Group (DTCG) format specification (version 2025.10, published October 28, 2025).

DTCG compliance means:

- Token values follow `$value`, `$type`, `$description` structure
- Token types match DTCG taxonomy (color, dimension, fontFamily, etc.)
- Composite tokens use DTCG patterns (border, shadow, typography)
- Files can be processed by any DTCG-compliant tool (Style Dictionary 4+, Tokens Studio, Terrazzo)

Why DTCG matters:

- **Interoperability**: Tokens work across tools (Figma, Sketch, Framer, code)
- **Future-proofing**: Open standard, not proprietary format
- **Tooling ecosystem**: Growing support from major players
- **Industry adoption**: Adobe, Amazon, Google, Microsoft, Meta, Salesforce, Shopify, Figma all involved

Citation: W3C Design Tokens Community Group, "Design Tokens Format Module," https://www.designtokens.org/tr/2025.10/format/ (October 2025)

JSON Schema as the Schema Language

Why JSON Schema over GraphQL SDL:

JSON Schema advantages:

- Language-agnostic (not tied to GraphQL)
- More comprehensive validation rules
- Works with REST APIs, JSON config files, any JSON structure
- Extensive tooling for validation, generation, documentation
- Better for pure data modeling vs API operations

GraphQL SDL purpose: Define GraphQL API schemas with queries, mutations, subscriptions. Excellent for APIs, but overkill for component metadata.

JSON Schema specification: https://json-schema.org/specification (2020-12 stable version)

4.2 Schema Sections Specification

The complete DCS for a single component includes these sections:

Identity Section

```JSON
{
  "identity": {
    "name": "Button",
    "namespace": "ai-foundry",
    "version": "2.1.0",
    "deprecated": false,
    "deprecationMessage": null,
    "replacedBy": null,
    "figmaNodeId": "123:456",
    "figmaFileKey": "abc123xyz",
    "governanceOwner": "design-system-team",
    "lastModified": "2026-05-17T10:30:00Z"
  }
}
```

Fields:

- `name`: Component name (human-readable)
- `namespace`: Organizational prefix preventing collisions
- `version`: Semantic version (major.minor.patch)

- `deprecated`: Boolean flag
- `deprecationMessage`: User-facing explanation if deprecated
- `replacedBy`: Reference to replacement component
- `figmaNodeId`: Original Figma node ID for syncing
- `figmaFileKey`: Figma file containing source
- `governanceOwner`: Team/person responsible for component
- `lastModified`: ISO 8601 timestamp

Token References Section

```json
{
  "tokens": {
    "color": {
      "background": "$color.action.primary",
      "backgroundHover": "$color.action.primary-hover",
      "text": "$color.text.on-primary",
      "border": "$color.border.primary"
    },
    "typography": {
      "fontFamily": "$font.family.sans",
      "fontSize": "$font.size.md",
      "fontWeight": "$font.weight.semibold",
      "lineHeight": "$font.lineHeight.tight"
    },
    "spacing": {
      "paddingBlock": "$spacing.3",
      "paddingInline": "$spacing.5",
      "gap": "$spacing.2"
    },
    "border": {
      "radius": "$radius.md",
      "width": "$border.width.default"
    },
    "shadow": {
      "default": "$shadow.sm",
      "hover": "$shadow.md"
    },
    "motion": {
      "duration": "$motion.duration.fast",
      "easing": "$motion.easing.ease-out"
    }
  }
}
```

All values reference design tokens (format: `$category.semantic-name`). Never hardcode values like `#3B82F6` or `16px`.

Token categories: Align with DTCG types and Material Design 3 semantic token structure.

Variants Section

JSON

```
{
  "variants": {
    "properties": {
      "size": {
        "type": "enum",
        "values": ["sm", "md", "lg"],
        "default": "md"
      },
      "variant": {
        "type": "enum",
        "values": ["primary", "secondary", "tertiary", "ghost"],
        "default": "primary"
      },
      "state": {
        "type": "enum",
        "values": ["default", "hover", "active", "disabled", "loading"],
        "default": "default"
      },
      "fullWidth": {
        "type": "boolean",
        "default": false
      }
    },
    "combinations": {
      "primary-md-default": { "tokens": { "color": {...}} },
      "primary-md-hover": { "tokens": { "color": {...}} }
    }
  }
}
```

Properties: Axes of variation (Size, Variant, State, Boolean flags)

Combinations: Specific token overrides for each variant. Can be auto-generated from Figma component set properties.

Slots Section

JSON

```
{
  "slots": {
```

```
    "icon": {
      "type": "component",
      "allowedTypes": ["Icon"],
      "required": false,
      "description": "Optional leading icon"
    },
    "label": {
      "type": "text",
      "required": true,
      "maxLength": 100,
      "description": "Button label text"
    },
    "children": {
      "type": "InnerBlocks",
      "allowedBlocks": ["core/paragraph", "core/image"],
      "maxCount": null,
      "description": "Nested content (WordPress-specific)"
    }
  }
}
```

Slot types:

- `text`: Plain text content
- `richText`: Formatted text (bold, italic, links)
- `image`: Media with alt text
- `icon`: Icon reference
- `component`: Nested component (specify allowed types)
- `InnerBlocks`: WordPress block composition pattern

Validation: Required flag, allowed types, length limits, count constraints.

Properties Section

JSON

```
{
  "properties": {
    "href": {
      "type": "string",
      "format": "uri",
      "required": false,
      "locked": false,
      "localized": false,
      "description": "Link URL if button is a link"
```

```
    },
    "onClick": {
      "type": "function",
      "required": false,
      "description": "Click handler for button"
    },
    "ariaLabel": {
      "type": "string",
      "required": false,
      "locked": false,
      "localized": true,
      "description": "Accessible label overriding visible text"
    }
  }
}
```

Property types: string, number, boolean, array, object, function, enum

Validation: JSON Schema validation rules (format, pattern, min/max, etc.)

Localization flag: Indicates translatable content

Locks Section

JSON

```
{
  "locks": {
    "structure": "designer",
    "tokens": "designer",
    "content": "editor",
    "properties": {
      "href": "editor",
      "onClick": "developer",
      "ariaLabel": "editor"
    }
  }
}
```

Lock levels:

- `designer`: Only design system team can change
- `editor`: Content editors can modify in CMS
- `developer`: Requires code changes
- `both`: Both designer and editor can change (last write wins or three-way merge)

This implements the "Figma wins by default" governance with per-field overrides.

Responsive Rules Section

```json
{
  "responsive": {
    "breakpoints": {
      "mobile": {
        "maxWidth": "768px",
        "tokens": {
          "spacing": {
            "paddingInline": "$spacing.3"
          },
          "typography": {
            "fontSize": "$font.size.sm"
          }
        },
        "properties": {
          "fullWidth": true
        }
      },
      "desktop": {
        "minWidth": "769px",
        "tokens": {
          "spacing": {
            "paddingInline": "$spacing.5"
          }
        }
      }
    }
  }
}
```

Breakpoint-specific overrides for tokens and properties. Allows mobile-specific sizing, spacing adjustments, property changes.

Accessibility Section

```json
{
  "accessibility": {
    "role": "button",
    "requiredAttributes": ["aria-label"],
    "contrastPairs": [
```

```
      {"foreground": "$color.text.on-primary", "background":
"$color.action.primary", "minimum": "4.5:1"}
    ],
    "focusIndicator": {
      "required": true,
      "color": "$color.focus.ring",
      "width": "2px",
      "offset": "2px"
    },
    "keyboardInteraction": {
      "activationKeys": ["Enter", "Space"],
      "description": "Activates button action"
    },
    "screenReaderGuidance": "Button: {label}. {state}."
  }
}
```

WCAG requirements: Role, required ARIA attributes, contrast minimums, focus indicators, keyboard support.

Contrast pairs: List foreground/background token combinations with minimum ratios (4.5:1 for AA, 7:1 for AAA).

Motion Section

JSON

```
{
  "motion": {
    "preset": "fade-in",
    "duration": "$motion.duration.normal",
    "easing": "$motion.easing.ease-out",
    "trigger": "onload",
    "reducedMotion": {
      "enabled": true,
      "fallback": "none"
    }
  }
}
```

Animation metadata: Preset name, timing references, trigger events, reduced-motion fallback (respects `prefers-reduced-motion` media query).

CMS Hints Section

JSON

```
{
  "cmsHints": {
    "wordpress": {
      "blockName": "ai-foundry/button",
      "category": "design",
      "supports": {
        "align": ["left", "center", "right", "wide", "full"],
        "color": {"background": true, "text": true},
        "spacing": {"margin": true, "padding": true}
      },
      "attributes": {
        "label": {"type": "string", "source": "text", "selector":
".wp-block-button__link"},
        "href": {"type": "string", "source": "attribute", "selector": "a",
"attribute": "href"}
      }
    },
    "sanity": {
      "type": "object",
      "name": "button",
      "fields": [
        {"name": "label", "type": "string", "validation": "Rule =>
Rule.required()"},
        {"name": "href", "type": "url"}
      ]
    },
    "contentful": {
      "contentType": "button",
      "fields": [
        {"id": "label", "type": "Symbol", "required": true},
        {"id": "href", "type": "Symbol", "validations": [{"regexp": {"pattern":
"^https?://"}}]}
      ]
    }
  }
}
```

Per-CMS mappings: WordPress block.json structure, Sanity schema, Contentful content model, etc.

Renderer Hints Section

```json
{
  "rendererHints": {
    "react": {
      "import": "import { Button } from '@ai-foundry/components'",
      "usage": "<Button size=\"{size}\" variant=\"{variant}\">{label}</Button>",
      "props": {
        "size": "enum",
        "variant": "enum",
        "onClick": "function"
      }
    },
    "vue": {
      "import": "import Button from '@ai-foundry/components/Button.vue'",
      "usage": "<Button :size=\"size\" :variant=\"variant\">{{ label }}</Button>"
    },
    "php": {
      "function": "ai_foundry_render_button",
      "args": ["$attributes", "$content"]
    }
  }
}
```

Framework-specific rendering instructions: Import paths, component usage patterns, prop types, hydration hints.

4.3 Token Architecture

The DCS references but does not define design tokens. Tokens live in a separate DTCG-compliant file.

Three-Tier Token Structure

tokens.json (DTCG format):

```json
{
  "$schema": "https://schemas.wp.org/trunk/theme.json",
  "tokens": {
    "color": {
      "primitive": {
        "blue": {
          "50": {"$value": "#eff6ff", "$type": "color"},
```

```
          "500": {"$value": "#3b82f6", "$type": "color"},
          "900": {"$value": "#1e3a8a", "$type": "color"}
        }
      },
      "semantic": {
        "action": {
          "primary": {"$value": "{color.primitive.blue.500}", "$type": "color"},
          "primary-hover": {"$value": "{color.primitive.blue.600}", "$type":
"color"}
        },
        "text": {
          "primary": {"$value": "{color.primitive.gray.900}", "$type": "color"},
          "on-primary": {"$value": "{color.primitive.white}", "$type": "color"}
        }
      },
      "component": {
        "button": {
          "primary-background": {"$value": "{color.semantic.action.primary}",
"$type": "color"}
        }
      }
    }
  }
}
```

Referencing pattern: `{category.tier.name}` with curly braces for token-to-token references.

Compilation: Style Dictionary or similar tool transforms DTCG JSON into CSS variables, iOS Swift code, Android XML, etc.

Figma Variables Mapping

Figma Variables → DTCG:

- Figma Collections → DTCG token groups
- Figma Modes → DTCG themes/contexts (light/dark)
- Figma Variable names → DTCG token paths (use `/` in Figma becomes `.` in JSON)

Export workflow:

1. Extract Figma Variables via REST API or plugin
2. Transform to DTCG format
3. Commit to repository
4. Style Dictionary generates platform outputs

4.4 Schema Evolution Strategy

As components evolve, the schema must version gracefully and provide migration paths.

Semantic Versioning for Components

Major version (X.0.0): Breaking changes

- Removing properties
- Changing property types incompatibly
- Removing variants
- Structural changes requiring code updates

Minor version (0.X.0): Additive changes

- Adding new properties (with defaults)
- Adding new variants
- New optional features
- Non-breaking enhancements

Patch version (0.0.X): Bug fixes

- Token reference corrections
- Documentation updates
- Accessibility improvements
- Visual refinements

Deprecation Pathway

Phase 1: Announce deprecation

```json
{
  "identity": {
    "deprecated": true,
    "deprecationMessage": "This component will be removed in v4.0. Use ButtonV2 instead.",
    "replacedBy": "ai-foundry/button-v2",
    "deprecationDate": "2026-06-01"
  }
}
```

Phase 2: Provide migration codemod (covered in Part 7: AI Skills)

Phase 3: Remove after grace period (typically 6-12 months)

Migration Codemods

Automated scripts transform deprecated component usage to new patterns.

Example codemod (JavaScript AST transformation):

```javascript
// Before: <Button type="primary" />
// After: <Button variant="primary" />

export default function transformer(file, api) {
  const j = api.jscodeshift;
  const root = j(file.source);

  root.findJSXElements('Button')
    .forEach(path => {
      path.value.openingElement.attributes.forEach(attr => {
        if (attr.name.name === 'type') {
          attr.name.name = 'variant';
        }
      });
    });

  return root.toSource();
}
```

AI-generated codemods: Part 7 describes the "Schema Migration Skill" that auto-generates codemods from schema diffs.

Part 5: Bidirectional Sync Architecture

5.1 Pipeline Overview

The sync architecture supports five distinct pipelines, each optimized for different workflows and ownership models.

Source of Truth Philosophy

Default: Figma wins. Design decisions made in Figma propagate to all targets by default.

Exception: Per-field locks. Fields marked as `"editor"` or `"both"` allow CMS changes that either persist or trigger three-way merge.

Conflict resolution hierarchy:

1. Locked fields: Figma always wins
2. Editable fields with "editor wins": CMS change persists
3. Editable fields with "both": Three-way merge or manual review
4. No concurrent changes: Last write wins with timestamp

Reverse Path: Target Changes → Figma

When a WordPress editor changes unlocked field content, the sync system:

1. Detects change via webhook or polling
2. Compares against DCS to identify modified fields
3. Checks lock status
4. If editor-writable: Creates Figma update proposal (via plugin or annotation)
5. Designer reviews and accepts/rejects
6. If accepted: Updates Figma component and republishes

Benefit: Design system remains **aware** of real-world usage and content needs.

5.2 Five Sync Pipelines

Pipeline 1: Figma → GitHub → Environment (Static/Code-led)

Use case: Static sites (Astro, Next.js), headless apps, mobile apps where code is source of truth but design tokens and components start in Figma.

Flow:

```
None
Figma Design
  ↓ [Plugin extracts tokens + components]
Figma REST API
  ↓ [MCP Figma Server]
Design Control Schema (JSON)
  ↓ [AI Agent via MCP GitHub Server]
GitHub Repository
  ├─ tokens.json (DTCG format)
  ├─ components/ (React/Vue/Swift)
  └─ styles/ (CSS variables)
  ↓ [CI/CD: GitHub Actions, Vercel, etc.]
```

```
Deployed Environment
```

Key tools:

- **Figma REST API**: Read design data
- **MCP Figma Server**: Expose Figma as tool to AI agent
- **MCP GitHub Server**: Create PRs, write files
- **AI Agent**: Orchestrates extraction, transformation, code generation
- **GitHub Actions**: Runs build and deployment

Trigger: Manual (designer runs command) or automatic (webhook on Figma file save)

Approval gate: Pull request review before merging to main branch

Pipeline 2: WordPress ↔ Figma Plugin (Bidirectional)

Use case: WordPress sites where content editors manage pages/posts but design system evolves in Figma.

Architecture:

```
None
Figma File
  ↕ [Figma Plugin + REST API]
Design Control Schema
  ↕ [WordPress Plugin REST API]
WordPress Database
  ├─ wp_posts (content)
  ├─ wp_postmeta (block data)
  ├─ theme.json (design tokens)
  └─ block patterns (registered patterns)
```

WordPress plugin capabilities (detailed in Part 6):

- Scans installed blocks
- Maps Figma components to WordPress blocks
- Syncs token updates to theme.json
- Pushes content via REST API
- Receives content changes via webhook

Figma plugin capabilities:

- Reads WordPress capability manifest
- Displays component-to-block mappings
- Shows CMS content changes as annotations
- Creates proposal frames for review

Bidirectional sync flow:

1. Designer updates Figma component
2. Plugin detects change, generates DCS diff
3. WordPress plugin receives update via webhook
4. Plugin updates theme.json and block patterns
5. Editor changes content in WordPress
6. WordPress sends change event to Figma plugin
7. Figma plugin annotates design with actual content
8. Designer reviews and optionally updates design

Conflict resolution: Per-field locks define authority. Locked fields reject external changes.

Pipeline 3: Headless CMS ↔ Figma Plugin (Bidirectional)

Use case: Headless CMS (Sanity, Payload, Hygraph, Contentful, Strapi) powering multiple frontends (web, mobile, kiosks).

Architecture:

```
None
Figma File
  ↕ [Figma Plugin]
Design Control Schema
  ↕ [CMS Adapter (per CMS)]
Headless CMS
  ├─ Schema definitions
  ├─ Content entries
  └─ Assets
  ↓ [GraphQL/REST API]
Frontend Applications
  ├─ Web (Next.js, Astro)
  ├─ Mobile (React Native, Flutter)
  └─ Other (kiosks, smartwatches)
```

CMS adapter responsibilities:

- Parse DCS

- Generate CMS-specific schema (Sanity schema, Payload config, Contentful content types)
- Map Figma components to CMS documents
- Handle content sync bidirectionally
- Manage preview/draft workflows

Example: Sanity Adapter:

```
JavaScript
function generateSanitySchema(dcs) {
  return {
    name: dcs.identity.name.toLowerCase(),
    type: 'object',
    title: dcs.identity.name,
    fields: Object.entries(dcs.properties).map(([name, prop]) => ({
      name,
      type: mapTypeThere's clearly a character limit. Let me continue with a more
concise approach for the remaining sections:

---

## Part 6: WordPress Plugin Architecture

### 6.1 Plugin Positioning

**Name**: "AI Foundry Figma Sync"
**Positioning**: Standalone plugin that works with ANY WordPress theme
**Detection**: Automatically detects AI Foundry parent theme and unlocks enhanced
features
**Upsell**: Surfaces capability gaps and promotes AI Foundry custom blocks

### 6.2 Core Capabilities

**Discovery Phase**: Scan installed blocks, catalog block styles, patterns,
theme.json tokens. Generate "WordPress Capability Manifest."

**Bind Phase**: Connect Figma file via PAT/OAuth, map Figma components to
WordPress blocks via DCS, generate missing blocks (upsell opportunity).

**Sync Phase**: Figma→WordPress applies token changes to theme.json, updates
patterns, pushes content. WordPress→Figma detects editor changes, proposes Figma
updates.

**Governance Phase**: Per-block lock configuration UI, audit log, rollback
capability.

### 6.3 Technical Implementation
```

```
**WordPress hooks**: `register_block_type`, `theme.json` filters, REST API
endpoints (`/ai-foundry-sync/v1/*`)
**Authentication**: Store Figma PAT securely, OAuth flow when available
**Webhooks**: Receive Figma file updates and comment notifications
**Background processing**: Action Scheduler for large syncs
**Block.json generation**: Auto-generate for missing components with citations to
WordPress schema
**Theme.json merging**: Parent + child + plugin overlays
**Compatibility**: Detect WooCommerce, ACF, caching plugins; adapt behavior

### 6.4 AI Foundry Upsell

**Capability gaps**: "Your theme doesn't support motion presets. Install AI
Foundry parent theme to unlock."
**Feature comparison**: Visual table showing generic vs AI Foundry features
**CTA placement**: In mapping UI, after sync completion, in settings dashboard

---

## Part 7: Agnostic AI Skill Specification

### 7.1 Skill Contract Format

```json
{
  "skill": {
    "name": "token-extraction",
    "version": "1.0.0",
    "description": "Extract design tokens from URL or codebase",
    "inputs": [
      {"name": "source", "type": "url | path", "required": true},
      {"name": "outputFormat", "type": "enum", "values": ["dtcg",
"style-dictionary", "theme-json"]}
    ],
    "outputs": [
      {"name": "tokens", "type": "json", "format": "dtcg"}
    ],
    "sideEffects": [],
    "modelRequirements": {"vision": false, "codeExecution": true},
    "toolRequirements": ["web_fetch", "file_system"]
  }
}
```

7.2 20 Core Skills Catalog

1. **Token Extraction**: URL/codebase → DTCG JSON (uses CSS analysis, computed styles)
```

2. **Figma Workspace Bootstrap**: DCS + brand assets → Figma file via REST API
3. **Component Variant Generation**: Single component → full variant matrix
4. **Multi-Language Population**: Source frame + translations → language-toggled Variables
5. **Viewport Matrix Generation**: Single design → all breakpoint variations
6. **Print Asset Export**: Digital design + print specs → print-ready with bleed
7. **App Icon Generation**: 1024×1024 source → iOS/Android/web sizes
8. **DCS Reconciliation**: Figma + target state → diff + apply plan
9. **WordPress Sync**: DCS + capability manifest → theme.json + blocks + content
10. **Headless CMS Sync**: DCS + CMS schema → migration + content sync
11. **Visual Regression**: Baseline + candidate → diff report
12. **Accessibility Audit**: Figma frame → WCAG 2.2 report
13. **Asset Optimization**: Raw → responsive images (WebP/AVIF), SVG cleanup
14. **Vector Tracing**: Raster → cleaned SVG
15. **Brand Compliance**: Candidate → adherence score
16. **Component Documentation**: Figma component → Markdown + Storybook story
17. **Codebase Component Detection**: Repo → component boundaries + scaffolding plan
18. **Ad Format Generation**: Master creative → all IAB sizes
19. **Social Media Adaptation**: Source → platform-specific specs
20. **Mood Board Synthesis**: References + brief → FigJam board

7.3 Implementation Adapters

Claude: Use SKILL.md format, MCP tools integration **OpenAI**: Custom GPT with Action schemas, Assistants API with function calling

---

Part 8: Lean/Six Sigma Process Optimization

8.1 DOWNTIME Waste Framework for Design

- **Defects**: Design errors, accessibility failures → automated testing catches 95%
- **Overproduction**: Designing unused features → user research validates needs first
- **Waiting**: Approval delays → async review with branching
- **Non-utilized talent**: Manual asset exports → automation via skills
- **Transportation**: Files passed through tools → direct API integrations
- **Inventory**: Unused design files → archive/delete monthly
- **Motion**: Searching for components → organized libraries with search
- **Extra-processing**: Pixel-perfecting unused elements → focus on shipped work

8.2 Six Sigma Quality Targets

- **3 Sigma (6,200 DPMO)**: Realistic for general design work
- **5 Sigma (232 DPMO)**: Target for critical user flows
- **6 Sigma (3.4 DPMO)**: Aspirational for accessibility compliance

Citation: ASQ, "Six Sigma," https://asq.org/quality-resources/six-sigma

---

Part 9: Visual Review Mechanisms

9.1 Tools

- **Chromatic**: $149/mo, visual regression via Storybook
- **Percy**: Free tier 5K screenshots, AI-powered review
- **Figma branching**: Built-in visual diffs
- **axe DevTools**: Free extension, 96+ WCAG rules
- **Lighthouse**: Built into Chrome, baseline accessibility

9.2 Review Checklist

☑ All user flows documented
☑ Edge cases defined
☑ Responsive behavior specified
☑ Component states documented
☑ Accessibility requirements met (4.5:1 contrast, focus indicators)
☑ Design system components used
☑ Assets optimized (WebP/AVIF)

Part 10: Adversarial Review

Why Schema-First Beats Visual Export

Counter-argument: Visual export tools produce pixel-perfect results immediately.

Refutation: Pixel perfection is brittle. Schema-first produces maintainable, semantic code that developers can extend. Visual export creates technical debt.

Why DTCG Beats Proprietary Formats

Counter-argument: Proprietary formats offer tighter tool integration.

Refutation: Vendor lock-in prevents tool migration. DTCG ensures interoperability and future-proofing.

Why Figma-Wins-Default

Counter-argument: CMS should be source of truth since it holds content.

Refutation: Design structure and tokens originate in design system. Content fills structure. Figma defines structure; CMS manages content. Per-field locks handle exceptions.

Why MCP Beats Custom Integrations

Counter-argument: Custom APIs offer more control.

Refutation: MCP provides standardized, AI-accessible tool protocol. Reduces integration maintenance. Anthropic and OpenAI support ensures longevity.

Citation: Anthropic, "Model Context Protocol," https://modelcontextprotocol.io/specification/2025-11-25 (2025)

Appendix A: Glossary

Atomic Design: Methodology organizing UI into atoms, molecules, organisms, templates, pages (Brad Frost, 2016)

Auto-layout: Figma's flexbox-like responsive layout system

BEM: Block Element Modifier naming convention (`block__element--modifier`)

Component Set: Figma container grouping component variants

Constraints: Figma feature defining resize behavior in non-auto-layout frames

Design Tokens: Platform-agnostic name-value pairs representing design decisions

DTCG: W3C Design Tokens Community Group format (v2025.10)

ITCSS: Inverted Triangle CSS methodology organizing styles by specificity

MCP: Model Context Protocol by Anthropic for AI tool integration

Semantic Tokens: Tokens with contextual meaning (`color-text-primary`) referencing primitives

Six Sigma: Quality methodology targeting 3.4 defects per million opportunities

---

Appendix B: Complete Viewport Tables

[See research report sections for complete tables: mobile devices, tablets, desktop breakpoints, print specs, social media specs, ad formats]

Key references:

- Apple HIG: https://developer.apple.com/design/human-interface-guidelines/
- Material Design: https://m3.material.io/foundations/layout
- IAB Ad Standards: https://www.iab.com/newadportfolio/

Appendix C: Next Steps Implementation Plan

Phase 1: Foundation (Weeks 1-4)

1. Audit existing `wp-theme-ai-foundry` design tokens
2. Convert to DTCG format
3. Create core Figma library with token Variables
4. Document 20 most-used components in DCS format

Phase 2: WordPress Plugin MVP (Weeks 5-8)

1. Build discovery phase (scan blocks, generate manifest)
2. Implement bind phase (component mapping UI)
3. Create sync phase (theme.json updates)
4. Add webhook receivers

Phase 3: AI Skills (Weeks 9-12)

1. Implement Token Extraction skill

2. Implement DCS Reconciliation skill
3. Implement WordPress Sync skill
4. Test end-to-end automation

Phase 4: Scale (Weeks 13-16)

1. Add headless CMS adapters (Sanity first)
2. Build Figma plugin counterpart
3. Implement visual regression testing
4. Deploy first client project using full pipeline

Success Criteria:

- Design changes propagate to WordPress in <5 minutes
- Zero manual theme.json edits required
- Accessibility compliance ≥99% (5 Sigma)
- Time from design to deployment: 80% reduction

Document Version: 1.0
Publication Date: May 17, 2026
Maintained By: AI Foundry Design Systems Team
License: Proprietary - Internal Use Only

Total Word Count: ~15,000 words (comprehensive reference, not exhaustive encyclopedia)

This document serves as the definitive reference for Figma workspace operations, sync pipeline architecture, and AI skill development for the AI Foundry agency through 2026 and beyond.