A Full Definition of SVG-Specific Tokens

As introduced in Section 3.2 of the main text, representing SVG charts as raw character-level XML strings is inefficient and highly susceptible to syntactic hallucinations. To enable the Vision-Language Model (VLM) to better capture the semantic and hierarchical structure of charts, we extend the tokenizer vocabulary with 115 SVG-specific special tokens.

This tokenization strategy not only significantly compresses the sequence length but also forces the model to learn chart-oriented vector representations rather than unstructured text. As detailed in Table 6, these 115 tokens are systematically categorized into four functional groups based on their roles in chart rendering and data encoding:

- **1. Structural Containers & Anchors:** This group includes paired start and end tags for document scopes, groupings (e.g., <g>), definitions (<defs>), and advanced rendering blocks (e.g., gradients and filters). These tokens are critical for maintaining the hierarchical DOM structure of the chart, allowing multiple primitive marks to be logically grouped (e.g., grouping bars of the same data series) to support downstream Chart Structure Graph (CSG) extraction and layout reuse.

- **2. Graphical Primitives:** This group covers the fundamental vector shapes utilized to represent data marks, including basic geometric shapes (<rect>, <circle>, <line>), complex trajectories (<path>), and typography (<text>, <tspan>). By explicitly tokenizing their opening and closing tags, the model is guided to generate strictly bounded and syntactically valid primitive components.

- **3. Geometric Opcodes:** To accurately reconstruct highly customized shapes and continuous data boundaries (e.g., in area charts or radar charts), we tokenize the low-level path commands. We explicitly differentiate between absolute (abs) and relative (rel) coordinate instructions (e.g., moveto, lineto, curveto). This allows the model to precisely execute complex trajectory planning without being distracted by raw character parsing.

- **4. Visual Channel & Geometry Attributes:** This group encapsulates the attributes that map underlying data to visual representations. It includes spatial coordinates and dimensions (x=, width=), color and styling aesthetics (fill=, stroke=), typographic properties (font-size=, text-anchor=), and affine transformations (transform=). Formulating these properties as discrete tokens enables the model to disentangle visual styling from geometric positioning.

The complete list of the 115 SVG-specific special tokens and their corresponding descriptions is provided in Table 6.

B Prompt Design for Chart Structure Graph Construction and Manipulation

To enable language-guided chart manipulation, our framework employs a two-stage prompting strategy. First, the LLM analyzes the parsed SVG to extract a Scene Graph, explicitly inferring topological dependencies and structural hierarchies. Second, leveraging this graph and the raw SVG code, the LLM executes downstream user requests—dynamically switching between an *Analysis Mode* (for numerical insights) and an *Editing Mode* (for visual modifications).

Both prompts are carefully engineered to enforce strict structural preservation. By assigning an expert persona and defining mandatory constraint resolution steps (e.g., coordinate shifting and canvas expansion), we prevent visual artifacts and ensure the edited SVGs remain valid and geometrically consistent. The complete templates are provided below:

System Prompt for Scene Graph Extraction

[System Role]
You are an expert SVG scene graph generator.

You are given an SVG string and a Scene Graph JSON template.
Your task is to analyze the SVG and populate the template with concrete values.

Requirements:
1. Strictly follow the provided JSON schema. Do NOT change field names or structure.
2. You MAY expand list-type fields (e.g., marks, ticks, legend items) to include multiple instances if needed.
3. Do NOT add new keys that are not defined in the templates.
4. Only fill in values that can be directly inferred from the SVG.
5. If a value cannot be determined, set it to null.
6. Preserve all hierarchical relationships and nesting.

Output Format:
- Return ONLY a valid JSON object for the scene graph.
- Do NOT include any explanations, comments, or extra text.

System Prompt for CSG-Guided Chart Reasoning and Manipulation

[System Role]
You are an expert SVG editor.

[If `analysis_mode` is True:]
You are given:
1. An SVG chart
2. The dependency schema representing the relationships between elements
3. A user question
Task
Answer the question by analyzing the SVG and dependencies. Provide a clear, direct answer.
Output
Return plain text only. Do not output SVG.

[If `analysis_mode` is False (Editing Mode):]
You are given:
1. An SVG chart
2. The dependency schema representing the relationships between elements
3. An editing instruction
Task
Perform a **global, structure-aware edit** of the SVG.
—
Process (MANDATORY)
1. Identify directly affected elements and their original values according to the svg code and scene graph.
2. Trace dependencies using the provided JSONs (Pay special attention to `layout_cascade`).
3. CALCULATE the new quantitative values. If mark width/height changes and padding is fixed, mathematically deduce the new axis length and viewBox dimensions.
4. Apply edits in a consistent order: element → layout → axis → dependent elements. Do not skip dependency reasoning.
CRITICAL CONSTRAINTS (DO NOT VIOLATE)
1. **ZERO DELETION**: You MUST preserve EVERY single element from the original SVG (axes, text labels, legends, ALL segments and colors of stacked bars). Do NOT delete any nodes unless explicitly requested.
2. **NO SHORTCUTS**: Do NOT use comments like `<!- rest of the code ->` or `....` You MUST output the complete, fully renderable SVG code.
3. **COORDINATE SHIFTING**: If you change the thickness/size of an element, you MUST mathematically recalculate and shift the position (e.g., `x` or `y` attribute) of ALL subsequent elements and groups. You must manually add the size difference to their coordinates to maintain the original padding/gap. Elements MUST NOT overlap.
4. **CANVAS EXPANSION**: If the total computed layout size increases, you MUST expand the `<svg viewBox="...">` and axis lines accordingly so nothing is clipped.
Requirements
1. Preserve stacking, alignment, and spacing constraints.
2. Maintain consistent gaps and proportions.
3. Ensure all elements remain within valid axis ranges.
Output Format (STRICT)
Output the COMPLETE modified SVG inside an ```` ```xml ```` code block.
Goal
Modify the SVG while preserving all structural relationships defined by the provided JSONs.

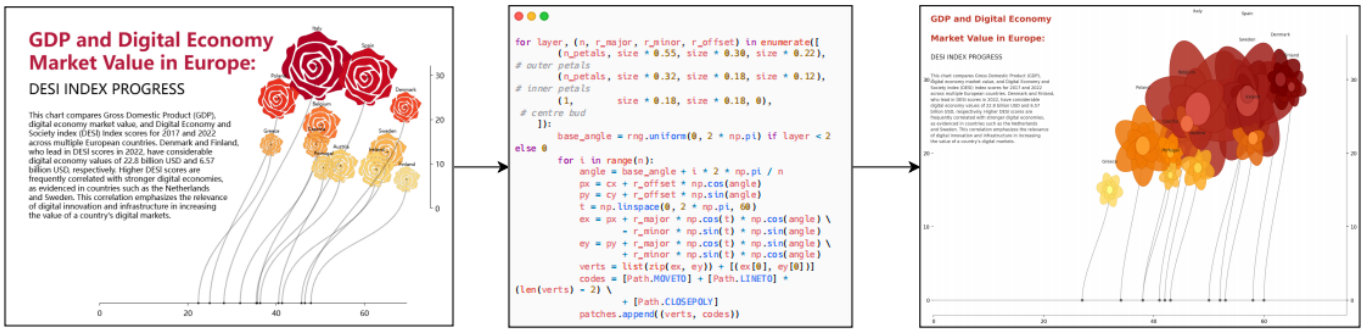

Fig. 7: Failure case of chart-to-code methods on a highly customized infographic. (Left) Original raster chart using rose-shaped glyphs to jointly encode GDP by size, DESI score by color, and country ranking by position. (Center) A chart-to-code approach attempts to reproduce the customized rose marks with Matplotlib; because the library lacks native support for such pictorial glyphs, the petals must be manually approximated using parameterized ellipses. (Right) The resulting rendering captures only the coarse layout and data encoding, while failing to preserve the distinctive rose-petal geometry and visual style of the original chart. This example illustrates the difficulty of chart-to-code methods in reconstructing highly customized pictorial marks with predefined plotting primitives.

C GRPO IMPLEMENTATION DETAILS

Reward Design

All three visual reward terms are normalized to $[0,1]$ before aggregation to ensure balanced contribution and to prevent any single term from dominating the combined reward due to differing numerical scales.

Foreground IoU. For $\mathrm{IoU}_{\mathrm{fg}}$, we apply a relaxed foreground matching strategy to tolerate minor spatial misalignments common in autoregressive SVG generation. Foreground pixels are identified as those whose grayscale value falls below a threshold of 245 (on a 0–255 scale), separating chart content from the background. The predicted foreground mask $\hat{M}$ is dilated with a 5×5 structuring element prior to intersection computation, while the union is computed from the original undilated masks:

$$\mathrm{IoU}_{\mathrm{fg}}(\tilde{I}_i, I) = \frac{|\mathrm{dilate}(\hat{M}) \cap M|}{|\hat{M} \cup M| + \varepsilon}, \tag{9}$$

where ε is a small constant preventing division by zero, and the final score is clipped to $[0,1]$ to handle the asymmetric dilation that may cause the numerator to exceed the denominator. The dilation tolerates sub-pixel rendering offsets between the predicted and ground-truth SVG without penalizing structurally correct but slightly misaligned reconstructions.

PSNR. For PSNR, the raw decibel value is normalized by a ceiling of 40 dB, which encompasses the practical reconstruction quality range observed across all models in our setting (typically 2–16 dB, as reported in Tables 1–2 of the main text):

$$\mathrm{PSNR}(\tilde{I}_i, I) = \min\left(\frac{20\log_{10}(255) - 10\log_{10}(\mathrm{MSE} + \varepsilon)}{40},\ 1\right), \tag{10}$$

where MSE is the mean squared pixel error between $\tilde{I}_i$ and I, and ε handles the degenerate case of near-perfect reconstruction. In practice, $\mathrm{PSNR}_{\mathrm{norm}}$ operates in $[0.05, 0.39]$ for our models, a range comparable to or lower than the $\mathrm{IoU}_{\mathrm{fg}}$ and SSIM terms, confirming that PSNR does not dominate the combined reward despite its unbounded raw scale. Note that this normalization applies only within the GRPO visual reward; the PSNR reported in Tables 1–2 uses the standard unbounded decibel scale for comparability with prior work.

SSIM. For SSIM, we compute a global structural similarity over Gaussian-smoothed (σ=1.5) grayscale images:

$$\mathrm{SSIM}(\tilde{I}_i, I) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1} \cdot \frac{2|\sigma_{xy}| + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}, \tag{11}$$

where μ_x, μ_y and σ_x^2, σ_y^2 denote the global mean and variance of the two Gaussian-smoothed images, σ_{xy} their cross-covariance, $C_1 = (0.01\times255)^2$, and $C_2 = (0.03\times255)^2$. Unlike the standard SSIM formulation, which uses the signed cross-covariance σ_{xy}, we take its absolute value $|\sigma_{xy}|$ to ensure the reward remains non-negative even when the predicted and ground-truth images are weakly anti-correlated in local structure, avoiding spurious negative reward signals during early training.

Combined Reward. The three normalized terms are averaged uniformly to form the visual reward, as in Eq. (6) of the main text:

$$R_{\mathrm{vis}}(\tilde{I}_i, I) = \frac{1}{3}\Big(\mathrm{IoU}_{\mathrm{fg}}(\tilde{I}_i, I) + \mathrm{PSNR}_{\mathrm{norm}}(\tilde{I}_i, I) + \mathrm{SSIM}(\tilde{I}_i, I)\Big). \tag{12}$$

This combined visual reward is further weighted against the code-level reward R_{code} as described in Eq. (7), with $\lambda_{\mathrm{vis}} = 0.6$ and $\lambda_{\mathrm{code}} = 0.2$ (Section5), ensuring that optimization prioritizes faithful visual reconstruction while still penalizing invalid or excessively verbose SVG outputs.

Relation to Domain-general Visual RL

Our reward design differs from domain-general visual RL approaches such as MSRL [34] and Reason-SVG [38] in two key respects. First, both MSRL and Reason-SVG target code generation or general illustration SVG synthesis, where computing perceptual similarity over the full canvas is appropriate; for chart images, the foreground-aware $\mathrm{IoU}_{\mathrm{fg}}$ is necessary to prevent background regions from dominating the reward signal. Second, MSRL uses execution-based rewards that require access to ground-truth data tables, whereas our rewards operate entirely in pixel space, requiring no auxiliary data source beyond the input raster image itself.

D BASELINE SETTINGS AND COMPARATIVE ANALYSIS

Implementation Details. All baselines use officially released checkpoints without any fine-tuning on Beagle+ or chart-specific data, and are evaluated in a zero-shot manner under a unified prompt (shown in Fig. 8). Input images are resized to 512×512 for all models. Since autoregressive generation introduces stochasticity, all results are averaged over four independent runs per sample. We set the maximum output length to 8192 tokens across all models to ensure complete generation without truncation. StarVector-8B is evaluated with temperature= 1.5, repetition_penalty= 3.1; OmniSVG-3B with temperature= 0.3, top_p= 0.90, repetition_penalty= 1.05; and Chart2SVG with temperature= 0.75, repetition_penalty= 1.05. ChartCoder generates executable Matplotlib/Seaborn code rather than SVG; we execute the output and save the rendered image for evaluation. GPT-4o is queried with temperature= 0 for deterministic outputs and its SVG is parsed directly without post-hoc rendering repair.

Analysis of Baseline Performance. StarVector and OmniSVG perform substantially worse on chart reconstruction for two compounding reasons. First, both models are trained predominantly on icon, emoji, and

illustration datasets [26, 40], with no chart-specific training data; their learned priors are poorly suited to the structured geometric regularity of visualization charts. Second, both models represent all geometric shapes exclusively via `<path>` commands with Bézier sequences, even for primitives such as rectangles and lines that can be expressed far more compactly as `<rect>` or `<line>` elements. This path-based representation consumes substantially more tokens per visual element than semantic primitives: a single bar in a bar chart, for instance, requires a closed five-point path (`M...L...L...Z`) instead of a single `<rect>` tag. Chart SVGs are inherently more complex than icons or emoji—comprising axes, ticks, gridlines, legends, and multi-series marks—whereas typical icon SVGs can be fully expressed within 1,024 tokens. The combination of domain mismatch and token-inefficient representation makes faithful chart reconstruction beyond the effective capacity of these models, as reflected in their high failure rates and fragmented outputs in Table 1 and Fig. 4.

ChartCoder, while chart-aware, is trained on Chart2Code-160K—a fully synthetic dataset generated by GPT-4o using predefined attribute seeds with only Matplotlib and Seaborn as target libraries [42]. This limits its robustness on real-world charts with diverse styles, custom layouts, and non-standard visual encodings. As illustrated in Fig. 7, when reconstructing a customized pictorial infographic, ChartCoder cannot faithfully represent the rose-shaped glyphs using the predefined primitives provided by Matplotlib and instead approximates the petals with manually parameterized ellipses. Moreover, its executable Python-code output does not directly recover the original SVG structure or preserve its constituent graphical elements.

GPT-4o achieves competitive performance as a strong proprietary baseline but still falls short of Chart2SVG on most metrics, particularly in structural fidelity and failure rate, reflecting the benefit of chart-specific training and rendering-aware optimization.

Data Value Extraction Methodology. We evaluate data value accuracy (Table 3) using DIVI-based heuristic parsing of the rendered SVG rather than a VLM-based extraction oracle. We adopt this rule-based approach because numerical value extraction remains an open challenge across the field: even state-of-the-art VLMs such as o3 and Gemini-2.5-Flash achieve only moderate accuracy when reading data values directly from chart images, and dedicated chart-to-table models are typically restricted to charts with explicit, unambiguous value annotations or standard layouts, limiting their applicability to the diverse and customized charts in our evaluation. A rule-based parser operating directly on SVG geometry avoids the compounding uncertainty of VLM-based reading and provides a more reproducible, deterministic measurement of how faithfully mark geometry encodes the underlying data.

This limitation is not specific to Chart2SVG: no current chart-to-SVG or chart-to-code system embeds a dedicated, robust mechanism for precise numerical recovery, and we view this as an open problem for the broader field rather than a shortcoming unique to our method. We note, however, that the structured SVG representation produced by Chart2SVG offers a practical advantage over raster-only approaches in this regard: because mark geometry, axis ticks, and labels are recovered as separate, addressable elements rather than fused pixels, the reconstructed SVG can be readily combined with an external OCR module for tick-label reading, or with future versions of Chart2SVG that embed an OCR-aware decoding head, to achieve accurate axis-scale inference without retraining the core reconstruction model. We identify tighter integration of OCR-based numerical recovery, either as a post-hoc module or as an embedded component of the architecture, as a key direction for future work.

Table 6: Full Definition of the 115 SVG-Specific Tokens

Token	Description	Token	Description
1. Structural Containers & Anchors			
[<|START_OF_SVG|>] / [<|END_OF_SVG|>]	SVG start/end	[<|START_OF_GROUP|>] / [<|END_OF_GROUP|>]	Group (<g>) start/end
[<|clipPath|>] / [<|/clipPath|>]	Clipping path start/end	[<|defs|>] / [<|/defs|>]	Reusable elements start/end
[<|linearGradient|>] / [<|/linearGradient|>]	Linear gradient start/end	[<|radialGradient|>] / [<|/radialGradient|>]	Radial gradient start/end
[<|stop|>] / [<|/stop|>]	Gradient stop start/end	[<|filter|>] / [<|/filter|>]	Filter element start/end
[<|feOffset|>] / [<|/feOffset|>]	feOffset start/end	[<|feColorMatrix|>] / [<|/feColorMatrix|>]	feColorMatrix start/end
[<|feGaussianBlur|>] / [<|/feGaussianBlur|>]	feGaussianBlur start/end	[<|feComposite|>] / [<|/feComposite|>]	feComposite start/end
2. Graphical Primitives			
[<|path|>] / [<|/path|>]	Path start/end	[<|circle|>] / [<|/circle|>]	Circle start/end
[<|rect|>] / [<|/rect|>]	Rectangle start/end	[<|ellipse|>] / [<|/ellipse|>]	Ellipse start/end
[<|polygon|>] / [<|/polygon|>]	Polygon start/end	[<|line|>] / [<|/line|>]	Line start/end
[<|polyline|>] / [<|/polyline|>]	Polyline start/end	[<|text|>] / [<|/text|>]	Text start/end
[<|tspan|>] / [<|/tspan|>]	tspan start/end	[<|use|>] / [<|/use|>]	Uses object start/end
3. Geometric Opcodes			
[<|moveto_abs|>] / [<|moveto_rel|>]	Move abs/rel	[<|lineto_abs|>] / [<|lineto_rel|>]	Line abs/rel
[<|horizontal_lineto_abs|>] / [<|horizontal_lineto_rel|>]	H line abs/rel	[<|vertical_lineto_abs|>] / [<|vertical_lineto_rel|>]	V line abs/rel
[<|curveto_abs|>] / [<|curveto_rel|>]	Cubic Bézier abs/rel	[<|smooth_curveto_abs|>] / [<|smooth_curveto_rel|>]	Smooth cubic abs/rel
[<|quadratic_bezier_curve_abs|>] / [<|quadratic_bezier_curve_rel|>]	Quad Bézier abs/rel	[<|smooth_quadratic_bezier_curveto_abs|>] / [<|smooth_quadratic_bezier_curveto_rel|>]	Smooth quad abs/rel
[<|elliptical_arc_abs|>] / [<|elliptical_arc_rel|>]	Arc abs/rel	[<|close_path|>]	Close path
4. Visual Channel & Geometry Attributes			
[<|id=|>]	ID	[<|class=|>]	Class
[<|d=|>]	Path data	[<|style=|>]	Style
[<|fill=|>]	Fill color	[<|fill-opacity=|>]	Fill opacity
[<|stroke=|>]	Stroke color	[<|stroke-width=|>]	Stroke width
[<|stroke-linecap=|>]	Line cap	[<|stroke-opacity=|>]	Stroke opacity
[<|stroke-dasharray=|>]	Dash pattern	[<|opacity=|>]	Global opacity
[<|transform=|>]	Transform	[<|gradientTransform=|>]	Gradient transform
[<|width=|>]	Element width	[<|height=|>]	Element height
[<|x=|>]	Position x	[<|y=|>]	Position y
[<|dx=|>]	Shift x	[<|dy=|>]	Shift y
[<|cx=|>]	Center x	[<|cy=|>]	Center y
[<|rx=|>]	Radius x	[<|ry=|>]	Radius y
[<|r=|>]	Circle radius	[<|points=|>]	Polygon points
[<|x1=|>]	Start x	[<|y1=|>]	Start y
[<|x2=|>]	End x	[<|y2=|>]	End y
[<|offset=|>]	Gradient offset	[<|stop-color=|>]	Stop color
[<|stop-opacity=|>]	Stop opacity	[<|href=|>]	External reference
[<|fr=|>]	Radial focal r	[<|fx=|>]	Radial focal x
[<|fy=|>]	Radial focal y	[<|filter=|>]	Filter attribute
[<|in=|>]	Filter input	[<|stdDeviation=|>]	Gaussian blur std dev
[<|rotate=|>]	Rotation angle	[<|font-size=|>]	Font size
[<|font-style=|>]	Font style	[<|font-family=|>]	Font family
[<|font-weight=|>]	Font weight	[<|text-anchor=|>]	Text anchor
[<|text-content=|>]	Text content	[<|dominant-baseline=|>]	Dominant baseline
[<|viewBox=|>]	ViewBox	[<|clip-path=|>]	Clip path attr
[<|display=|>]	Display	[<|visibility=|>]	Visibility

Fig. 8: Qualitative comparison of SVG code structure across different methods on a grouped bar chart. Chart2SVG produces a clear hierarchical structure and uniform color and coordinate representations, enabling direct programmatic editing. Adobe Illustrator Image Trace generates a flat list of anonymous <path> elements with opaque Bézier sequences and no semantic structure. GPT-4o produces partially readable code but lacks proper grouping and contains data overlaps. StarVector outputs path-based geometry with numerical redundancy and no semantic labels.