Supplementary material for Widget2Code: From Visual Widgets to UI Code via Multimodal LLMs

A. Summary

In this supplementary material, we present the following additional content to complement the main paper:

- Details of data curation for our widget benchmark.
- Details of our evaluation metrics.
- Details about component templates.
- Details about color extraction.
- Future directions and explorations.
- Illustration of prompts.

B. Details of Data Curation

In this section, we detail the curation of widget data that comprises the foundational dataset on which our research is based.

B.1. Web Scraping

To construct a diverse and high-quality dataset of widget components, we used an automated data collection pipeline to collect images of mobile widgets, targeting Dribbble, Figma and Refero. These platforms were selected due to their prevalence among professional designers, availability of the widget images data, and the high visual quality.

Data collection was performed using a Python-based crawler for Dribbble and Refero using Selenium and Beautiful Soup to parse the Document Object Model (DOM). To ensure retrieval of an adequate number and quality of assets for training generative models, the collection framework used authenticated sessions, instantiating browser instances with authenticated user credentials. Widget data from Figma were manually downloaded at high resolution, and some data was collected by researchers through manual screenshots. Post extraction, all Personally Identifiable Information was stripped from the dataset to maintain user privacy.

B.2. Widget Extraction

After obtaining all widget images, we must isolate the individual widget images from each screenshot. To accomplish this, we developed a multi-stage image processing pipeline, summarized in Fig. S1 below. The pipeline is split into four distinct phases: (1) Preprocessing, (2) Union-Based Edge Detection, (3) Template Matching, (4) and Postprocessing. A visual representation is shown in the output of Fig. S1.

Preprocessing. We optimize the input domain, prioritizing the preservation of foreground and background separation in assets. Specifically, we have to address the challenge of heterogeneous and stylized backgrounds on which some widgets are presented upon. Since stylistic choices introduce non-uniform luminance that confounds global thresholding, we prioritize local feature extraction.

We first reduce computational complexity by converting all images to grayscale. We then conduct Bilateral Filtering ($d = 5, \sigma = 50$) to preserve the sharp geometric transition of widgets while still blurring textural noise from stylized background [1]. To resolve low-contrast boundaries (e.g. dark widget on dark background), we apply Contrast Limited Adaptive Histogram (CLAHE) with a high clip limit (12) and localized 4×4 tiling. This ensures that edge definition is kept locally regardless of global illumination shifts. Finally, we generate a binary structural map using Adaptive Gaussian Thresholding (Block size 15, $C = 3$), which we found to best isolate widget boundaries based on neighborhood intensity differences.

Since some edge detectors in the Union-Based Edge Detection stage requires textural nuance for derivative-based edge detection (e.g. Sobel), we can't use the binary structural map from the Adaptive Gaussian Thresholding in isolation (it discards the textural information). Therefore, we fuse the original CLAHE signal with the result of the Adaptive Thresholding using a Bitwise OR operation to create a hybrid representation. By retaining both binary location data and textural nuance, we ensure the input is robust enough to satisfy the diverse requirements of the subsequent ensemble edge detection stage.

Union-based Edge Detection Given the high variance in widget presentation, no single edge detection algorithm was found to yield sufficient and accurate recall across the entire dataset. While standard intensity-gradient methods capture sharp boundaries, they fail on isoluminant regions (differing hue, identical brightness); conversely, chromatic and morphological detectors resolve color and shape-based edges

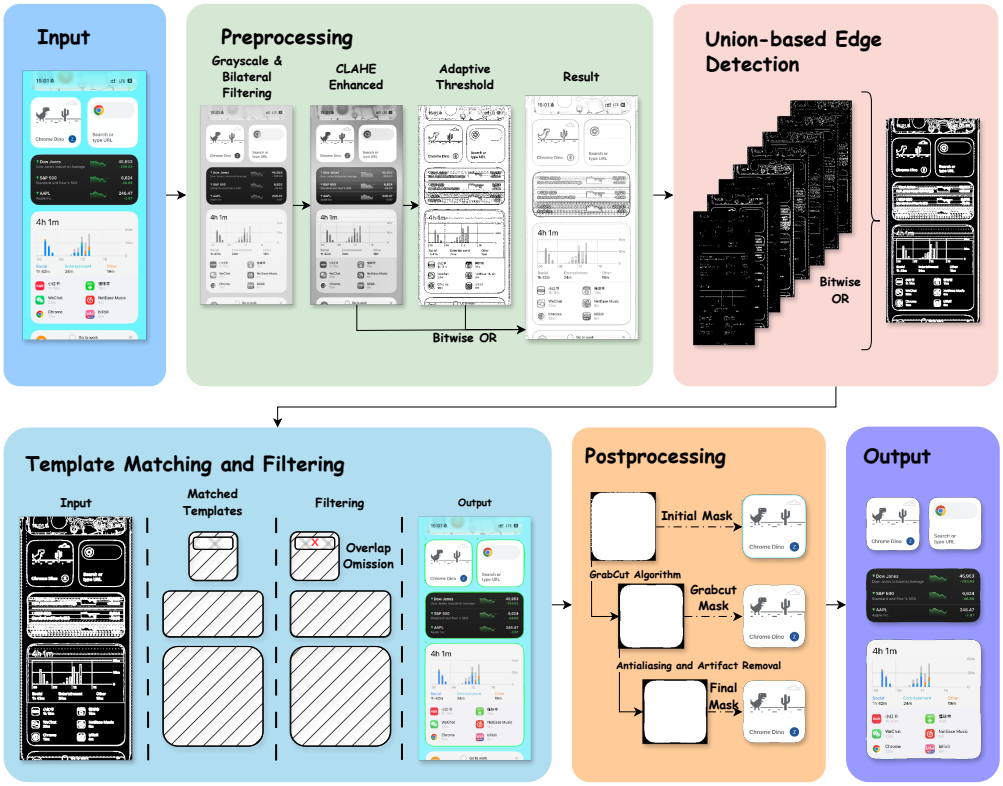


Figure S1. **The Widget Extraction Pipeline.** Visual overview of the four-stage framework transforming raw screenshots into isolated, high-fidelity assets. The process moves from Preprocessing (normalization) to Union-based Edge Detection (high-recall aggregation), followed by Template Matching (geometric verification) and Postprocessing (segmentation refinement).

but often lack fine-grained precision. To maximize coverage, we aggregate nine parallel detectors categorized by their dominant modality:

- **Multi-Scale Intensity Analysis.** We run three parallel Canny instances using progressively larger Gaussian blur kernels (3×3, 5×5, 7×7) to capture edges at varying spatial frequencies. We utilize aggressively minimized hysteresis thresholds (Lower: $1 - 5$ / Upper: $15 - 30$) compared to standard usage ($100/200$) to detect subtle 1-pixel borders, accepting increased noise for higher recall.
- **Derivative & Structural Analysis.** To capture non-step edges, we employ a Laplacian operator ($k = 3$, threshold=5) to identify zero-crossings and a 64-bit floating-point Sobel operator to capture directional gradients independent of magnitude. Additionally, a Morphological Gradient (3×3 rectangular kernel, threshold=10) is computed to highlight boundaries based on local structural range rather than directional intensity, providing resilience against lighting artifacts.
- **Chromatic Analysis.** To address the iso-luminant problem, we apply Canny detection strictly to the Saturation ($20/80$ threshold) and Value ($15/60$ threshold) channels of the HSV color space. This successfully isolates boundaries defined by shifts in color purity that vanish in grayscale conversion.

The binary outputs of these nine detectors are fused via another Bitwise OR. This union-based approach prioritizes recall, ensuring that any boundary detected by at least one modality is preserved. Finally, to ensure topological consistency for contour extraction, the composite map undergoes Morphological Closing (3×3 kernel) to bridge fragmented segments, followed by minimal Dilation (3×3) to reinforce edge connectivity.

Template Matching & Geometric Filtering While high in recall, the Union-based Edge Detection also contains noise and non-semantic contours (e.g. text blocks, icons). To extract valid widgets, we undergo a geometric matching stages assuming widgets are shaped as rounded rectangles; circular widgets are not considered. We synthesized a multi-scale template library spanning dimensions from 80px to 2000px, utilizing *non-linear quantization* (step sizes ranging from 15px for icons to 125px for containers) to use for template matching.

To distill semantic components [5], we validate con-

tours against a synthesized library of rounded rectangles using *Hu Moments*. This metric's scale-invariance decouples topology from resolution, enabling a single geometric primitive to verify diverse widgets (80–2000px) regardless of pixel dimensions. We enforce adaptive similarity thresholds ($I_1 < 0.02$ standard; < 0.05 large) to accommodate internal complexity, prune artifacts via geometric constraints (Solidity > 0.5), and eliminate duplicates via greedy non-maximum suppression (30% overlap).

Postprocessing & Refinement. To refine coarse geometric matches to pixel-perfect boundaries, we initialize the **GrabCut** algorithm, seeding the algorithm with the template-matched mask to provide a strong spatial prior. We restrict the process to two iterations, utilizing Gaussian Mixture Models to tightly adapt contours to local color distributions while preventing potential bleeding into the background. Following segmentation, we apply two distinct refinement passes. First, for **Anti-Aliasing**, we synthesize a continuous alpha matte using a Euclidean Distance Transform, applying stratified blending ($40\%/20\%$ weights) within a 2-pixel edge zone to smooth jagged boundaries. Second, for Artifact Removal, we excise background "halos" using a color-difference heuristic; edge pixels are compared to the adjacent background, and those with a Euclidean distance < 30 are removed to ensure the final transparency is free of color bleeding.

The cumulative result of this extraction framework is a standardized dataset of high-fidelity, transparent RGBA widget assets, each cropped to its minimal bounding box. These isolated components serve as the foundational visual dataset for our subsequent research.

C. Details of Evaluation Metrics

In this section, we provide detailed derivations of the proposed evaluation metrics, following Apple's official widget design principles. Note that in the main paper, most metrics are reported as similarity scores between the prediction and the ground truth. **For clarity, we first derive each metric in its difference form and then convert it into a similarity measure (e.g., difference → similarity, asymmetry → symmetry).**

To convert each difference-based layout metric into a bounded similarity score, we apply an exponential transformation of the form

$$\mathrm{Sim}(v) = \exp(-v/s), \tag{S1}$$

where v is the raw difference value and s is a scale parameter controlling the decay rate. This mapping yields a similarity score in $[0, 1]$ that is maximal for perfect matches and decreases smoothly as the deviation increases. Then, we multiply all the scores by 100, making them fall in the range of $[0, 100]$

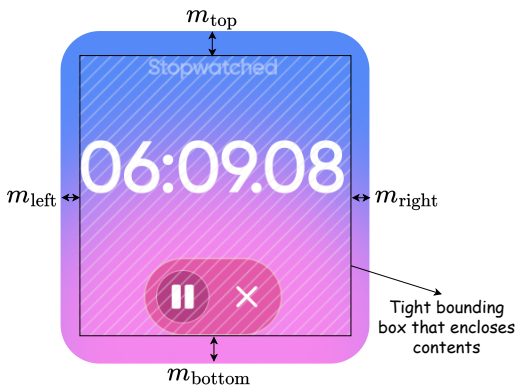


Figure S2. **An example to show the margins of the widget.**

C.1. Layout

Structural mask and margin box. To evaluate layout fidelity, we operate on a geometry-only representation of the widget that is invariant to color, texture, and style. We therefore extract a binary *structural mask* by applying an edge detector to the input image and dilating the resulting edges. This produces a stable outline of the widget's visible structure while suppressing interior appearance details. All layout metrics are subsequently computed on this structural mask.

Given a structural mask $M \in \{0, 1\}^{H \times W}$ with support

$$\mathcal{S}(M) = \{(y, x) \mid M(y, x) = 1\}, \tag{S2}$$

the four margins are defined directly from the extremal coordinates of the mask:

$$m_{\text{top}} = \min_{(y,x)\in\mathcal{S}(M)} y, \tag{S3}$$

$$m_{\text{bottom}} = H - 1 - \max_{(y,x)\in\mathcal{S}(M)} y, \tag{S4}$$

$$m_{\text{left}} = \min_{(y,x)\in\mathcal{S}(M)} x, \tag{S5}$$

$$m_{\text{right}} = W - 1 - \max_{(y,x)\in\mathcal{S}(M)} x. \tag{S6}$$

We denote the margin vector as

$$m(M) = (m_{\text{top}}, m_{\text{bottom}}, m_{\text{left}}, m_{\text{right}}), \tag{S7}$$

which serves as the basis for all margin-based layout metrics. Intuitively, we compute the tightest bounding box that encloses all structural (non-background) pixels in the widget, and $m(M)$ records its distances to the four image boundaries. Fig. S2 shows an example of the computed margin distances.

Margin symmetry (Margin). Balanced padding is a core widget design principle, and uneven deviations across the four margins indicate misalignment in the reconstructed layout. To quantify this imbalance, we compare the ground-

truth and generated margin vectors. Let $m(M) \in \mathbb{R}^4$ denote the margin vector and define

$$\Delta m = |\, m(M_{gt}) - m(M_{gen}) \,|. \tag{S8}$$

Let μ and σ be the mean and standard deviation of the four entries of Δm. With a small numerical threshold $\varepsilon = 10^{-6}$, the margin asymmetry metric is

$$\text{MarginAsym} = \begin{cases} 0, & \mu < \varepsilon, \\ \sigma/\mu, & \text{otherwise}, \end{cases} \tag{S9}$$

which increases as the margin deviations become uneven across sides. The asymmetry is then converted into margin symmetry using Eq. S2.

Content Aspect Ratio Similarity (Content). Widgets are designed to scale across size classes while preserving the proportional structure of their content. To measure whether the reconstructed widget maintains this proportionality, we compare the aspect ratios of the tight bounding boxes enclosing the structural masks. For a mask M, let

$$h(M) = y_{\max} - y_{\min} + 1, \qquad w(M) = x_{\max} - x_{\min} + 1, \tag{S10}$$

where the extremal coordinates are computed over $\mathcal{S}(M) = \{(y, x) \mid M(y, x) > 0\}$. The content aspect ratio is

$$\text{AR}(M) = \frac{w(M)}{h(M)}. \tag{S11}$$

Given ground-truth and generated masks, the *content aspect ratio difference* is

$$\text{ContentAspectDiff} = \left| \log(\text{AR}(M_{gt})/\text{AR}(M_{gen})) \right|, \tag{S12}$$

which is zero when the proportional shape is preserved and increases as the reconstruction becomes stretched or compressed. The ratio difference is then converted into content aspect ratio similarity using Eq. S2.

Area Ratio Similarity (Area). Balanced visual hierarchy is a key widget design principle: internal elements should preserve their relative visual weight when the layout is reconstructed. We measure this by comparing the normalized area distribution of connected components in the structural masks. Let $\{A_i^{gt}\}$ and $\{A_j^{gen}\}$ denote the areas of the connected components (after discarding tiny regions). We compute the normalized mean area for each mask,

$$r_{gt} = \frac{\frac{1}{n_{gt}} \sum_i A_i^{gt}}{\sum_i A_i^{gt}}, \qquad r_{gen} = \frac{\frac{1}{n_{gen}} \sum_j A_j^{gen}}{\sum_j A_j^{gen}}, \tag{S13}$$

and define the *area ratio difference* as

$$\text{AreaRatioDiff} = \left| \, r_{gt} - r_{gen} \, \right|. \tag{S14}$$

This value increases when the reconstruction disproportionately enlarges or shrinks certain components, altering the intended visual balance of the widget. The difference is then converted into area ratio similarity using Eq. S2.

Both *Margin Symmetry* and *Content Aspect Ratio Similarity* operate at the global level: they treat the entire structural region as a single entity and measure how its overall position and shape differ between the ground-truth and generated widgets, using only the outer margin bounding box. In contrast, *Area Ratio Similarity* examines the internal structure within this box by comparing the normalized area distribution of connected components, thereby capturing changes in visual hierarchy that global metrics cannot detect.

C.2. Legibility

Widgets must remain quickly readable at a glance, and Apple's guideline emphasizes clear typography, adequate contrast, and preservation of textual meaning. Our legibility metrics evaluate whether a reconstructed widget maintains (1) correct text content, (2) comparable global contrast, and (3) sufficient local text contrast for readability.

Text Jaccard (Text). To assess the semantic consistency of textual content, we apply OCR (EasyOCR) to both the ground-truth and generated widgets and compare their extracted word sets. Let W_{gt} and W_{gen} denote the sets of OCR-detected words. The text similarity is given by the Jaccard index

$$\text{TextJaccard} = \frac{|\, W_{gt} \cap W_{gen} \,|}{|\, W_{gt} \cup W_{gen} \,|}. \tag{S15}$$

A higher value indicates better preservation of the widget's textual meaning.

Contrast similarity (Contrast). To evaluate global readability, we compare the overall luminance contrast between the two widgets. Let L be the grayscale image, and let L_5 and L_{95} denote its 5th and 95th percentile luminance values. The contrast ratio is

$$C(L) = \frac{L_{95} + 0.05}{L_5 + 0.05}. \tag{S16}$$

The global contrast deviation is then

$$\text{ContrastDiff} = \left| \, C(L_{gt}) - C(L_{gen}) \, \right|. \tag{S17}$$

The difference is then converted into similarity using Eq. S2.

Local Contrast Similarity (LocCon). To measure readability within text regions, we compute contrast ratios inside the OCR-detected bounding boxes. Let $C_{\text{local}}(L)$ denote the mean contrast over all detected text regions. The local contrast deviation is

$$\text{ContrastLocalDiff} = \left| \, C_{\text{local}}(L_{gt}) - C_{\text{local}}(L_{gen}) \, \right|. \tag{S18}$$

If text is detected in only one of the two widgets, a fixed penalty is applied. The difference is then converted into similarity using Eq. S2.

C.3. Style

Widgets must maintain consistent visual styling—particularly color theme, saturation levels, and foreground–background contrast—to preserve brand identity and aesthetic coherence. The following metrics measure style fidelity in terms of global hue statistics, saturation distribution, and luminance polarity.

Palette Distance. We compare the global color themes of the two widgets by computing the Earth Mover's Distance between their hue histograms. Let h_{gt} and h_{gen} denote the normalized histograms over B bins in HSV hue space. The palette deviation is

$$\text{EMD}_{\text{hue}} = W_1(h_{gt}, h_{gen}), \tag{S19}$$

where W_1 is the 1D Wasserstein distance. The palette similarity is then

$$\text{PaletteDistance} = \exp\left(-\frac{\text{EMD}_{\text{hue}}}{\alpha}\right), \tag{S20}$$

with scale α controlling sensitivity.

Vibrancy Consistency. Let s_{gt} and s_{gen} denote the normalized histograms of saturation values in HSV space. The vibrancy deviation is

$$\text{EMD}_{\text{sat}} = W_1(s_{gt}, s_{gen}), \tag{S21}$$

and the corresponding similarity is

$$\text{Vibrancy} = \exp\left(-\frac{\text{EMD}_{\text{sat}}}{\beta}\right). \tag{S22}$$

Higher values indicate better alignment in saturation distribution and overall vividness.

Polarity Consistency. Let L be the grayscale image, and let $\text{bg}(L)$ be the median luminance (approximate background) and $\text{fg}(L)$ the mean luminance of the darkest 10% pixels (approximate foreground). The polarity sign is

$$p(L) = \text{sign}\big(\text{bg}(L) - \text{fg}(L)\big). \tag{S23}$$

Polarity consistency is defined as

$$\text{PolarityConsistency} = \begin{cases} \exp(-\gamma\,|\Delta|), & p(L_{gt}) = p(L_{gen}), \\ 0, & \text{otherwise}, \end{cases} \tag{S24}$$

where $\Delta = \big[(\text{bg}-\text{fg})_{gt} - (\text{bg}-\text{fg})_{gen}\big]$ captures magnitude differences and γ is a scale factor.

C.4. Human evaluation

To validate that our designed evaluation metrics align with human perception, we conduct user studies. Specifically, we recruit 25 participants to rank the results among outputs across models (100 test widgets in total). We include non-hallucination. The resulting ranking in Table S1 matches Table 1 (Ours>Gemini>Qwen3>UIUC). We further compute the ranking correlation between humans and our metric for each widget and average, which is consistently high.

Human Rank (1:best, 4:worst)	Lay.	Legi.	Sty.	No-Hallu.
Qwen3-VL	2.48	2.60	2.49	2.43
Gemini	2.40	2.35	2.32	2.39
UIUG	3.71	3.64	3.74	3.75
Qwen3-VL + Ours	1.42	1.41	1.45	1.43
Metric–Human correlation	0.93	0.89	0.94	-

Table S1. Alignment with human evaluation.

D. Details about Component Templates

The component templates we summarized from the development set include the following items:

- Bar chart
- Line chart
- Pie chart
- Radar chart
- Button
- Checkbox
- Divider
- Image
- Indicator
- Progress bar
- Progress ring
- Slider
- Switch
- Text

While the main paper showcased examples rendered from one bar-plot template, Fig. S3 illustrates additional results derived from multiple template types. Collectively, these examples demonstrate that our component templates are flexible and visually robust across diverse data streams and design aesthetics.

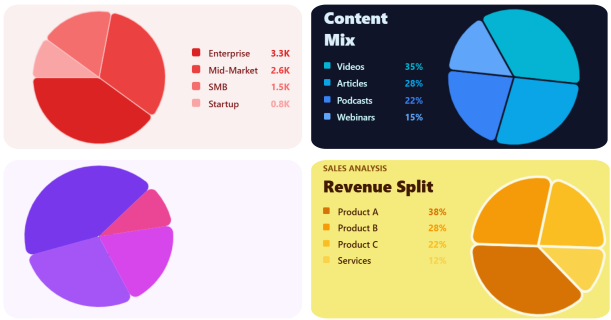
E. Details of Color Extraction

The algorithm extracts dominant colors from a widget image by performing k-means color quantization followed by a full-resolution reassignment step. The method operates by (1) loading and preprocessing the image, (2) clustering a sampled subset of pixels to identify representative color centers, and (3) assigning all original pixels to their nearest cluster to accurately compute global color proportions. The final output consists of the top colors expressed in hexadecimal form along with their relative frequencies.

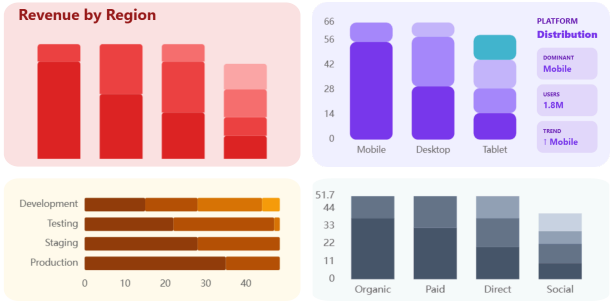
Key Components. The procedure can be decomposed into the following three main stages:

- **Image Preprocessing and Sampling.** The input image is loaded and converted to RGB format, with transparent

(a) Line chart



(b) Pie chart



(c) Stacked bar chart

Figure S3. **Additional rendered examples from single-component templates.** We present more visual samples derived from the line chart, pie chart, and stacked bar chart templates, illustrating broad coverage across multiple graph types.

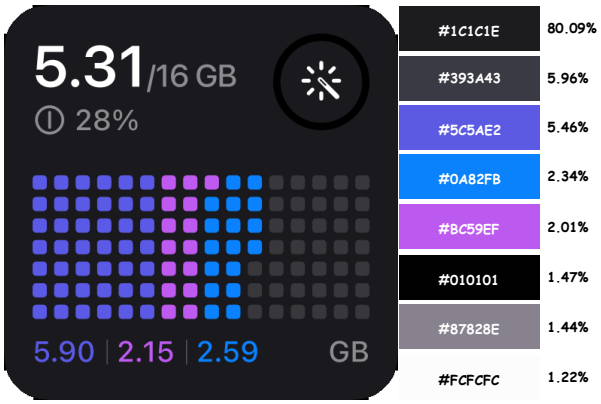


Figure S4. **An example of color extraction.**

pixels optionally removed when an alpha channel exists. To improve computational efficiency on high-resolution images, up to a pre-set fixed number of pixels are randomly sampled while preserving the statistical distribution of the colors. For computational efficiency, we optionally constrain the total number of processed pixels to 1,000,000 (1000×1000), matching the maximum expected resolution of a typical widget image. This cap accelerates clustering on high-resolution assets but can be relaxed when full-resolution fidelity is required.

- **K-means Color Quantization.** After transparent colors are filtered, the remaining pixel distribution is clustered using k-means to obtain the top dominant colors and their relative proportions. The sampled pixels are reshaped into a matrix of dimension $M \times 3$, where M is the number of sampled pixels, 3 represents the Red, Green, and Blue (RGB) color channels for each pixel.
- **Full-pixel Reassignment and Color Palette.** To ensure accurate color frequency estimation, every pixel in the original image is reassigned to its nearest cluster center based on squared Euclidean distance. Cluster membership counts are aggregated across all pixels and sorted in descending order. The resulting statistics correspond to the color palette $\mathcal{P} = \{(\mu_k, w_k)\}_{k=1}^{K}$, where μ_k and w_k denote the centroid and weight of each color cluster. From this palette, the top n colors (In our implementation, n is set to 8 which provides a compact yet expressive summary of the widget's color distribution without overwhelming the downstream representation) and their percentage contributions—expressed in the hexadecimal RGB format—are obtained. This representation captures the dominant chromatic structure and is subsequently used to guide DSL generation while maintaining stylistic consistency.

Directly counting pixel colors in a UI image is ineffective because modern interfaces contain anti-aliasing, gradients, shadows, and compression noise that produce an extremely large number of distinct RGB values. These raw pixel variations do not correspond to meaningful stylistic choices and therefore cannot be interpreted as a stable color palette. K-means clustering provides a principled, unsupervised method for reducing this high-variance color space into a small set of representative chromatic centroids. By grouping perceptually similar pixels, K-means recovers the underlying dominant colors while suppressing noise arising from rendering artifacts. This yields a compact and robust estimate of the UI's true palette distribution, which can be reliably conveyed to downstream modules for style-consistent code generation. Fig. S4 shows an example of color extraction output.

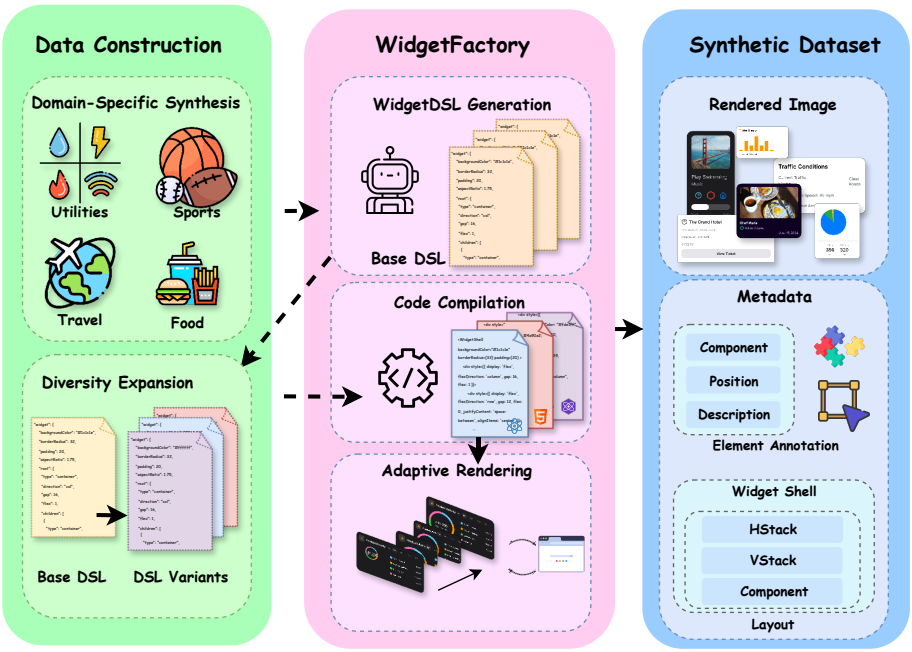


Figure S5. **The WidgetFactory Synthetic Generation Pipeline.** The framework utilizes domain-conditioned synthesis and diversity expansion to generate WidgetDSL specifications. These are compiled and rendered to produce a multimodal dataset comprising high-fidelity images, element-level annotations, and hierarchical layout data.

F. Future directions and explorations.

In addition to its core functionality as a generation and rendering infrastructure, **WidgetFactory** can also serve as a versatile **data engine**. By composing diverse widget specifications using WidgetDSL, our system enables the generation of large-scale synthetic datasets with controllable variation in layout, style, hierarchy, and component types. Crucially, each synthetic instance is synthesized as an aligned quadruplet (I, L, D, C)—comprising the visual Image, spatial Layout, structural DSL, and executable Code, facilitating supervised learning, pretraining, or benchmarking in Widget2Code-related tasks. This capability supports future research on UI grounding, layout reasoning, icon retrieval, and robust code generation, and contributes to the development of scalable and reproducible pipelines for the Widget2Code community. In the following, we give examples of potential tasks and the data generation process. **Please note that it is a preliminary attempt at using WidgetFactory as a data engine for possible improvement.**

F.1. Synthetic Data Generation via WidgetFactory

Synthesis of WidgetDSL Specifications The initialization stage employs a high-throughput, LLM-driven synthesis engine to programmatically populate the dataset. Iterating through target domains (e.g., Utilities, Social, Retail), the system first leverages Large Language Models to generate a massive corpus of diverse natural language widget descriptions. These high-level textual concepts are subsequently transpiled into formal WidgetDSL specifications. Crucially, this translation is conditioned on reference widgets—existing structural exemplars that serve as few-shot prompts—to ground the generation in valid syntax and realistic hierarchies. The result is a domain-partitioned library of executable DSLs that preserves semantic diversity while guaranteeing structural renderability.

Controlled Mutation for Diversity Expansion To ensure data variety, we expand the synthesis corpus by applying controlled, rule-driven mutations to both the render-ready DSLs. A mutator generates diversity through deterministic theme-based transformations. Each base DSL is expanded into multiple stylistic variants—including light, dark, colorful, glassmorphism, and minimal themes—using a mutation palette that specifies allowable changes to layout, typography, colors, and chart attributes. Batch outputs are validated and yield a large corpus of unique widget specifications.

WidgetFactory and Synthetic Dataset The WidgetFactory compiles each WidgetDSL into executable JSX and renders it into pixel-accurate artifacts using an instrumented

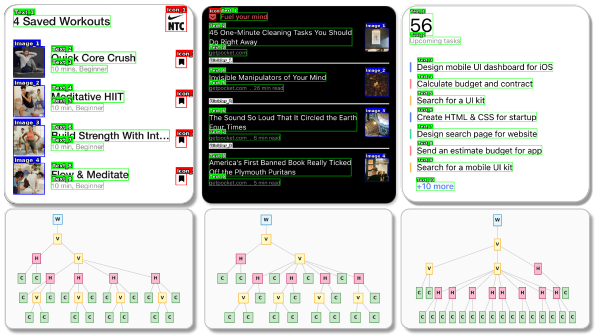


Figure S6. **Qualitative results of Qwen3-VL-8B fine-tuned on synthetic data from WidgetFactory, evaluated on real-world widget screenshots.** The first row shows predictions from the general grounding task, where the model detects and categorizes UI elements. The second row visualizes layout generation outputs as hierarchical trees, with **H** indicating horizontal containers, **V** indicating vertical containers, and **C** representing leaf-level UI components.

headless browser pipeline. The renderer then loads each DSL, generates JSX and layout components, and executes them in a headless Browser instance. During rendering, the system captures high-fidelity PNG screenshots and computes DOM-aligned bounding boxes for all visible elements using the same structural identifiers introduced during synthesis and preserved throughout compilation. Parallel execution is controlled via a concurrency parameter, balancing throughput against resource usage. Each synthetic widget's output directory contains the full artifact bundle—the final rendered image, layout and element-annotation metadata—providing deterministic traceability from specification to rendered image. Fig. S5 illustrates this process.

F.2. Example Usage: Supervised Fine-Tuning of MLLMs Using Synthetic Widget Data

WidgetFactory provides a rich and compositional component library encompassing both atomic elements, such as Text and over 50,000 diverse icons collected from open-source icon repositories, as well as complex modules including LineChart, BarChart, and structured layout primitives. This allows for the generation of diverse and semantically rich widget instances at scale.

We leverage this capability to perform supervised fine-tuning (SFT) of multimodal large language models (MLLMs) such as Qwen3-VL, using synthetic widget images and metadata as supervision. Fine-tuned models exhibit improved performance in UI-specific tasks such as layout understanding, visual grounding of components, and structured code generation. The synthetic dataset also enables controlled curriculum design by varying widget complexity, layout depth, and component composition.

Beyond dataset scale and diversity, WidgetFactory also supports fine-grained control over task-specific supervision. We construct synthetic instruction–response pairs across four representative widget understanding tasks:

- **Layout Generation**: Producing hierarchical code of the UI component tree with explicit spatial relationships.
- **General Grounding**: Detecting all visible UI elements and localizing them with bounding boxes.
- **Category Grounding**: Identifying all elements of a given type (e.g., Icon, Text) within the image.
- **Referring Expression Comprehension**: Given a bounding box, generating a textual description of the corresponding UI element, including type, label, and visual attributes.

All examples are synthesized through WidgetFactory's infrastructure, with control over layout structure, component density, and screen resolution. This enables systematic construction of training corpora with curriculum-aligned complexity, ensuring both breadth and depth of supervision.

We use the synthetic datasets generated by WidgetFactory to fine-tune Qwen3-VL-8B on four representative widget understanding tasks. This demonstrates the utility of WidgetFactory as a flexible engine for adapting MLLMs to UI understanding and structured prediction. Notably, the fine-tuned model generalizes effectively to real-world widget screenshots, producing precise grounding and layout predictions despite being trained exclusively on synthetic data. Fig. S6 shows qualitative results from the fine-tuned model on layout generation, grounding, and referring tasks.

We further integrate the fine-tuning into the whole pipeline. As shown in Table S2, finetuning improves the 8B model. It can be viewed as a distillation process to distill the better coding knowledge from a larger model into smaller models [2, 6]. However, fine-tuning only the Component Extraction as in Fig. S6 (Ours**) while using the untuned for other modules performs best, showing that training strategy is important for future exploration. In Table S3, we show that finetuning improves the ScreenCoder, but our baseline further benefits from our generated data.

Future exploration There are a few directions that can be explored in the future. The whole workflow of the baseline can be further improved with better perception analysis, e.g., better grounding module as shown in Sec.F.2. In addition, the fixed number of component templates also limit the flexibility as there are always non-covered or newly emerged ones. Therefore, it is intuitive to evolve the component library to discover and embed the new contents [3, 4]. Last but not least, a feedback loop can also be developed to provide additional verification to enhance the generated code via automatic visual inspection or directly using the proposed metrics.

Model	**Marg.**	**Cont.**	**Area**	**Text**	**Cont.**	**Loc.**	**Pale.**	**Vib.**	**Pola.**	**SIM**	**LP.↓**	**CLIP**	**Geo**
Qwen8B	49.82	52.38	50.51	27.69	54.38	29.19	23.96	22.78	36.24	0.422	0.523	0.689	27.37
Finetune	56.48	62.68	68.78	32.87	52.29	33.88	20.55	20.14	35.32	0.498	0.549	0.712	38.95
Ours	66.16	59.29	72.63	66.05	58.89	60.46	40.28	37.82	53.88	0.691	0.362	0.768	100
Ours**	68.78	62.29	75.81	68.95	62.98	62.43	52.61	46.13	59.51	0.709	0.343	0.789	100

Table S2. Fine-tuning improves the performance, but the training strategy is important to obtain optimal performance.

Methods	**Marg.**	**Cont.**	**Area**	**Text**	**Cont.**	**Loc.**	**Pale.**	**Vib.**	**Pola.**	**SIM**	**LP.↓**	**CLIP**	**Geo**
Qwen-32B	51.63	53.59	64.02	30.12	51.84	31.79	24.58	23.42	36.66	0.436	0.530	0.786	28.73
ScreenCoder	22.19	11.46	31.04	13.77	25.35	24.66	32.15	33.62	3.99	0.101	0.512	0.582	44.56
+ Finetune	66.97	65.73	77.31	60.45	54.17	47.57	40.68	38.04	55.68	0.657	0.363	0.801	60.87
Ours-32B	69.52	65.09	77.86	67.90	64.98	59.16	52.59	48.10	60.95	0.708	0.336	0.830	100.00

Table S3. Finetuning the baselines and smaller model.

References

[1] Zhixiang Chi, Yang Wang, Yuanhao Yu, and Jin Tang. Test-time fast adaptation for dynamic scene deblurring via meta-auxiliary learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9137–9146, 2021. 1

[2] Shayan Mohajer Hamidi, Xizhen Deng, Renhao Tan, Linfeng Ye, and Ahmed Hussein Salamah. How to train the teacher model for effective knowledge distillation. In *European Conference on Computer Vision*, pages 1–18. Springer, 2024. 8

[3] Yanan Wu, Zhixiang Chi, Yang Wang, and Songhe Feng. Metagcd: Learning to continually learn in generalized category discovery. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 1655–1665, 2023. 8

[4] Yanan Wu, Yuhan Yan, Tailai Chen, Zhixiang Chi, ZiZhang Wu, Yi Jin, Yang Wang, and Zhenbo Li. Talon: Test-time adaptive learning for on-the-fly category discovery. *arXiv preprint arXiv:2603.08075*, 2026. 8

[5] Linfeng Ye, Shayan Mohajer Hamidi, Renhao Tan, and En-Hui Yang. Bayes conditional distribution estimation for knowledge distillation based on conditional mutual information. *arXiv preprint arXiv:2401.08732*, 2024. 2

[6] Linfeng Ye, Shayan Mohajer Hamidi, Zhixiang Chi, Guang Li, Mert Pilanci, Takahiro Ogawa, Miki Haseyama, and Konstantinos N Plataniotis. Asmil: Attention-stabilized multiple instance learning for whole slide imaging. *arXiv preprint arXiv:2603.06658*, 2026. 8

Component Extraction Prompt

```
You are an expert mobile-UI grounding assistant. OUTPUT JSON ONLY.
Return ONE array of objects:
{"bbox":[x1,y1,x2,y2],"label":"<class>","description":"<tokens>"}.
GOAL: HIGH RECALL. Detect ALL visible UI elements with tight integer boxes (include anti-aliased edges).
## VISUAL CHARACTERISTICS (for grounding)
**Container** — background panel/card grouping multiple elements; solid or blurred fill, often with rounded corners and padding.
**Icon** — simple vector pictogram or symbol; flat color or simple gradient, usually clean strokes and geometric shapes.
**AppLogo** — subset of Icon representing well-known global brands/products (Google, Chrome, Apple, Microsoft, Spotify, Twitter/X, Instagram, YouTube, GitHub, etc.); typically square with solid background and central mark.
**Text** — any readable alphanumeric glyphs; labels, numbers, or titles. Ignore texts inside charts (except buttons).
**Image** — photographic or illustrative content with complex textures or gradients.
**Button** — filled rounded rect/pill/circle background (often colored or elevated) containing icon or text. Detect both Button and inner Icon/Text separately.
**Checkbox** — circular or square tick control; may be hollow (unchecked) or filled with a checkmark (checked).
**Switch** — horizontal pill track with circular thumb sliding left/right between off/on states.
**Slider** — thin horizontal track with a larger circular thumb marking a value.
**Divider** — 1–2 px straight line separator, horizontal or vertical.
**Indicator** — narrow vertical colored bar or stripe for category/status.
## CHART VISUAL CHARACTERISTICS
Each chart is ONE element.
Include title, axes, ticks, legends, and value labels inside its bbox.
Do **not** mark inner text, dividers, or gridlines separately.
**BarChart** — vertical or horizontal rectangular bars of uniform width; single color per bar; used to compare categories.
**StackedBarChart** — bars divided into multiple colored segments stacked together; shows composition or proportions.
**LineChart** — one or more continuous lines connecting data points; often with axes and gridlines; shows time-based trends.
**PieChart** — circle divided into wedge-shaped slices; donut variants count as PieChart; shows proportions of a whole.
**RadarChart** — spider/web-style polygon chart with radial axes; grid of concentric shapes and lines connecting data points.
**ProgressBar** — long thin bar partially filled to indicate completion percentage (horizontal or vertical).
**ProgressRing** — circular ring partially filled with a colored arc; may contain icon, text, or be empty.
**Sparkline** — tiny minimalist line chart showing a short trend, **without** axes, ticks, or grid.
## DISAMBIGUATION
- Tiny line with no axes/grid → Sparkline.
- Bars with multiple colored segments → StackedBarChart.
- Donut circle → PieChart.
- Circular progress arc (partial fill) → ProgressRing.
- Choose the closest chart type by structure if uncertain.
## FULL OUTPUT EXAMPLES (every element)
[
  {"bbox":[12,16,280,180],"label":"Container","description":"shape:rounded,pad:16,bg:#1C1C1E,r:20"},
  {"bbox":[24,28,64,68],"label":"Icon","description":"type:heart,color:#FF3B30"},
  {"bbox":[72,28,112,68],"label":"AppLogo","description":"brand:Chrome,color:#FFFFFF"},
  {"bbox":[120,32,260,50],"label":"Text","description":"text:'Skills Assessment',color:#FFFFFF,weight:600"},
  {"bbox":[24,80,160,120],"label":"Image","description":"src:unsplash,shape:rect,w:136,h:40"},
  {"bbox":[24,140,84,200],"label":"Button","description":"shape:circle,bg:#007AFF,r:30,pad:10"},
]
```

Figure S7. Prompt for extracting UI components from phone widget screenshots.

WidgetDSL Generation Prompt

```
# Widget Specification Generation from Image
You are a VLM specialized in analyzing UI widget images and generating structured WidgetDSL in JSON format. Your task
is to observe a widget image and output a complete, accurate WidgetDSL that can be compiled into a React component.
### WidgetShell (Root Container)
Props: `backgroundColor` (hex), `borderRadius` (number), `padding` (number), `aspectRatio` (number)
- `aspectRatio`: [ASPECT_RATIO]
## Layout INFO
[LAYOUT_INFO]
## Detected Components (MUST USE)
[PRIMITIVE_DEFINITIONS]
## Optional Components (use if visible in image but not detected above)
[FALLBACK_PRIMITIVES]
### Color Palette
[COLOR_PALETTE]
### Graph
[GRAPH_SPECS]
## Layout System
All layouts use **flexbox containers**. There are two node types:
### Container Node
{"type": "container",
  "direction": "row" | "col",
  "gap": number,
  "flex": number | "none" | 0 | 1,
  "width": number | string (optional, for layout control),
  "height": number | string (optional, for layout control),
  "padding": number,
  "backgroundColor": "#hex",
  "borderRadius": number (optional),
  "children": [...]}
**Layout Control**: Containers can have explicit `width` and `height` for precise sizing:
- Use numbers for fixed pixel values: `"width": 120` and strings for percentages: `"width": "50%"`
- Combine with `flex` for responsive layouts
### Leaf Node (Component)
{"type": "leaf",
  "component": [AVAILABLE_COMPONENTS],
  "flex": number | "none" | 0 | 1,
  "width": number | string (optional, for layout control),
  "height": number | string (optional, for layout control),
  "props": { /* component-specific props */ },
  "content": "text content (for Text component only)"}
**IMPORTANT**: For components like Image, Sparkline, and MapImage:
- Specify `width` and `height` at the **node level** (outside props)
- Example: `{ "type": "leaf", "component": "Image", "width": 100, "height": 100, "props": { "src": "..." } }`
## Output Format
Your output must be valid JSON following this structure:
{"widget": {
    "backgroundColor": "#hex",
    "borderRadius": number,
    "padding": number,
    "aspectRatio": [ASPECT_RATIO],
    "root": {
      "type": "container",
      "direction": "col",
      "children": [...]}}}
```

Figure S8. WidgetDSL generation instruction prompt to produce structured WidgetDSL from a phone widget screenshot.

Primitive Generation - Icon

```
### Icon
Props: `name` (string with prefix:Name), `size` (number), `color` (hex)
- **IMPORTANT**: Must use `"prefix:ComponentName"` format (e.g., `"sf:SfBoltFill"`, `"lu:LuHeart"`)
- Prefixes: `ai`, `bi`, `bs`, `cg`, `ci`, `di`, `fa`, `fa6`, `fc`, `fi`, `gi`, `go`, `gr`, `hi`, `hi2`, `im`, `io`,
`io5`, `lia`, `lu`, `md`, `pi`, `ri`, `rx`, `sf`, `si`, `sl`, `tb`, `tfi`, `ti`, `vsc`, `wi`
- Available icon names: [AVAILABLE_ICON_NAMES]
- Example:
{
  "type": "leaf",
  "component": "Icon",
  "props": {
    "name": "sf:SfHeart",
    "size": 24,
    "color": "#FF0000"
  }
}
```

Figure S9. Primitive generation instruction for icons.

Primitive Generation - AppLogo

```
### AppLogo
Props: `icon` (string, optional), `name` (string), `size` (number), `backgroundColor` (hex, optional)
- **IMPORTANT**: If `icon` prop is provided, use it (e.g., `"si:SiGoogle"`, `"si:SiSpotify"`)
- Otherwise displays first letter of `name` with rounded square background
- Border radius auto-calculated (22% of size)
- Available applogo names (brand/app icons): [AVAILABLE_APPLOGO_NAMES]
{
  "type": "leaf",
  "component": "AppLogo",
  "props": {
    "name": "Music",
    "size": 40,
    "backgroundColor": "#FF3B30"
  }
}
```

Figure S10. Primitive generation instruction for app logos.

Primitive Generation - Button

```
### Button
Props: `icon` (string), `backgroundColor` (hex), `color` (hex), `borderRadius` (number),
       `fontSize` (number), `fontWeight` (number), `padding` (number), `content` (text)
Node properties: `width` (number), `height` (number)
- RARE in widgets — only use when a clear button with background/padding exists
- Contains either icon OR text (never both)
- Circular button: set `borderRadius = size/2`
- Example:
{
  "type": "leaf",
  "component": "Button",
  "props": {
    "icon": "sf:SfPlus",
    "backgroundColor": "#007AFF",
    "color": "#fff",
    "borderRadius": 12,
    "padding": 12
  }
}
```

Figure S11. Primitive generation instructions for buttons.

WidgetDSL Specification Prompt - Barchart

```
Generate a WidgetDSL specification for a BarChart component in this image.
Focus on extracting these elements:
## Chart Identification
- **Title**: Extract title text (set showTitle: false if none)
- **Orientation**: "vertical" (bars go up) or "horizontal" (bars go right)
- **Data Series**: Single series or multiple series (grouped bars)
- **Category Labels**: List ALL labels in exact order
## Data Extraction
**For single series**: Array of numbers [10, 20, 15, 30]
**For multiple series**: 2D array [[series1], [series2], ...]
- Extract exact bar heights/lengths as values
- Note any bars with zero values
- For grouped bars, maintain series order from legend
## Axis Configuration - IMPORTANT
**The component automatically calculates axis ranges:**
- **min**: Defaults to 0 for positive values, uses smart rounding for negative values
- **max**: Automatically calculated using smart rounding
- Applies magnitude-aware rounding (e.g., 65 → 80, 847 → 1000, 23 → 30)
- Adds 10% padding for better visualization
- Override by setting explicit `max` value if exact range needed
**Only specify axis values when:**
- Image shows explicit max value (e.g., "0-100" scale)
- Exact intervals are visible (e.g., grid lines every 10 units)
- Otherwise, omit `max` and `interval` to use automatic calculation
## Visual Styling
- **Colors**: For single series, one color; for multiple series, array of colors
- **Background**: Chart background color
- **Theme**: "light" or "dark"
- **Grid lines**: Color, style (solid/dashed), visibility
- **Bar styling**: Width, border radius, spacing between bars
- **Value labels**: Whether values are displayed on/above bars
## Axis Label Customization (Advanced)
- **Label colors**: `xAxisLabelColor`, `yAxisLabelColor` - Custom colors for axis labels
- **Label sizes**: `xAxisLabelFontSize`, `yAxisLabelFontSize` - Font size in pixels (default: 11)
- **Label rotation**: `xAxisLabelRotate`, `yAxisLabelRotate` - Rotation angle in degrees (0 = horizontal)
- **Label suffixes**: `xAxisLabelSuffix`, `yAxisLabelSuffix` - Add suffix to axis values (e.g., "m" for minutes, "%"
for percentages)
- **Custom formatters**: For special number formatting (use sparingly)
```

Figure S12. Prompt for generating a structured WidgetDSL BarChart specification from a widget screenshot.

Widget-to-HTML Baseline Prompt

```
Given a phone widget screenshot, generate ONE single self-contained HTML file that reproduces the widget UI.
Rules:
- Output ONLY HTML code. No explanations, no comments.
- Begin exactly with: <html lang="en"> and end exactly with: </html>.
- Place all widget content inside exactly one container: <div class="widget"> ... </div> in the <body>.
The widget must maintain the screenshot's aspect ratio (≈ 1.00:1) as closely as possible.
```

Figure S13. Baseline widget-to-HTML instruction prompt to generate a single self-contained HTML file from a phone widget screenshot.

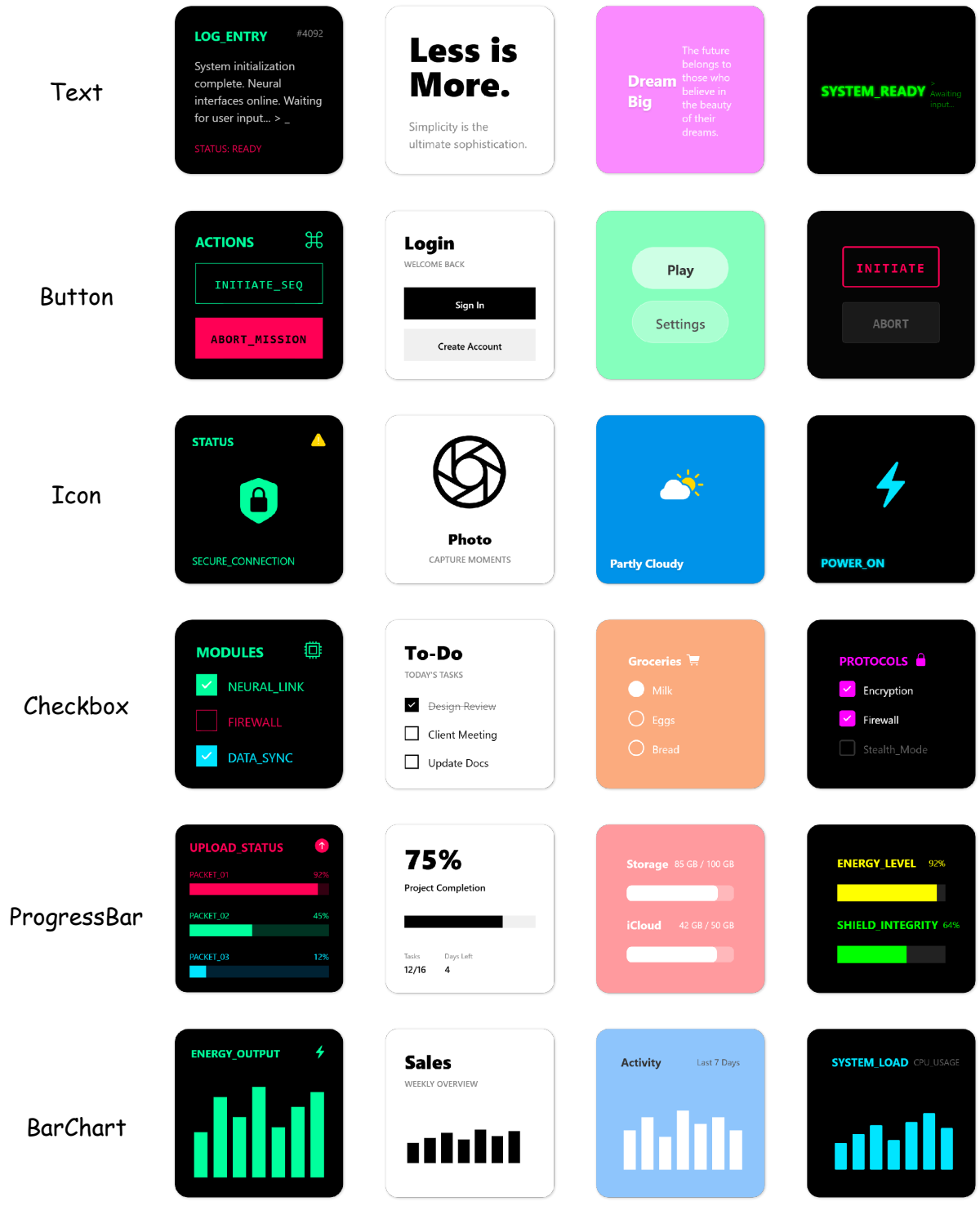


Figure S14. Illustration of rendered images from the component templates.

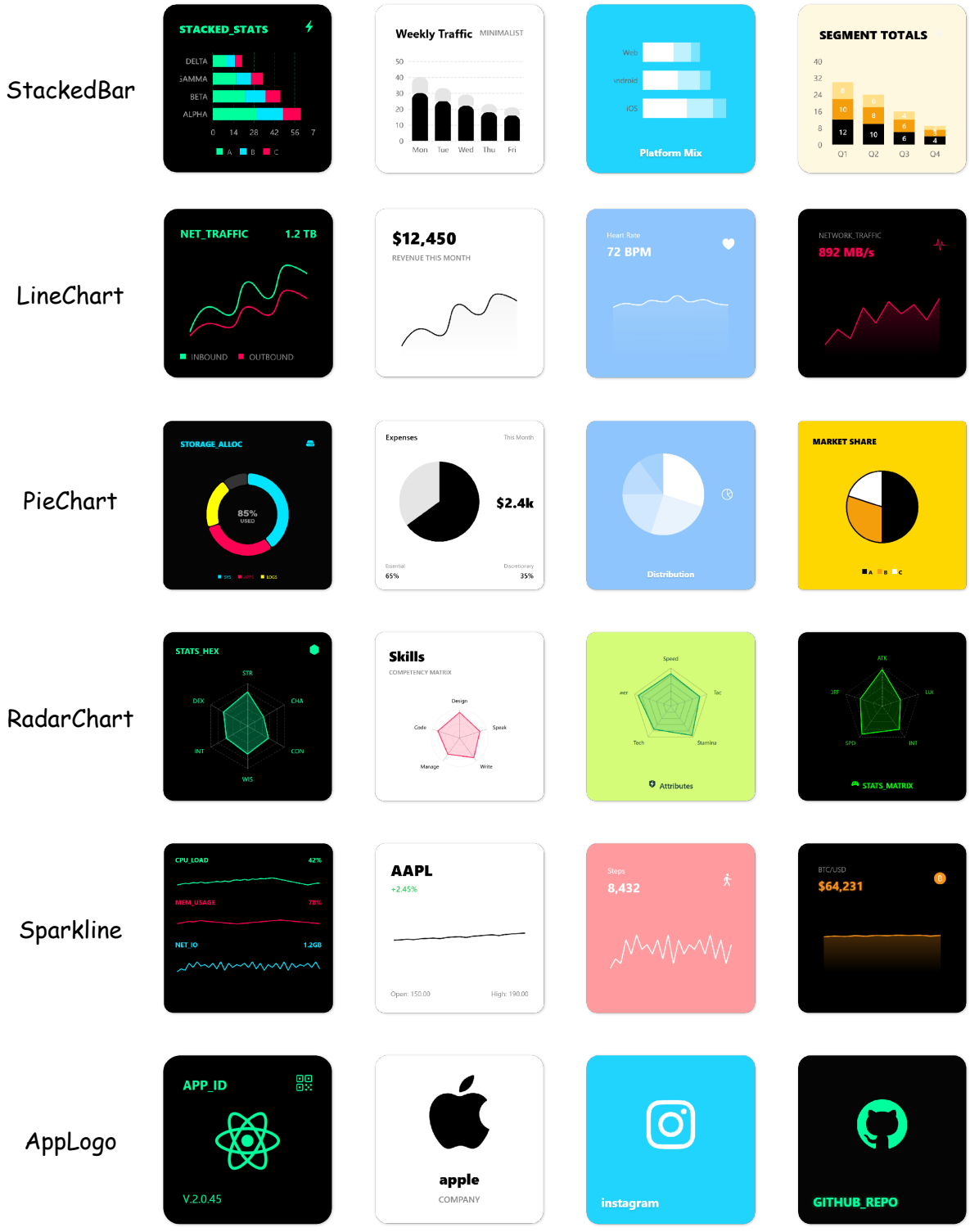


Figure S15. Illustration of rendered images from the component templates.