BABEL: VISUAL LANGUAGE EMERGENCE

Anonymous authors
Paper under double-blind review

ABSTRACT

Can a visual language emerge from scratch rather than being designed in advance? Visual representations today are built on fixed, human-designed languages, from SVG to plotting libraries to domain-specific languages. The choice of language determines the available abstractions and the size of the search space, yet the language itself is rarely questioned. We introduce Babel, a bi-level framework in which an outer loop evolves primitive operations from expressiveness gaps while an inner loop generates visualizations with the current language. Starting from only four geometric primitives, Babel proposes, validates, and refines primitives over a scheduled stream of tasks. As gaps close, valid updates decline and growth saturates only when one registry is shared across tasks. Crucially, what emerges are mechanisms rather than policies: primitives compress repeated execution without taking task-specific decisions away from the caller. On a benchmark spanning six scientific visualization domains, the emerged language expresses the same figures in 4.06× fewer calls than its seed basis and 1.67× fewer than a hand-crafted DSL, at quality indistinguishable from both and from free-form Python. Its primitives also transfer across domains at no cost in quality, suggesting that what emerges is a general drawing language rather than domain-specific shortcuts. These results show that the representation space itself can be optimized during task solving, with reusable mechanism-level abstractions emerging from the task distribution.

1 INTRODUCTION

When producing a visualization, the first decision is about what language to draw in: SVG paths, matplotlib calls, TikZ commands, or some hand-crafted domain-specific language (DSL) Belouadi et al. (2024;); Rodriguez et al. (2025); Wei et al. (2025). This choice is typically made once and then frozen for the lifetime of the system. A language determines which abstractions are available, how large the search space is, and what kinds of errors are possible Barke et al. (2024); Grand et al. (2024); Hu et al. (2024). The same task can be compact in a well-matched language yet require a long sequence of low-level operations in a poorly matched one Rodriguez et al. (2026). Despite this, the language itself is almost never treated as a variable to be optimized; it is treated as a given.

When framed in terms of program synthesis, the choice of a DSL is equivalent to the choice of an inductive bias Ye et al. (2026); Macfarlane and Bonnet (2026); Barke et al. (2024). A DSL defines the space of programs a model can write, determining which solutions are short and easy to find, which are long and hard to reach, and which cannot be expressed at all Krogmeier and Madhusudan (2025); Barke et al. (2024). Too narrow a language forces the model to approximate high-level concepts through chains of low-level operations Grayeli et al. (2024). Too wide a language, such as general-purpose Python, offers no structural guidance and leaves the model to search a vast undifferentiated space Liu et al. (2025). The ideal language would make the program space isomorphic to the problem structure, but what counts as a good isomorphism depends on the task distribution. No single hand-designed language can anticipate every task distribution it will eventually be asked to face Liu et al. (2024).

Historically, the visual languages that endure were not designed in a single act of engineering Zhang et al. (2026); Hayatpur et al. (2024). Feynman diagrams began as informal sketches, evolved through community use, and only later solidified into standard notation Mastrolorito et al. (2025). Circuit schematics and chemical structural formulas followed similar trajectories: practitioners invented symbols to express recurring concepts, others adopted and refined the useful ones, and the language stabilized once it could cover the domain Shi et al. (2026). In each case the driving force was the same: an expressiveness gap where existing notation fell short triggered the invention of new notation,

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

and growth continued until the gaps closed Ashery et al. (2025); Lipinski et al. (2024); Akkerman et al. (2024); Taniguchi et al. (2024). This suggests a natural question: if a system is given tasks and rendered feedback but no predetermined visual vocabulary, will a structured visual language emerge on its own, and if so what will it look like?

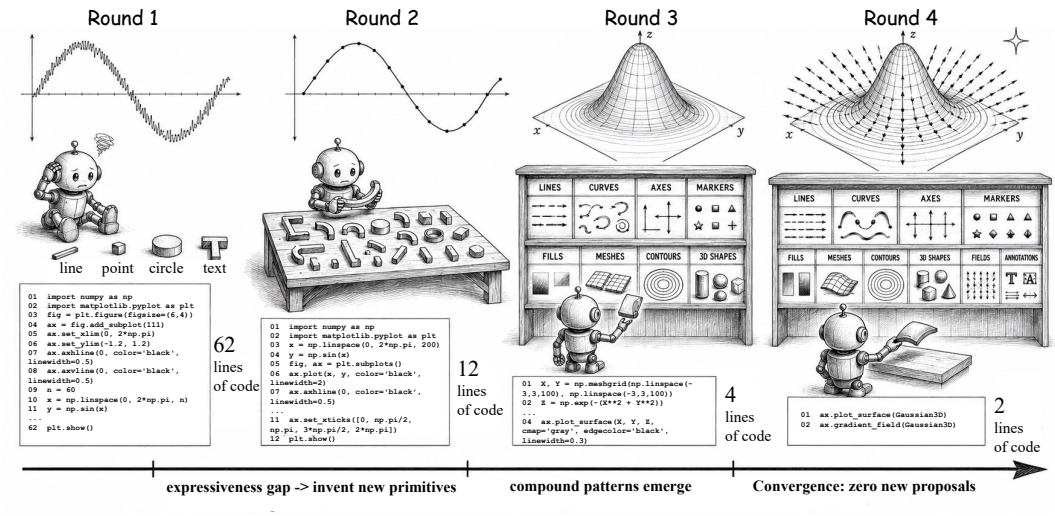


Figure 1: Emergence of a visual language from four seed primitives. Each column is one registry snapshot: the rendered figure, a schematic view of the language state, and a program written in that language. As the registry grows, the same class of scientific figures becomes expressible with progressively fewer calls, here from 62 lines to 12, 4, and 2.

We introduce Babel, a framework that treats visual language design as a bi-level optimization problem. The inner loop solves a batch of visualization tasks with a fixed language snapshot. The outer loop examines the rendered results and updates the shared primitive registry to address expressiveness gaps. Babel starts from only four primitives: line, point, circle, and text. No higher-level functions or abstraction hierarchy are specified in advance. As shown in Figure 1, the system proposes, validates, and registers new primitives as it moves through the scheduled task batches. Existing primitives may also receive small compatible refinements. We study this trajectory itself and ask how a minimal visual vocabulary grows, and what it converges on, under task-driven pressure.

Our experiments reveal a consistent emergence dynamic. When the mismatch between the current primitives and task demands is large, the language expands rapidly. As expressiveness gaps are progressively filled, the rate of valid updates declines. The emerged primitives organize into functional families, progressing from elementary curves to composite patterns to three-dimensional surfaces as new expressiveness gaps arise. On a benchmark spanning six scientific visualization domains, the emerged language expresses each figure in 28.3 commands against 114.9 for its own four-primitive seed basis, 47.3 for a hand-crafted DSL, and 41.9 for free-form Python, a 4.06× reduction over the seed basis at a pass rate of 98.2% against 96.1%, 88.1%, and 95.9%. We measured the pipeline's run-to-run variance directly and report it as a threshold on every comparison, which makes the compression result the load-bearing one and the quality result a non-inferiority claim. Transplanting a vocabulary emerged for one domain onto another costs nothing on average, which suggests what accumulates is a general drawing language rather than a set of domain-tuned conveniences. These findings suggest that the language mediating between a model and its output space need not be fixed in advance, but can itself be optimized while the tasks are being solved.

2 Related Work

Visual Program Generation with Fixed Languages. Structured visual generation typically targets human-fixed formats, including SVG Jain et al. (2023); Wang et al. (2026); Wu et al. (2025); Malashenko et al. (2025); Tian et al. (2025), TikZ Greisinger and Eger (2026); Belouadi et al. (2024), Vega-Lite Satyanarayan et al. (2017), plotting engines like Matplotlib Maddigan and Susnjak (2023); Dibia (2023); Yang et al. (2024) and Viser Yi et al. (2025), or hand-designed DSLs Gupta and

Kembhavi (2023); Subramanian et al. (2023). However, reliance on static, hand-crafted primitive sets prevents these systems from adapting dynamically across domain shifts.

Library Learning and Abstraction Discovery. Library learning discovers reusable components to compress search spaces, pioneered by DreamCoder Ellis et al. (2021), LAPS Ellis et al. (2018), Stitch Bowers et al. (2023), and Babble Cao et al. (2023), and extended via LLM-driven lifelong skill accumulation Grand et al. (2024); Wang et al. (2023). Rather than compressing syntactic AST repetitions, Babel triggers primitive emergence directly from task-induced expressiveness deficits, treating visual vocabulary evolution as an explicit optimization target.

Tool Synthesis and Self-Evolving Agents. While tool-making agents synthesize functions on the fly Cai et al. (2024); Liang et al. (2023); Qian et al. (2023); Wang et al. (2024), they often discard them after single tasks. Recent self-evolving frameworks enable persistent skill retention Lee et al. (2026); Zhang et al. (2026); Chen et al. (2026); Tang et al. (2026); Zhai et al. (2026); Zhang et al. (2026). Babel advances this paradigm by accumulating verified primitives into a shared, composable registry, yielding a coherent visual language rather than fragmented ad-hoc scripts (see Appendix D for extended discussions).

3 METHOD

Babel searches over visual languages rather than operating within a fixed one. A fixed VLM uses the current language to solve a batch of tasks, then examines each rendered result and proposes a change. This alternation exposes abstractions that are missing from the current language.

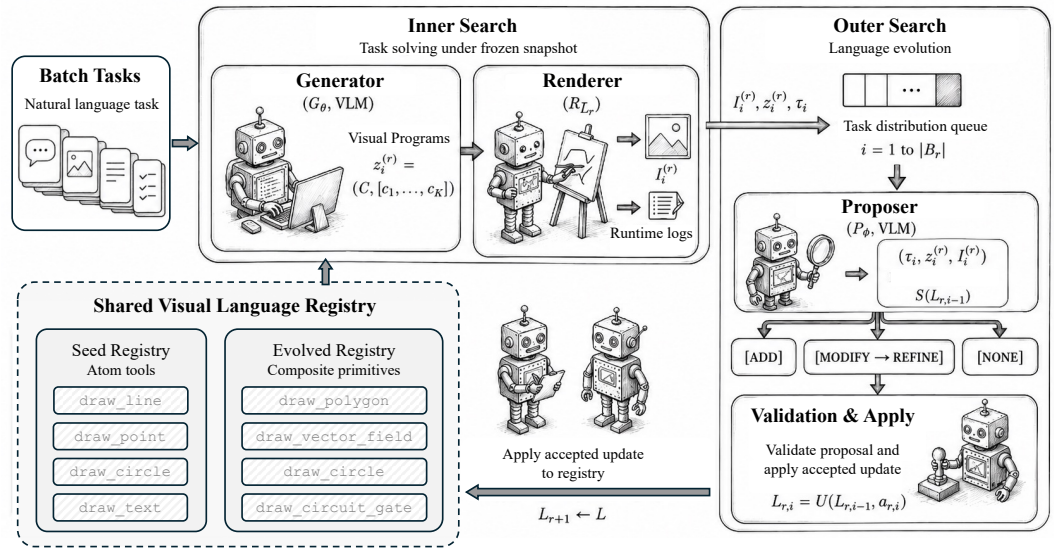


Figure 2: **Overview of Babel.** Babel alternates between two levels of discrete search: an *inner search* solves batch tasks using a frozen language snapshot $\bar{\mathcal{L}}_r$, and an *outer search* where an proposer inspects rendered results, diagnoses missing abstractions, and updates the shared visual language registry $\mathcal{L}$.

3.1 PRELIMINARIES

We consider a distribution of visual generation tasks $\mathcal{D} = \{\tau_i\}$, where each task specifies a desired output through a natural-language instruction. A visual language is a registry $\mathcal{L} = \{p_j\}_{j=1}^{M}$ whose primitives are defined as $p_j = (\text{name}_j, \text{schema}_j, \text{render}_j)$. The language induces a program space $\mathcal{P}(\mathcal{L})$, and the renderer $R_{\mathcal{L}}$ maps each program to an image.

An effective language should support programs that are accurate and concise while avoiding redundant or narrowly specialized primitives. We express this bi-level objective as:

$$\mathcal{L}^* = \arg\min_{\mathcal{L}} \mathbb{E}_{\tau\sim\mathcal{D}} \left[\min_{z\in\mathcal{P}(\mathcal{L})} \mathcal{J}(\tau, R_{\mathcal{L}}(z)) + \alpha|z| \right] + \lambda\Omega(\mathcal{L}), \tag{1}$$

where $\mathcal{J}$ measures instruction and image inconsistency, $|z|$ counts primitive calls, and $\Omega(\mathcal{L})$ denotes a preference for compact, reusable, and compositional abstractions. The language space is discrete, so Babel does not optimize Equation 1 by gradient descent. A fixed generator G_θ approximates the inner search: $\hat{z}_\tau = G_\theta(\tau, S(\mathcal{L}))$, where $S(\mathcal{L})$ serializes the available primitive interfaces.

3.2 BABEL

Discrete realization of the bi-level objective. Equation 1 defines the language-level objective, but its outer search is discrete because changing $\mathcal{L}$ modifies both symbolic interfaces and executable renderer code. Babel therefore approximates the two levels with alternating VLM inference. At each round, the current language is first frozen to generate programs for a batch of tasks. The resulting programs and renderings then provide task-level evidence for discrete edits to the language. Each accepted edit changes the program space available to subsequent generation.

Visual language state. At round r, Babel maintains a primitive registry $\mathcal{L}_r = \{p_j\}_{j=1}^{M_r}$. Each primitive contains a symbolic interface and an executable renderer. A visual program is represented as:

$$z = (C, [c_1, \ldots, c_K]), \qquad c_k = (e_k, f_k, \vartheta_k, m_k), \tag{2}$$

where C specifies the canvas, e_k is a stable element identifier, $f_k \in \mathcal{N}(\mathcal{L}_r)$ selects a registered primitive, ϑ_k contains its typed arguments, and m_k stores optional metadata. We initialize the search from $\mathcal{L}_0 = \{\texttt{draw_line}, \texttt{draw_point}, \texttt{draw_circle}, \texttt{draw_text}\}$. This deliberately low-level basis makes mismatches between the current language and the task distribution observable through long programs, repeated constructions, or unsuccessful renderings.

Round-wise generation. The emergence set is partitioned into scheduled batches $\mathcal{B}_1, \ldots, \mathcal{B}_R$. Let $\mathcal{L}_{r-1}$ denote the registry before round r. Babel first takes a frozen snapshot $\bar{\mathcal{L}}_r = \mathcal{L}_{r-1}$. All tasks in the batch are solved under the same snapshot:

$$z_i^{(r)} = G_\theta\big(\tau_i, S(\bar{\mathcal{L}}_r)\big), \qquad I_i^{(r)} = R_{\bar{\mathcal{L}}_r}\Big(z_i^{(r)}\Big), \quad \tau_i \in \mathcal{B}_r, \tag{3}$$

where $I_i^{(r)}$ denotes the rendered image or rendering failure. Freezing the registry during generation ensures that all task-level evidence within a round is collected under the same visual language.

Language proposal and update. After generation, Babel processes the tasks sequentially. Let $\mathcal{L}_{r,i-1}$ denote the registry before processing task τ_i, with $\mathcal{L}_{r,0} = \bar{\mathcal{L}}_r$. Based on the task, its generated program, rendered output, and current registry, the proposal stage predicts

$$a_{r,i} = P_\theta\Big(\tau_i, z_i^{(r)}, I_i^{(r)}, S(\mathcal{L}_{r,i-1})\Big), \qquad a_{r,i} \in \{\textsc{Add}, \textsc{Modify}, \textsc{None}\}. \tag{4}$$

The three actions have distinct roles:

- ADD introduces a primitive for a visual concept outside the responsibilities of the current registry;
- MODIFY makes a compatible refinement to an existing abstraction;
- NONE retains the current language.

The refiner is instructed to preserve the primitive name, responsibility, and default behavior. Thus P_θ denotes the full proposal stage, including this optional refinement. The registry is then updated by

$$\mathcal{L}_{r,i} = \begin{cases} U(\mathcal{L}_{r,i-1}, a_{r,i}), & \text{if } a_{r,i} \text{ is valid,} \\ \mathcal{L}_{r,i-1}, & \text{otherwise.} \end{cases} \tag{5}$$

Accepted edits become immediately available to later proposals, while the programs and renderings within the round remain those generated from the frozen snapshot $\bar{\mathcal{L}}_r$. Thus, generation approximates the inner search in Equation 1, while proposal and registration implement a discrete outer search over visual languages. Proposal constraints provide a procedural realization of $\Omega(\mathcal{L})$ by favoring compact, reusable, and compositional abstractions.

All domains update the same registry, so abstractions discovered in one domain can be reused by subsequent tasks from any other domain. Detailed proposal rules, refinement procedures, validation checks, and scheduling are provided in Appendix B, where Algorithm 1 also summarizes the complete procedure as pseudocode.

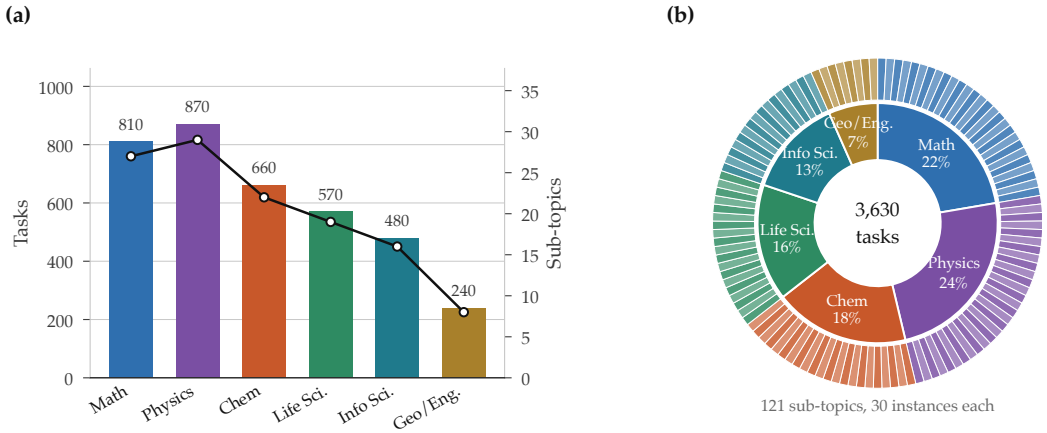


Figure 3: **Benchmark statistical distributions. (a)** Task (bars) and sub-topic (line) counts per discipline. **(b)** Hierarchical sunburst of the 6 disciplines and 121 sub-topics with percentage breakdowns.

4 BABEL BENCHMARK

4.1 DATASET OVERVIEW & SCALE

Our benchmark holds **3,630** visual instances over **6** disciplines and **121** sub-topics (Figure 3). Every sub-topic carries exactly 30 instances, split evenly across three difficulty tiers, so the totals are 1,211 Easy, 1,210 Medium, and 1,209 Hard. Domains are uneven by design because they differ in how many distinct drawing conventions they use; Appendix A details all 121 sub-topics along with with representative visual exemplars across the 6 disciplines shown in Figure 5.

4.2 DATA GENERATION

Our **dual-paradigm architecture** (Figure 4) couples an *LLM-driven pipeline* for conceptual math/physics tasks with a *domain-engine pipeline* for multi-discipline precision.

LLM-driven code synthesis. An LLM generates `Matplotlib` scripts refined via execution tracebacks in an isolated `Agg` sandbox, ensuring 100% validity with human exception handling. A multimodal reviewer pairs figures with source code to generate bilingual prompts and complexity hints.

Domain-engine procedural synthesis. To eliminate hallucinations, domain content spans all six disciplines via specialized engines (e.g., `RDKit`, `schemdraw`). Each sub-topic yields 30 instances across three difficulty tiers using deterministic seeds $\boldsymbol{\mathcal{S}} = \mathbf{base_seed} + (\mathbf{i_{subclass}} \cdot \mathbf{N_{diff}} \cdot \mathbf{N_{slot}}) + (\mathbf{diff_tier} \cdot \mathbf{N_{slot}}) + \mathbf{slot_idx}$(where N_{slot}, N_{diff}, i_{subclass}, and diff_tier denote tier capacity, difficulty count, subclass index, and tier index), ensuring exact index-based reproducibility. A vision-language model appends bilingual prompts with $\leq$ 40-character complexity hints.

4.3 EMERGENCE AND EVALUATION SPLITS

Each of the 121 sub-topics splits its 30 instances equally across difficulty tiers into disjoint *emergence* and *held-out* sets (1,815 tasks and 5 Easy, 5 Medium, 5 Hard each). Search accesses only the emergence split until registry freezing. We split by instance rather than sub-topic to test if emergence generalizes across new instances of a distribution without pre-assuming topic specificity. Cross-topic transfer is evaluated separately in §5.3 via registry sharing scopes.

5 EXPERIMENT

Five questions organize what follows. Does an emerged language express scientific figures more efficiently than languages designed in advance, at equal quality (§5.2), including two authored outside this project (Appendix B.10)? What makes accumulated primitives behave like a language rather than a pile of tools (§5.3)? How much evidence does the outer search need (§5.3)? What kind of abstraction does it converge on (§5.5)? And does any of this hold outside the one generator we ran (§5.4)?

Three negative results sit on equal footing with those. §5.1 measures the pipeline's noise floor, which retracts two effects we had read as findings; Appendix B.12 reports that the obvious fix for the defect

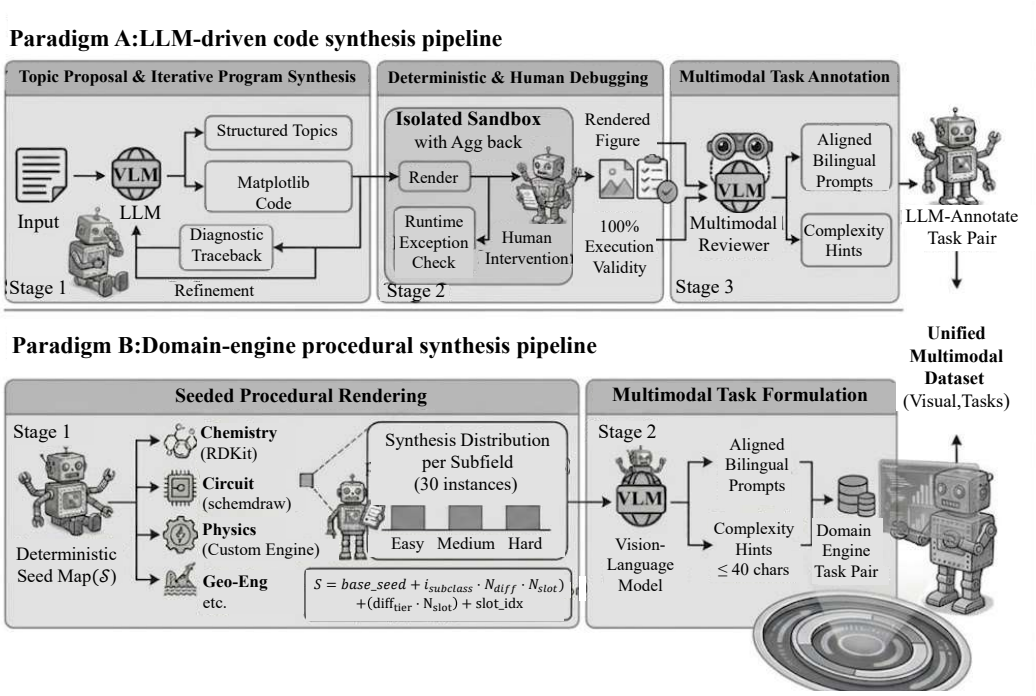


Figure 4: **Dual-paradigm generation architecture. Top (Paradigm A):** Code-driven instance synthesis for math and physics via deterministic sandboxing and task annotation. **Bottom (Paradigm B):** Domain-engine procedural generation across six disciplines with reproducible seed mapping.

we identify makes the gate weaker, not stronger; and Appendix B.17 adds two further null results, one on recovering a program from a rendered image and one on editing a figure locally, the second of which refuted our own hypothesis and exposed a defect in our patch engine.

5.1 Evaluation Protocol

We freeze the registry after the final emergence batch and evaluate on the held-out half of all six domains, with no primitive proposed, refined, or repaired in response to a held-out task. Quality is scored by a separate VLM call seeing the instruction, image, and program, returning 0–100 with a pass at 70; every arm is re-scored by a second VLM from an unrelated family. Efficiency is the command count, the operational form of $|z|$ in Equation 1. Four reference languages run under one generator, judge, and prompt contract, so the language is the only variable: **Seed-4**, the initial four-primitive registry; **Human-28**, a DSL hand-designed by human experts before any emergence run; unrestricted **Python**; and **Babel**, the frozen emerged registry.

What this pipeline can resolve. Re-judging fixed programs gives $\sigma = 0.599$ percentage points (**pp**), while regenerating then re-judging gives $\sigma = 1.257$pp. We adopt **1.43pp** as the working minimum detectable effect threshold (with resolution lying in $[1.43, 4.9]$pp under clustering). While the compression results clear this threshold by an order of magnitude, findings in §5.3 and Appendix B.16 fall below it. Appendix B.15 gives full details on the metrics, judge rubrics, statistical power derivations, clustered variance analysis, and why pixel-based metrics (e.g., SSIM, CLIP) fail on this benchmark.

5.2 An Emerged Language Is More Compact at Equal Quality

Table 1 summarizes the primary evaluation across 1,815 held-out scientific tasks. The emerged language (**Babel**) expresses each figure in 28.3 commands against 114.9 for the seed basis, 47.3 for the expert-designed DSL, and 41.9 for unrestricted Python, while maintaining an unmatched 98.24% pass rate. Per-domain breakdowns and the same arms plotted as a quality-effort trade-off are in Appendix B.14 (Table 10, Figure 10).

Efficiency via expressive abstraction. Compression over the seed basis is $4.06\times$ (CI $[3.69, 4.47]$) across all domains, and $1.67\times$ (CI $[1.51, 1.87]$) over Human-28. Because generator, prompts,

Table 1: **Aggregate performance and paired contrasts against Babel (isolated). Pass$_2$** is a second judge from an unrelated model family; brackets denote cluster-bootstrap 95% CIs over 121 subtopics. **ΔPass** measures differences against the 1.43 pp resolution threshold.

Language	Pass↑	Pass$_2$↑	Cmd↓	Prim.	ΔPass, judge 1 (pp)	ΔPass, judge 2 (pp)	Cmd ratio
Seed-4	96.14	94.49	114.9	2.78	+2.09 [0.33, 3.69]	+2.75 [1.10, 4.46]	4.06×
Human-28	88.10	85.73	47.3	4.54	+10.14 [6.83, 13.66]	+11.52 [7.88, 15.37]	1.67×
Python	95.92	94.10	41.9	4.28	+2.31 [0.00, 4.79]	+3.14 [0.77, 5.90]	0.7–1.5×†
Babel (shared)	96.80	96.36	31.2	4.34	+1.43 [−0.50, 3.14]	+0.88 [−0.72, 2.37]	1.10×
Babel (isolated)	**98.24**	**97.25**	**28.3**	4.14	—	—	—

and decoding are identical across arms, this difference stems entirely from available abstractions. Crucially, compression does not require broader vocabularies (4.1 distinct primitives/figure for Babel vs. 4.3 for Python and 4.5 for Human-28), proving that individual emerged primitives are fundamentally more expressive.

The cost of premature human abstraction. Human-28 trails even the 4-primitive baseline by 8.0 pp, with deficits concentrated in domains targeted most heavily (Math: 77.3% vs. 97.8%; Chem: 83.9% vs. 96.4%). Over-specified primitives (e.g., `draw_molecule`) dictate geometric assumptions that block non-standard task execution without offering fallbacks. In contrast, Seed-4 succeeds verbosely (96.14%), demonstrating that low-level expressiveness is rarely the correctness bottleneck, though it incurs verbosity and fragility that Babel effectively abstracts away.

Comparison to external DSLs. On 121 representative sub-topics, Babel maintains equivalent quality to TikZ/PGF and raw SVG while requiring 2.19× fewer commands than SVG. TikZ advantages remain confined to tasks leveraging `pgfplots` axis automations (Appendix B.10).

5.3 Sharing, and How Much Evidence the Search Needs

Saturation is a consequence of sharing, not of the task schedule. With all 121 subtopics writing to one registry, admissions fall +4 to +1 as it grows 4 → 8 → 11 → 13 → 14; isolation yields no decay (+13, +10, +13, +15). The decay ratio moves monotonically with scope — 0.25, 1.00, 1.15 at 1, 6, and 121 subtopics per registry. Isolation does not prevent local saturation; it prevents the accumulation of it. Appendix B.6 details the schedule confound analysis and within-registry admission timelines.

Sharing forces reuse, and reuse is what a language means. The 10 primitives emerged in shared mode fill 48.6% of all primitive slots across the 1,815 held-out figures. `draw_filled_regions`, proposed on a UV-Vis spectrum in Chemistry, is used in 1,165 figures across all six domains; `draw_polyline_series`, from an infrared spectrum, in 996; `draw_node_link_diagram`, from a binary search tree in Information Science, in 587. None encodes its origin domain: spectra, seafloor profiles, and reaction-energy curves are one drawing act, found once.

Isolation instead reinvents the same abstractions. At the matched budget the isolated arm's 27 names reduce to 21 concept stems, 3 invented more than once; at N=15 the 121 names reduce to 99 stems with 12 duplicated, including five arrow primitives, four polyline variants, and three ellipse variants. Shared mode faces the same 121 subtopics and duplicates no stem.

Table 2: **Sharing controls whether growth saturates.** Budget is one task per subtopic, so rows differ only in registry scope. Q1–Q4 count primitives *new to their own registry* and Q4/Q1 is the decay ratio; multi-registry rows are unweighted means.

Scope	Reg.	New per quartile				Q4/Q1	Em.	$\overline{\lvert\mathcal{L}\rvert}$	Pass↑
		Q1	Q2	Q3	Q4				
Shared	1	+4	+3	+2	+1	**0.25**	10.0	14.0	96.80
Per-domain	6	+7	+8	+12	+7	1.00	5.3	9.3	95.76
Isolated	121	+13	+10	+13	+15	1.15	1.2	5.2	97.69

Table 3: **Emergence budget sweep.** N is emergence tasks per subtopic. Every pairwise pass-rate difference is below the 1.43pp threshold, so read that column as flat; **Cmd** spans 1.64×. ‡These three processed their schedule twice, so *Em.* is not monotone in N.

Budget	Tasks seen	Pass (%)↑	Pass_2 (%)↑	Score↑	Cmd↓	Union	Em.
N=1‡	121	97.69	96.86	94.7	46.3	61	1.2
N=5‡	605	98.07	97.47	94.7	31.9	98	2.1
N=10‡	1,210	97.08	95.54	94.4	29.2	121	2.5
N=15	1,815	**98.24**	**97.25**	**94.9**	**28.3**	118	2.4

The two modes trade compactness against local fit. In Table 2 isolation's 97.69% against sharing's 96.80% is a 0.88pp difference below the 1.43pp threshold. Sharing returns a 14-primitive language that saturates, transfers across domains (§5.4), and fits on one page, against 121 private vocabularies averaging 1.2 emerged primitives each. We treat isolation as an upper bound on per-task specialization, not the configuration we advocate.

Table 3 varies the emergence budget from one to fifteen tasks per subtopic. A single Easy task per subtopic (121 tasks, 6.7% of the emergence pool) already yields 97.69%. Fifteen-fold more evidence adds +0.55pp, and every pairwise difference in that column falls below the 1.43pp threshold.

What it buys instead is compression, monotonically: 46.3 → 31.9 → 29.2 → 28.3 commands per figure, diminishing after N=5. That 1.64× span is clear of the noise floor, so the metrics come apart: one rendering failure reveals that a class of figures needs an abstraction, but discovering the *right* parameterization requires seeing the class vary. Appendix B.6 details sample requirements across objective levels and the retracted N=10 variance analysis.

5.4 Emerged Vocabularies Generalize Across Domains and Models

Table 4 evaluates cross-domain transfer by loading each domain's emerged vocabulary against every domain's held-out tasks across all 36 ordered pairs; Figure 6 in Appendix B.9 renders the same matrix as a heatmap.

Table 4: **Cross-domain transfer, 36 cells × 60 tasks (2,160 total).** Rows load one domain's vocabulary, columns supply the tasks, and values are pass rate (%). Each column's diagonal is its in-domain reference; Chemistry is the sole exception (§5.4).

Source vocabulary	Em.	InfoSci	Chem	Geo	Math	Phys	Life	Mean
Info. Science	2	*88.3*	98.3	96.7	95.0	96.7	91.7	94.4
Chemistry	5	46.7	*90.0*	33.3	66.7	65.0	38.3	56.7
Geo/Eng.	3	98.3	98.3	*100.0*	93.3	96.7	91.7	**96.4**
Mathematics	9	98.3	95.0	96.7	*96.7*	95.0	93.3	95.8
Physics	5	98.3	**100.0**	95.0	93.3	*98.3*	90.0	95.8
Life Science	6	96.7	98.3	100.0	98.3	98.3	*93.3*	**97.5**
Column mean		87.8	96.7	86.9	90.6	91.7	83.1	89.4

Transfer is close to free, and often better than staying in domain. Excluding Chemistry as a source, the 25 remaining transplant cells average 96.1% against 94.3% for the in-domain reference of the same columns: a vocabulary emerged elsewhere does slightly better on a domain than the one emerged for it. The Chemistry column reads 90.0% under its own vocabulary and 95.0–100% under all five others, and the two-primitive Information Science registry reaches 98.3% on Chemistry and 96.7% on Geoscience.

Registry size does not predict transfer. The two best sources are among the smallest, Geoscience at 3 primitives scoring 96.4% and Life Science at 6 scoring 97.5%, while the largest, Mathematics at 9, scores 95.8%. Shape matters, not count, consistent with the shared arm reaching 96.80% benchmark-wide on 10 primitives.

The Chemistry row is the informative failure. As a source vocabulary Chemistry averages 56.7%, dropping to 33.3% on Geoscience and 38.3% on Life Science while performing normally as a

target. All five primitives are curve and surface abstractions built for spectra and molecular renderings (`draw_cubic_bezier_path`, `draw_sphere_series`, `draw_cylinder_series`) with nothing for a labelled region or a node-link structure. Appendix B.9 analyzes the generator's over-selection failure mode and sample diversity retractions.

Swapping the generator under a frozen registry on a 149-task subset (Table 6, detectable difference 4.98 pp) leaves `gemini-3.7-flash` at top performance (95.30% vs. 98.66%, within noise), while `claude-sonnet-4-6` (65.77%) and `gpt-4o` (16.78%) drop from under-generation ($\sim$12 commands). We restrict our claims to reasoning-tier generators and `gemini-3.7-flash` (Appendix B.9).

5.5 Mechanisms, Not Policies: What Emerges and What It Buys

The converged primitives exhibit a structural trait absent in both seed and hand-designed DSLs: 9 of 10 shared primitives consume lists of per-element records (explicit coordinates, colors, labels, styles) for uniform stroking, averaging 1.8 schema fields with zero Boolean flags. This pattern holds across 26 of 34 per-domain primitives (5.4 fields on average).

Emerged primitives thus provide drawing *mechanisms* without layout decisions: `draw_node_link_diagram` consumes endpoints and node positions without computing layouts, and `draw_filled_regions` consumes boundaries without placing peaks. Leaving all geometry and styling to the generator enables a single primitive to serve binary search trees, protein networks, and process diagrams alike, as reflected in mechanism names (`draw_polyline_series`, `draw_filled_regions`) rather than domain-specific ones (`draw_nmr_spectrum`).

This is the axis on which Human-28 fails, its 8.0pp deficit against the seed basis concentrating where its internalization bites hardest (§5.2). The useful question for abstraction design is not how much a primitive knows about its subject, but how little it decides for the caller.

Stating the constraint is worth 4.90pp, and it buys quality rather than compression. Holding the 1,815 tasks, generator, and prompt contract fixed and varying only the proposer's design block gives 93.33% against 98.24%, a $+4.90$pp gain ($+4.79$pp under the second judge), clearing even the conservative 4.9pp resolution reading. Commands per figure move by 0.35 and are unchanged, so the constraint does not buy compression. What it buys is a narrower vocabulary that works better: emerged primitives per registry fall from 3.96 to 2.43 and their union across 121 registries from 252 to 118 (Table 8). Fewer and narrower abstractions, at a higher pass rate, is the predicted signature of forbidding a primitive to decide anything.

Mechanism-oriented interface steering. What the comparison establishes is that telling the proposer *how* a primitive may expose its interface changes what the search finds by more than any budget or scheduling choice we varied. Appendix B.11 details the block differences and statistical tests, while Appendix B.12 examines why code-level admission gates fail to enforce these constraints automatically.

6 Conclusion

Treating the visual language as an optimization target yields a compact vocabulary requiring 4.06$\times$ fewer commands than the 4-seed basis and 1.67$\times$ fewer than a hand-designed DSL over 1,815 held-out figures, at non-inferior quality to Python ($\sigma = 1.257$pp). Three core principles govern this emergence: (1) *saturation requires sharing*, dropping quartile additions from 11 to 1; (2) *emerged vocabularies are domain-general*; and (3) *abstractions succeed as drawing mechanisms rather than subject-matter policies*.

AI Use Statement

In this work, we used generative AI tools for several research tasks for which disclosure is required. Generative models are also the object of study rather than only an authoring aid: the generator, proposer, refiner, and judge described in §3 are all frozen vision-language models queried through their public APIs, and the specific models and decoding settings used for every reported arm are listed in Appendix B, with the verbatim prompts in Appendix C. Emerged primitives are therefore AI-generated artifacts by construction: each primitive's symbolic interface and executable renderer were proposed by a model and then admitted only after passing the automated validation checks of

Appendix B.4. The authors reviewed the resulting registries, re-read every emerged primitive reported in the paper, and take responsibility for the claims made about them.

Generative AI tools were also used to assist with the implementation of the emergence pipeline, the rendering harness, the evaluation and judging pipeline, and the figure-plotting and data-analysis code. All AI-assisted code was read, tested, and corrected by the authors before being used to produce reported numbers; the figure-building script additionally audits its own output, as described in Appendix B.15. We used generative AI for language polishing of the manuscript, for improving clarity and readability, for assisting with literature search and identifying related work, and for formatting LaTeX and tables. Every such passage was reviewed and revised by the authors.

We did not use generative AI to formulate the research hypothesis, to design the method or the experimental protocol, to decide which results to report, or to interpret experimental outcomes; in particular the negative and retracted results (§5.1, Appendix B.17, Appendix B.12) reflect author judgment about what the measurements do and do not support. Tasks involving mathematical claims, formal proofs, or assistance with proof construction are not applicable to this work. We take responsibility for the final content of this work, including its methodology, experiments, results, claims, text, and code, including any content produced or modified with the aid of generative AI.

Ethics Statement

This work studies how a visual drawing language can emerge from task feedback. All experiments are conducted on programmatically generated scientific figures and involve no human subjects, no user study, and no personally identifying, sensitive, or human-annotated data. The benchmark instances are synthesized by the two pipelines of §4.2: figures are produced either by executing model-written `Matplotlib` code in an isolated sandbox or by deterministic domain engines seeded as described there, so every instance is reproducible from its seed and carries no third-party content. We use open-source scientific libraries in accordance with their licenses and rely on commercial model APIs under their terms of use.

Two risks deserve explicit mention. First, the emerged primitives are model-proposed executable code admitted by automated checks; we execute all generated programs and all candidate renderers inside a sandboxed, non-interactive rendering backend with no network or filesystem access beyond the figure output, and we recommend the same containment for anyone reusing the pipeline, since admitting model-written code into a shared registry is otherwise a code-execution risk. Second, our admission gate certifies that a primitive is well-formed, not that it is correct or better: Appendix B.12 documents a case where an admitted primitive that computes its own field lines drops a sub-topic from 100% to 20%. A vocabulary grown this way should not be treated as validated for settings where an incorrect figure carries real cost, such as instructional material, clinical or safety documentation, or published scientific results, without human review of the primitives in use.

Our evaluation also has limited scope. It covers six scientific disciplines, 121 sub-topics, one judge family with a second judge from an unrelated family as a check, and predominantly one generator tier, so the reported numbers should not be assumed to transfer to other visual domains, languages, notational traditions, or weaker generators; Table 6 shows quality collapsing on lower-tier generators under the same frozen registry. Section E states the remaining limitations in full, including the single-run-per-configuration caveat and the prompt-language contamination we found in the bilingual split.

Reproducibility Statement

We provide detailed information to support the reproducibility of the benchmark, the emergence procedure, and the evaluation. Section 3 specifies the method, with the visual-program representation in Equation 2, the round-wise generation and update rules in Equations 3–5, and the complete procedure as Algorithm 1; Appendix B gives the primitive registry format, the proposal and refinement rules, the validation checks applied before admission, the emergence schedule and sharing scopes, and the generator and judge configurations. Section 4.2 describes benchmark construction and gives the deterministic seed mapping that makes every instance recoverable by index, and Appendix A lists all 121 sub-topics; Section 4.3 defines the emergence and held-out splits, which are disjoint and fixed before any evaluation.

Section 5.1 defines the evaluation protocol, the four reference languages, and the measured noise floor that every comparison in the paper is read against; Appendix B.15 gives the metric definitions, judge rubrics, statistical power derivations, and clustered variance analysis in full, together with the judge-agreement study in Appendix B.8. The exact prompts for the generator, judge, proposer, and refiner are reproduced verbatim in Appendix C, since results here depend on prompt contracts as much as on code. Per-domain breakdowns are in Appendix B.14, the transfer protocol in Appendix B.9, the external-DSL comparison in Appendix B.10, and the proposer design-block comparison in Appendix B.11. Results that fall below the resolution threshold are marked as such in the text and the corresponding retractions are reported in Appendix B.16 and Appendix B.17 rather than omitted. Because the outer search queries commercial model APIs, exact reruns are subject to provider-side model drift; we therefore report the run-to-run variance we measured and treat the compression results, which clear that floor by an order of magnitude, as the load-bearing claims. Source code for the emergence pipeline, the benchmark generators, and the evaluation harness, along with the frozen registries needed to reproduce the reported arms, are included in the supplementary material.

REFERENCES

[1] Daniel Akkerman, Phong Le, and Raquel G. Alhama. The emergence of compositional languages in multi-entity referential games: from image to graph representations. In *Conference on Empirical Methods in Natural Language Processing*, 2024. URL https://api.semanticscholar.org/CorpusID:273901297.

[2] Ariel Flint Ashery, Luca Maria Aiello, and Andrea Baronchelli. Emergent social conventions and collective bias in llm populations. *Science Advances*, 11(20):eadu9368, 2025.

[3] Shraddha Barke, Emmanuel Anaya Gonzalez, Saketh Ram Kasibatla, Taylor Berg-Kirkpatrick, and Nadia Polikarpova. Hysynth: Context-free llm approximation for guiding program synthesis. *Advances in Neural Information Processing Systems*, 37:15612–15645, 2024.

[4] Jonas Belouadi, Anne Lauscher, and Steffen Eger. Automatikz: Text-guided synthesis of scientific vector graphics with tikz. In *International Conference on Learning Representations*, volume 2024, pages 55917–55943, 2024.

[5] Jonas Belouadi, Simone P Ponzetto, and Steffen Eger. Detikzify: Synthesizing graphics programs for scientific figures and sketches with tikz. *Advances in Neural Information Processing Systems*, 37:85074–85108, 2024.

[6] Jonas Belouadi, Simone Paolo Ponzetto, and Steffen Eger. Detikzify: Synthesizing graphics programs for scientific figures and sketches with tikz. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 85074–85108. Curran Associates, Inc., 2024. doi: 10.52202/079017-2701. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/9a8d52eb05eb7b13f54b3d9eada667b7-Paper-Conference.pdf.

[7] Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. Top-down synthesis for library learning. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi: 10.1145/3571234. URL https://doi.org/10.1145/3571234.

[8] Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun, editors, *International Conference on Learning Representations*, volume 2024, pages 54067–54089, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/ed91353f700d113e5d848c7e04a858b0-Paper-Conference.pdf.

[9] David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. babble: Learning better abstractions with e-graphs and anti-unification. *Proc. ACM Program. Lang.*, 7 (POPL), January 2023. doi: 10.1145/3571207. URL https://doi.org/10.1145/3571207.

[10] Tingyang Chen, Shuo Lu, Kang Zhao, Weicheng Meng, Hanlin Teng, Tianhao Li, Chao Li, Xule Liu, Jian Liang, Zhizhong Zhang, Yuan Xie, Heng Qu, Kun Shao, and Jian Luan. Harnessx: A composable, adaptive, and evolvable agent harness foundry, 2026. URL https://arxiv.org/abs/2606.14249.

[11] Victor Dibia. LIDA: A tool for automatic generation of grammar-agnostic visualizations and infographics using large language models. In Danushka Bollegala, Ruihong Huang, and Alan Ritter, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 113–126, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-demo.11. URL https://aclanthology.org/2023.acl-demo.11/.

[12] Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. Learning libraries of subroutines for neurally–guided bayesian program induction. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper_files/paper/2018/file/7aa685b3b1dc1d6780bf36f7340078c9-Paper.pdf.

[13] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 835–850, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454080. URL https://doi.org/10.1145/3453483.3454080.

[14] Gabriel Grand, Lio Wong, Maddy Bowers, Theo X Olausson, Muxin Liu, Joshua B Tenenbaum, and Jacob Andreas. Lilo: Learning interpretable libraries by compressing and documenting code. In *International Conference on Learning Representations*, volume 2024, pages 30399–30446, 2024.

[15] Arya Grayeli, Atharva Sehgal, Omar Costilla-Reyes, Miles Cranmer, and Swarat Chaudhuri. Symbolic regression with a learned concept library. *Advances in Neural Information Processing Systems*, 37: 44678–44709, 2024.

[16] Christian Greisinger and Steffen Eger. Tikzilla: Scaling text-to-tikz with high-quality data and reinforcement learning, 2026. URL https://arxiv.org/abs/2603.03072.

[17] Tanmay Gupta and Aniruddha Kembhavi. Visual programming: Compositional visual reasoning without training. In *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 14953–14962, 2023. doi: 10.1109/CVPR52729.2023.01436.

[18] Devamardeep Hayatpur, Brian Hempel, Kathy Chen, William Duan, Philip Guo, and Haijun Xia. Taking ascii drawings seriously: How programmers diagram code. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, pages 1–16, 2024.

[19] Ziniu Hu, Ahmet Iscen, Aashi Jain, Thomas Kipf, Yisong Yue, David A Ross, Cordelia Schmid, and Alireza Fathi. Scenecraft: An llm agent for synthesizing 3d scenes as blender code. In *ICML*, pages 19252–19282, 2024.

[20] Ajay Jain, Amber Xie, and Pieter Abbeel. Vectorfusion: Text-to-svg by abstracting pixel-based diffusion models. In *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1911–1920, 2023. doi: 10.1109/CVPR52729.2023.00190.

[21] Paul Krogmeier and P Madhusudan. Synthesizing dsls for few-shot learning. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA2):642–669, 2025.

[22] Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses, 2026. URL https://arxiv.org/abs/2603.28052.

[23] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9493–9500, 2023. doi: 10.1109/ICRA48891.2023.10160591.

[24] Olaf Lipinski, Adam J. Sobey, Federico Cerutti, and Timothy J. Norman. Speaking your language: Spatial relationships in interpretable emergent communication. *ArXiv*, abs/2406.07277, 2024. URL https://api.semanticscholar.org/CorpusID:270379805.

[25] Max Liu, Chan-Hung Yu, Wei-Hsu Lee, Cheng-Wei Hung, Yen-Chun Chen, and Shao-Hua Sun. Synthesizing programmatic reinforcement learning policies with large language model guided search. In *International Conference on Learning Representations*, volume 2025, pages 88845–88900, 2025.

[26] Weiyu Liu, Geng Chen, Joy Hsu, Jiayuan Mao, and Jiajun Wu. Learning planning abstractions from language. In *International Conference on Learning Representations*, volume 2024, pages 3578–3596, 2024.

[27] Matthew Macfarlane and Clem Bonnet. Searching latent program spaces. *Advances in Neural Information Processing Systems*, 38:103463–103520, 2026.

[28] Paula Maddigan and Teo Susnjak. Chat2vis: Generating data visualizations via natural language using chatgpt, codex and gpt-3 large language models. *IEEE Access*, 11:45181–45193, 2023. doi: 10.1109/ACCESS.2023.3274199.

[29] Boris Malashenko, Ivan Jarsky, and Valeria Efimova. Leveraging large language models for scalable vector graphics processing: A review, 2025. URL https://arxiv.org/abs/2503.04983.

[30] Fabrizio Mastrolorito, Fulvio Ciriaco, Maria Vittoria Togo, Nicola Gambacorta, Daniela Trisciuzzi, Cosimo Damiano Altomare, Nicola Amoroso, Francesca Grisoni, and Orazio Nicolotti. fragsmiles as a chemical string notation for advanced fragment and chirality representation. *Communications Chemistry*, 8(1):26, 2025.

[31] Cheng Qian, Chi Han, Yi Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. CREATOR: Tool creation for disentangling abstract and concrete reasoning of large language models. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 6922–6939, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.462. URL https://aclanthology.org/2023.findings-emnlp.462/.

[32] Juan Rodriguez, Haotian Zhang, Abhay Puri, Rishav Pramanik, Aarash Feizi, Pascal Wichmann, Arnab Mondal, Mohammad R Samsami, Rabiul Awal, Perouz Taslakian, et al. Rendering-aware reinforcement learning for vector graphics generation. *Advances in Neural Information Processing Systems*, 38:60496–60534, 2026.

[33] Juan A Rodriguez, Abhay Puri, Shubham Agarwal, Issam H Laradji, Pau Rodriguez, Sai Rajeswar, David Vazquez, Christopher Pal, and Marco Pedersoli. Starvector: Generating scalable vector graphics code from images and text. In *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 16175–16186. IEEE, 2025.

[34] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. Vega-lite: A grammar of interactive graphics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):341–350, 2017. doi: 10.1109/TVCG.2016.2599030.

[35] Xinyu Shi, Li-Yi Wei, Nanxuan Zhao, Jian Zhao, and Rubaiat Habib Kazi. Notational animating: An interactive approach to creating and editing animation keyframes. In *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems*, pages 1–27, 2026.

[36] Sanjay Subramanian, Medhini Narasimhan, Kushal Khangaonkar, Kevin Yang, Arsha Nagrani, Cordelia Schmid, Andy Zeng, Trevor Darrell, and Dan Klein. Modular visual question answering via code generation. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 747–761, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-short.65. URL https://aclanthology.org/2023.acl-short.65/.

[37] Jiabin Tang, Tianyu Fan, and Chao Huang. AutoAgent: A fully-automated and zero-code framework for LLM agents. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens, editors, *Findings of the Association for Computational Linguistics: ACL 2026*, pages 42924–42974, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.2129. URL https://aclanthology.org/2026.findings-acl.2129/.

[38] Tadahiro Taniguchi, Ryo Ueda, Tomoaki Nakamura, Masahiro Suzuki, and Akira Taniguchi. Generative emergent communication: large language model is a collective world model. *Advanced Robotics*, 40:499 – 524, 2024. URL https://api.semanticscholar.org/CorpusID:275213110.

[39] Yilin Tian, Yue Shang, Chaoxu Kang, and Kaibo Zhao. A review of text-guided vector graphics generation. In *2025 21st International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*, pages 1–6, 2025. doi: 10.1109/ICNC-FSKD67701.2025.11198089.

[40] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models, 2023. URL https://arxiv.org/abs/2305.16291.

[41] Haomin Wang, Jinhui Yin, Qi Wei, Wenguang Zeng, Lixin Gu, Shenglong Ye, Zhangwei Gao, Yaohui Wang, Yanting Zhang, Yuanqi Li, Yanwen Guo, Wenhai Wang, Kai Chen, Yu Qiao, and Hongjie Zhang. Internsvg: Towards unified svg tasks with multimodal large language models. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust, editors, *International Conference on Learning Representations*, volume 2026, pages 38649–38705, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/412f33b7bfecf8d0318113a325339c96-Paper-Conference.pdf.

[42] Zhiruo Wang, Daniel Fried, and Graham Neubig. Trove: Inducing verifiable and efficient toolboxes for solving programmatic tasks, 2024. URL https://arxiv.org/abs/2401.12869.

[43] Jingxuan Wei, Cheng Tan, Qi Chen, Gaowei Wu, Siyuan Li, Zhangyang Gao, Linzhuang Sun, Bihui Yu, and Ruifeng Guo. From words to structured visuals: A benchmark and framework for text-to-diagram generation and editing. In *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 13315–13325. IEEE, 2025.

[44] Ronghuan Wu, Wanchao Su, and Jing Liao. Chat2svg: Vector graphics generation with large language models and image diffusion models. In *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 23690–23700, 2025. doi: 10.1109/CVPR52734.2025.02206.

[45] Zhiyu Yang, Zihan Zhou, Shuo Wang, Xin Cong, Xu Han, Yukun Yan, Zhenghao Liu, Zhixing Tan, Pengyuan Liu, Dong Yu, Zhiyuan Liu, Xiaodong Shi, and Maosong Sun. MatPlotAgent: Method and evaluation for LLM-based agentic scientific data visualization. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Findings of the Association for Computational Linguistics: ACL 2024*, pages 11789–11804, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.701. URL https://aclanthology.org/2024.findings-acl.701/.

[46] Zhentao Ye, Ruyi Ji, Yingfei Xiong, and Xin Zhang. Accelerating syntax-guided program synthesis by optimizing domain-specific languages. *Proceedings of the ACM on Programming Languages*, 10(POPL): 1066–1093, 2026.

[47] Brent Yi, Chung Min Kim, Justin Kerr, Gina Wu, Rebecca Feng, Anthony Zhang, Jonas Kulhanek, Hongsuk Choi, Yi Ma, Matthew Tancik, and Angjoo Kanazawa. Viser: Imperative, web-based 3d visualization in python, 2025. URL https://arxiv.org/abs/2507.22885.

[48] Zian Zhai, Xingyu Tan, Gaowang Zou, Xiaoyang Wang, and Wenjie Zhang. Hyperagent: Planning and acting over tool-schema hypergraphs for tool-use llm agents, 2026. URL https://arxiv.org/abs/2608.02650.

[49] Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-harness: Harnesses that improve themselves, 2026. URL https://arxiv.org/abs/2606.09498.

[50] Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin gödel machine: Open-ended evolution of self-improving agents. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust, editors, *International Conference on Learning Representations*, volume 2026, pages 104223–104294, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/aa5f5e6eb6f613ec412f1d948dfa21a5-Paper-Conference.pdf.

[51] Jingyue Zhang, JD Zamfirescu-Pereira, Elena L Glassman, Damien Masson, and Ian Arawjo. How notations evolve: A historical analysis with implications for supporting user-defined abstractions. In *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems*, pages 1–23, 2026.

Appendix

Contents

A Sub-topic Inventory 17
B Implementation Details 18
B.1 Visual Program and Primitive Registry . . . 18
B.2 Program Generation . . . 20
B.3 Primitive Proposal and Refinement . . . 20
B.4 Candidate Validation . . . 20
B.5 Emergence Schedule and Sharing Scope . . . 21
B.6 Extended Sharing Dynamics and Budget Retraction . . . 21
B.7 Primitive Design Constraints Supplied to the Proposer . . . 22
B.8 Judge Agreement . . . 22
B.9 Cross-Domain Transfer Protocol . . . 23
B.10 Two Drawing Languages Authored Outside This Project . . . 24
B.11 The Two Proposer Design Blocks . . . 25
B.12 Adding a Quality Gate Is Harder Than It Looks . . . 25
B.13 Case Studies . . . 26
B.14 An Emerged Language Is More Compact at Equal Quality . . . 28
B.15 Evaluation Protocol in Full . . . 29
B.16 Emergence-Sample Diversity . . . 30
B.17 Two Null Results: Local Editing and Image Reconstruction . . . 30
C Prompts 31
C.1 Visual Program Generator . . . 31
C.2 VLM-as-Judge Evaluator . . . 35
C.3 Primitive Proposer . . . 35
C.4 Primitive Refiner . . . 39
D Detailed Related Work 43
E Limitations 44

A SUB-TOPIC INVENTORY

The benchmark's 121 sub-topics, grouped by discipline. Each contributes exactly 30 instances, split 15 for emergence and 15 held out, balanced across three difficulty tiers (§4.3). Names are the English renderings used in the paper; the released dataset keys them in Chinese.Representative visual exemplars across all disciplines are illustrated in Figure 5.

Mathematics 27 sub-topics, 810 instances

Algebraic Functions	Analytic Geometry	Box Plot (Multi-Group)
Conic Sections	Correlation Heatmap	Descriptive Statistics
Exp & Log Functions	Heatmap / Correlation Matrix	Histogram + KDE
Hyperbola	Isotherms & Isobars	Lines & Planes in 3D
Parametric Curves	Piecewise Functions	Plane Geometry
Polygons	Probability Diagram	Probability Distributions
Probability Tree	Scatter & Correlation	Shaded Composite Figures
Solid Geometry	Tangent & Derivative	Trigonometric Functions
Vector Dot Product	Violin Plot	t / F Distributions

Physics 29 sub-topics, 870 instances

Bridge Rectifier	Charged Particle in E-Field	Conservation of Energy
DC Circuit	Electromagnetic Induction	Electrostatic Field
Fluid Mechanics	Free-Body Diagram	Gas Laws
Geometric Ray Optics	Heat Transfer	Kinematics
Lenses & Mirrors	Light Curve	Magnetic Field of a Current
Magnets	Op-Amp Circuit	Phase Changes
Planetary Orbits	Pressure & Buoyancy	RC / RL Transients
RLC AC Circuit	Series-Parallel Circuit	Simple Harmonic Motion
Thermal Expansion	Vibrational-Rotational Levels	Voltmeter & Ammeter Wiring
Wave Interference	Wave Reflection & Refraction	

Chemistry 22 sub-topics, 660 instances

13C NMR	1H NMR	Amino Acids
Ball-and-Stick Model	Chain Reaction Mechanism	Complex Ionic Crystal
Cubic Crystal System	Drug Molecules	H Atom Energy Levels
Heterocycles	Hexagonal Crystal System	Hydrocarbons
IR Spectroscopy	Macromolecule Cartoon	Mass Spectrometry
Molecular Orbitals	Nitrogen Functional Groups	Oxygen Functional Groups
Reaction Energy Profile	Space-Filling Model	UV-Vis Spectroscopy
VSEPR Geometry		

Life Sciences 19 sub-topics, 570 instances

Amino Acid Metabolism	Animal / Plant Cell	Carbohydrate Metabolism
Carbohydrates & Biomolecules	Cell Structure	Chloroplast Structure
DNA Replication Fork	Energy Metabolism	Immune Response Pathway
Lipid Metabolism	Meiosis Diagram	Membrane Transport
Mitochondrion Structure	Neurovascular Pathway	Organ Cross-Section
Plant Phylogeny	Protein Interaction Network	Skeletal Structure
Transcription & Translation		

Information Science 16 sub-topics, 480 instances

Adjacency Matrix / List	Algorithm Flowchart	Binary Search Tree
Binary Tree	Business Network Graph	Business Process Diagram
Component Architecture	Decision Flowchart	Decision Tree
Deployment Diagram	Directed Cyclic Graph	Graph Coloring
Mind / Concept Map	Neural Network Architecture	State Transition Diagram
Trie		

Geoscience & Eng. 8 sub-topics, 240 instances

Constellation Chart
Magmatic Intrusion Section
Structural Geology Section
Gear Transmission
Seafloor Topography Profile
World Population Map
Geographic Heatmap
Sedimentary Strata Section

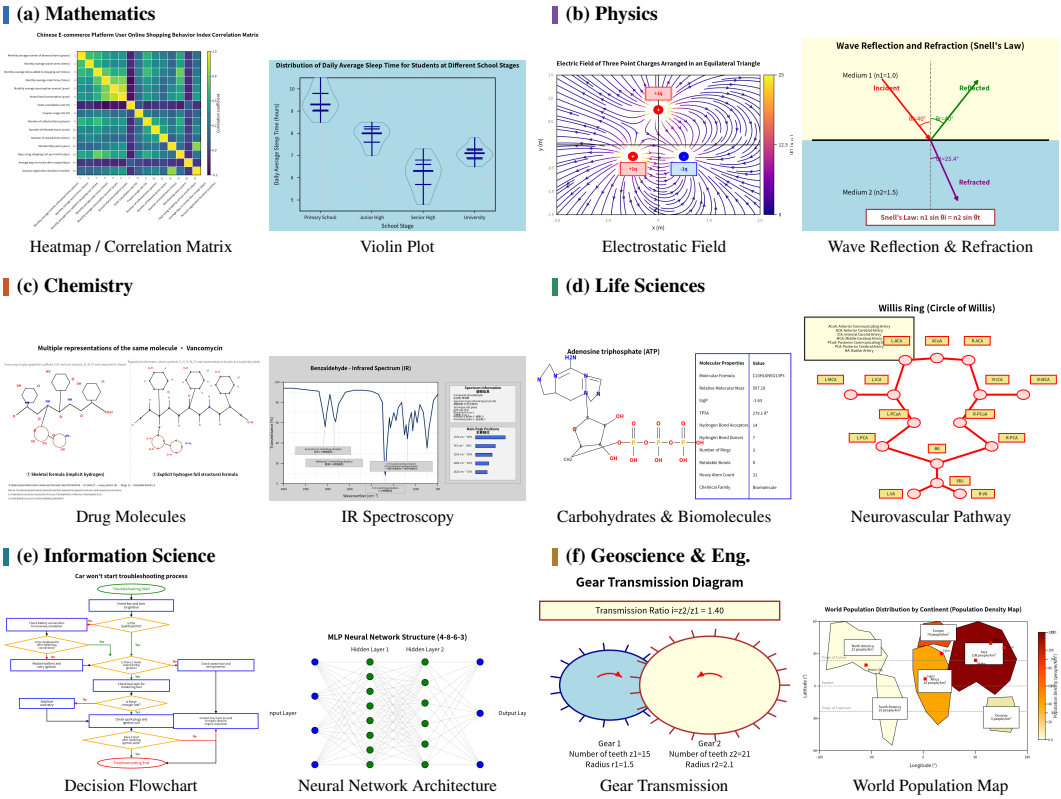


Figure 5: **Two held-out benchmark instances per discipline**, with sub-topic names labeled below each cell (fixed-size cells retain original aspect ratios). Coverage spans function plots, field/energy diagrams, molecular spectra, cellular schematics, state graphs, and geological sections, reflecting broad visual heterogeneity across 121 sub-topics.

B Implementation Details

Algorithm 1 states the round-wise emergence procedure of §3 as pseudocode; the subsections below specify each of its subroutines in turn, namely Generate and Render (Appendix B.2), Propose and Refine (Appendix B.3), Validate (Appendix B.4), and the batch schedule that supplies $\{\mathcal{B}_r\}$ (Appendix B.5).

B.1 Visual Program and Primitive Registry

Each visual program is a JSON object with a canvas specification and an ordered list of commands. The canvas records the pixel size, logical coordinate ranges, coordinate-system metadata, and background color. A command contains a stable element identifier, a primitive name, a parameter dictionary, and optional metadata:

Example: visual program

```
{
  "canvas": {
    "size": [800, 600],
    "coordinate_system": "cartesian",
    "x_range": [-5, 5],
    "y_range": [-5, 5],
    "background": "white"
```

Algorithm 1 Round-wise visual language emergence

Require: Task batches $\{\mathcal{B}_r\}_{r=1}^{R}$, seed registry $\mathcal{L}_0$

$\mathcal{L} \leftarrow \mathcal{L}_0$
for $r = 1, \ldots, R$ **do**
  $\bar{\mathcal{L}}_r \leftarrow \mathcal{L}$
  for all $\tau_i \in \mathcal{B}_r$ **do**
    $z_i^{(r)} \leftarrow \textsc{Generate}(\tau_i, \bar{\mathcal{L}}_r)$
    $I_i^{(r)} \leftarrow \textsc{Render}(z_i^{(r)}, \bar{\mathcal{L}}_r)$
  end for
  for $i = 1, \ldots, |\mathcal{B}_r|$ **do**
    $a_{r,i} \leftarrow \textsc{Propose}(\tau_i, z_i^{(r)}, I_i^{(r)}, \mathcal{L})$
    if $a_{r,i} = \textsc{Modify}$ **then**
      $a_{r,i} \leftarrow \textsc{Refine}(a_{r,i}, \tau_i, \mathcal{L})$
    end if
    if $a_{r,i} \neq \textsc{None} \wedge \textsc{Validate}(a_{r,i}, \mathcal{L})$ **then**
      $\mathcal{L} \leftarrow \textsc{Apply}(a_{r,i}, \mathcal{L})$
    end if
  end for
end for
return $\mathcal{L}$

```
  },
  "commands": [{
    "element_id": "main_segment",
    "func": "draw_line",
    "kwargs": {
      "start": [0, 0],
      "end": [1, 1],
      "color": "red"
    },
    "meta": {}
  }]
}
```

The canvas size is measured in pixels and is independent of the logical coordinate ranges. Every `element_id` must be unique and nonempty. Every `func` must resolve to a primitive in the current registry. The keys and values in `kwargs` must conform to that primitive's schema. Fields equal to their declared defaults may be omitted.

A registry entry stores the primitive name, family, natural-language description, schema, editable fields, smoke-test arguments, provenance, and renderer code. The schema represents every parameter with a field name, type, required flag, semantic description, and default value. The following abbreviated record illustrates the representation of an emerged vector primitive:

Example: registry entry

```
{
  "name": "draw_vector",
  "family": "scientific",
  "description": "Draws a directed line segment.",
  "schema_dict": {
    "fields": [
      {"name": "start", "type": "array[number]",
       "required": false, "default": [0, 0]},
      {"name": "end", "type": "array[number]",
       "required": false, "default": [1, 1]},
      {"name": "color", "type": "string",
       "required": false, "default": "black"}
    ]
  },
```

```
  "editable_fields": ["start", "end", "color"],
  "smoke_test_kwargs": {
    "start": [0, 0], "end": [1, 1]
  },
  "render_code": "..."
}
```

The seed registry contains four entries. `draw_point` places a marker at a coordinate, `draw_line` joins two coordinates, `draw_circle` draws a circle, and `draw_text` places a string. Their schemas expose only geometry and basic visual attributes. No curve, axis, repeated pattern, scientific symbol, or three-dimensional surface is predefined.

The renderer executes each command in an isolated matplotlib axes object. It maps the logical canvas range to the output image, renders the returned artist onto a transparent layer, and composites layers in command order. Isolation prevents one primitive from directly changing the artists created by another primitive. It also makes element-level failures easier to locate.

B.2 Program Generation

For each generation call, Babel serializes the current registry into a compact list of names, descriptions, typed fields, and defaults. The generator receives this serialization together with the task instruction and the visual program contract. It is asked to choose the smallest set of registered primitives that covers the requested elements and to return one JSON object without surrounding prose.

The output first passes document-level validation. Babel checks the canvas fields, command structure, unique element identifiers, and JSON value types. It then verifies that every function name exists in the registry and that every argument is declared by the selected schema. If parsing fails or an unregistered primitive appears, the task is marked unsuccessful without a corrective generation retry. A valid program is sent to the renderer, which records either the output image or the execution error. Both outcomes are retained as evidence for the proposal phase.

B.3 Primitive Proposal and Refinement

Every task in an emergence batch enters the proposal phase, including tasks that render successfully. A successful but unnecessarily long program can expose an efficiency gap, while a failed or incomplete rendering can expose an expressiveness gap. The proposer receives the task, program, image or failure message, current registry, and primitive provenance. It must select exactly one action from ADD, MODIFY, and NONE.

An ADD proposal contains a complete registry entry. It is appropriate when the task requires a visual concept that is not covered by an existing primitive. A MODIFY proposal selects one existing entry and states a narrowly scoped refinement. It is appropriate only when the requested behavior belongs to the primitive's current responsibility. A NONE proposal records no update. This action is used when the language already supports the task or when a useful change would violate the interface constraints.

The proposer follows compact design rules. Each primitive has one visual responsibility. Similar behavior is expressed through a typed parameter when this does not enlarge the interface excessively. Complex figures should be composed from reusable primitives rather than encoded in one task-specific renderer. Parameters are added only when required by observed tasks. Primitive names use concise `draw_<noun>` forms.

A MODIFY action invokes a dedicated refiner. The refiner receives the exact target name, current schema, renderer, and triggering task. Its output replaces the existing entry rather than creating a versioned copy. The prompt instructs the refiner to preserve the old behavior through compatible defaults, but this requirement is not verified against the previous smoke-test arguments.

B.4 Candidate Validation

Validation contains two mandatory stages. First, the schema checker counts fields and Boolean fields, rejecting proposals above the hard limits of ten and two. Second, the renderer code is parsed into a Python abstract syntax tree. The checker rejects syntax errors, dangerous imports and calls, and unsafe attribute access.

The prompts impose additional contracts on candidate code. Coordinate fields should use numeric JSON arrays, expression fields should document their permitted symbols, and renderers should not call `set_xlim`, `set_ylim`, or `set_aspect`. These constraints are not independently checked by the registration code. When strict registration is enabled, the candidate is also executed with its declared smoke-test arguments and must return a nonempty matplotlib artist. Strict registration is disabled by default, and a MODIFY candidate is not tested with the previous primitive's smoke-test arguments.

A candidate that passes the active checks is registered without task-level score comparison. The validation procedure does not establish that the primitive improves visual quality. We measure that property only after the registry is frozen.

B.5 EMERGENCE SCHEDULE AND SHARING SCOPE

The emergence tasks cover the six scientific domains of Section 4.1. We divide each domain into ordered batches and interleave them by round: a complete cycle visits one batch from each domain before returning to the next batch of the first domain. Every batch first completes generation with a fixed registry snapshot, and its task outputs are then processed sequentially for proposals. Each emergence task is processed once, and the final registry is the state obtained after the last scheduled batch.

The registry state is a controlled variable rather than a fixed property of the system, and Section 5.3 varies it directly. Under *shared* scope, all rounds write to one registry, so a primitive added while processing one domain can be selected, modified, or reused by any later task, and separate domains cannot independently register duplicate abstractions. Under *per-domain* scope, each of the six domains maintains its own registry. Under *isolated* scope, each of the 121 subtopics maintains its own registry and is reset to the seed basis before its first round; in this mode a primitive is never visible outside the subtopic that proposed it. Isolated scope is the configuration in which registry names may coincide across subtopics without referring to the same renderer, which is why we report its size as a per-registry mean rather than as the cardinality of the union of names.

The emergence budget N is the number of emergence tasks drawn per subtopic, stratified over difficulty. With the default N=15 and 10 tasks per round, a subtopic is processed in two rounds of 10 and 5 tasks; with $N \leq 10$ a single round suffices. Table 3 sweeps $N \in \{1, 5, 10, 15\}$, and the stratification used at each budget is N=1: one Easy; N=5: 2 Easy, 2 Medium, 1 Hard; N=10: 3 Easy, 3 Medium, 4 Hard; N=15: 5 Easy, 5 Medium, 5 Hard.

B.6 EXTENDED SHARING DYNAMICS AND BUDGET RETRACTION

Registry Growth Timelines and Confounds We count names, not events: 8 of the shared arm's 18 admissions readmit a name already held, which only a shared registry can do. Within-registry counts agree: shared puts 15 of 18 admissions in its first half of rounds, per-domain 26/14, isolated at N=15 293 of 338 in the first of two rounds, and the shared arm's last 31 rounds add one primitive.

Two confounds we cannot remove. A shared registry both closes gaps and lengthens the proposer's serialized prompt. And the isolated row of Table 2 ran its 121-task schedule twice where the others ran it once, so "identical evidence" holds for the task sequence but not the proposal opportunities. That cuts against the isolated arm decaying, not for it: a second pass has more reason to have saturated, and it still shows 1.15.

Isolated Redundancy Breakdown At N=15 the isolated arm's duplicated stems include five arrow primitives (`draw_arrow`, `draw_arrows`, `draw_arrow_series`, `draw_arrow_collection`, `draw_arrow_segments`), four polyline variants, and three ellipse variants. Hence the per-registry mean rather than the union: 121 small private vocabularies, not one language any program can draw on.

Retraction and Variance Analysis at $N = 10$ We previously read the dip to 97.08% at N=10 as evidence that more evidence is not monotonically better, tracing it to two Mathematics subtopics whose admitted primitives absorbed a layout decision. The 1.16pp dip is inside the noise floor, so we withdraw the inference. The failure mode is real and we still assert it, but on the evidence of §B.13, where a single subtopic moves 80pp, and of the Chemistry transfer row in §5.4: both far too large to be noise. What the N=10 churn cannot establish is the effect's frequency, and 9 of 121 subtopics falling below the four-primitive basis at N=15 is the more defensible statement of how

often the failure mode actually occurs. The two levels of Equation 1 appear to have different sample requirements, and the outer search's VLM calls are the expensive resource here.

B.7 Primitive Design Constraints Supplied to the Proposer

The proposer receives a fixed set of design constraints alongside the task evidence. They are stated as constraints on the interface a primitive exposes, not as guidance about which visual concepts are worth abstracting, so the search remains free to decide *what* to add while being constrained in *how*. Section 5.5 reports the structural properties of the primitives that result.

The governing constraint is that a primitive executes geometry it is handed and does not decide geometry on the caller's behalf. A primitive should accept the concrete points, segments, rows, or labels to draw as a list of per-element records, rather than accepting a compact semantic description and deriving positions internally. A record-list field costs one schema field while allowing the caller to specify each element's own coordinates, color, width, label, and visibility, which is how a primitive stays small while remaining fully expressive. Correspondingly, a primitive must not own a layout algorithm: no internal tree balancing, force-directed placement, automatic axis fitting, automatic color assignment, or automatic projection of three-dimensional points. The operational test given to the proposer is whether the primitive contains a decision the caller could not override; if it does, that decision must be exposed as data in the record list.

The remaining constraints follow from this one. Each primitive has a single responsibility expressible in one sentence without conjunctions. Schemas target at most eight fields, with a hard cap of ten and at most two Boolean fields. A new visual concept is introduced by ADD rather than by inflating an existing primitive, and a growing string enumeration or an accumulation of Boolean flags is treated as a signal to compose separate primitives instead. Function families are covered by one generic primitive taking a constrained expression, never by one primitive per function. A MODIFY must keep the previous smoke-test arguments renderable, with new fields defaulting to the old behavior. Names describe the mechanism rather than the subject matter it happens to serve, which is the constraint that yields `draw_labelled_segments` rather than `draw_binary_search_tree`.

These constraints are prompt-level instructions, and only two of the eleven are checked by the registration code: the field-count cap and the Boolean-count cap, both at the proposal stage rather than at registration. The backward-compatibility rule for a MODIFY is checked only under a strict-registration flag that is off by default, and even then it executes the candidate's own smoke-test arguments rather than the previous primitive's, so it cannot detect a compatibility regression (Appendix B.4).

B.8 Judge Agreement

Every arm was scored twice, by two VLMs from unrelated model families, using the identical rubric, the identical rendered images, and the identical pass threshold of 70. Judge 1 is the model that also served as generator; judge 2 is an independent re-scoring pass. Table 5 reports both.

Table 5: **Two-judge agreement across every arm.** Judge 2 is uniformly stricter, more so on the reference languages (1.65–2.37pp) than on the Babel arms (0.44–1.54pp), which is what one expects if the emerged figures are cleaner rather than merely scored more generously. It preserves 32 of 36 pairwise orderings; all four inversions separate arms by under 0.4pp under either judge.

	Pass rate (%)			Mean score		
Arm	Judge 1	Judge 2	Δ	Judge 1	Judge 2	Δ
Seed-4	96.14	94.49	1.65	91.9	88.2	3.74
Human-28	88.10	85.73	2.37	88.6	84.5	4.13
Python	95.92	94.10	1.82	94.4	92.4	2.02
Babel, shared N=1	96.80	96.36	0.44	93.6	90.8	2.73
Babel, per-domain N=1	95.76	95.32	0.44	93.3	90.4	2.90
Babel, isolated N=1	97.69	96.86	0.83	94.7	92.0	2.72
Babel, isolated N=5	98.07	97.47	0.61	94.7	91.8	2.86
Babel, isolated N=10	97.08	95.54	1.54	94.4	91.3	3.14
Babel, isolated N=15	98.24	97.25	0.99	94.9	92.2	2.70

The cross-language conclusions do not depend on the judge. Recomputing the paired contrasts of Table 1 under judge 2 widens every gap: Babel over the seed basis moves from +2.09 to +2.75pp (CI $[1.10, 4.46]$), over Python from +2.31 to +3.14pp (CI $[0.77, 5.90]$), and over the hand-designed DSL from +10.14 to +11.52pp (CI $[7.88, 15.37]$). All three intervals exclude zero under both judges, and the Python interval, whose lower bound touches zero under judge 1, clears it under judge 2. The shared-versus-isolated comparison remains indistinguishable either way, at +1.43pp and +0.88pp with intervals straddling zero.

That judge 2 penalises the reference languages roughly twice as hard as the Babel arms is worth stating plainly, because it could be read either as the emerged figures genuinely being cleaner or as a residual bias we have not isolated. The rubric contains no criterion that references the language, and both judges see the program, so neither reading can be excluded from these data alone. What the two judges settle is the weaker claim we rely on: no ordering that the paper asserts is an artifact of one scorer.

B.9 Cross-Domain Transfer Protocol

The transfer matrix of Section 5.4 evaluates all 36 ordered pairs over the six domains. Source vocabularies are the six per-domain registries of the sharing experiment, loaded read-only: no primitive is proposed, refined, or admitted during a transfer cell, so a cell measures the frozen vocabulary alone.

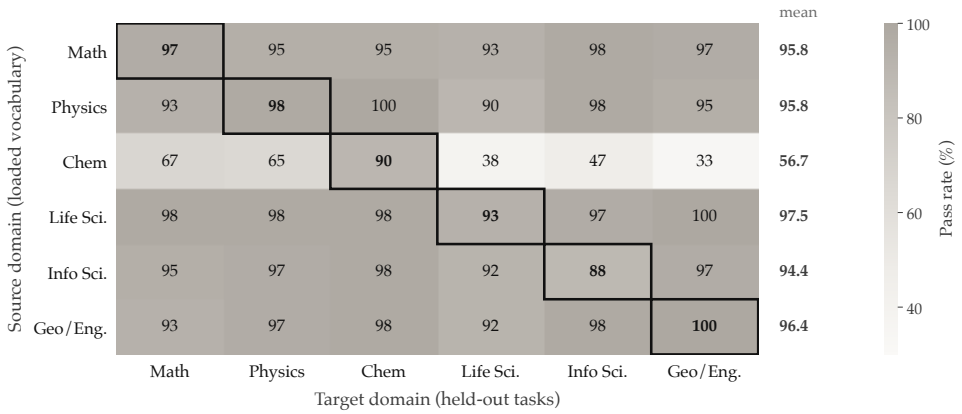


Figure 6: **Cross-domain vocabulary transfer** over 2,160 tasks; same data as Table 4. Five of six rows transfer at or above the boxed diagonal. Chemistry's primitives are selected on 46 to 58 of 60 foreign tasks and draw them wrongly.

Target task sets are sampled once per domain and reused across every source, which is what makes a column internally comparable and its diagonal cell a valid reference. Each target set holds 60 held-out instances covering all subtopics of that domain at 20 Easy, 20 Medium, and 20 Hard. Sampling is seeded, so all six cells of a column receive an identical task list in an identical order. The matrix therefore contains $36 \times 60 = 2{,}160$ generation and judging calls, and reports 89.4% overall.

We measure transfer relative to the diagonal rather than in absolute terms. Domains differ in intrinsic difficulty, so a raw off-diagonal value conflates the cost of switching vocabulary with the difficulty of the target; subtracting the column's diagonal cell isolates the former. A pre-flight check aborts the run when a source registry directory is missing or empty, rather than proceeding with the seed registry. This is a direct response to the harness defect described in Appendix B.16, where a silent fallback produced a full matrix of plausible but meaningless numbers.

Two limits of this design are worth stating. The grid is over domains rather than the 121 subtopics, because a subtopic-level matrix would require 14,641 cells and roughly 220,000 tasks; a domain-level grid answers the question we posed and remains readable. And each cell is a single run, so the per-cell values carry the same run-variance caveat as the rest of our results, though the row-level pattern in Table 4 rests on six cells apiece.

Failure mode analysis for Chemistry primitives. The cause of Chemistry's 56.7% transfer drop is not disuse: its primitives are selected on 46 to 58 of the 60 tasks in each foreign domain, so the generator reaches for them readily and the figures come out wrong. Their mechanism-shaped schemas are what makes them attractive to the generator and what makes the failure interesting: an abstraction can satisfy every constraint in Appendix B.7 and still be wrong for the task, a Bezier path being a plausible way to draw a cell membrane and a poor one. Mechanism orientation makes a primitive reusable where it applies; it does not tell the generator where it does not.

Emergence sampling diversity retraction. Holding the budget at 121 tasks and varying only their spread over subtopics leaves every pairwise pass-rate difference between 0.39 and 1.38pp, all below the 1.43pp threshold, so we retract the claim that concentrated sampling costs quality. The effect the mechanism predicts appears instead in the vocabulary: concentrating on 9 subtopics emerges 8 primitives against 10 under full coverage. Appendix B.16 gives the arms in full, together with a harness defect that silently invalidated an earlier version of them and the fix it required.

Table 6: **Swapping the generator, holding the language fixed.** One 149-task subset, one frozen registry, one judge; only the generating model changes. Each runs its own recommended configuration, so parameters are not matched. The detectable difference is 4.98pp.

Generator	Pass (%)↑	Score↑	Cmd↓	Render fail↓	ΔPass (pp)
`gpt-5.6-luna (xhigh)`	**98.66**	**94.13**	28.4	**0**	—
`gemini-3.7-flash`	95.30	89.86	**23.7**	2	−3.36 (within noise)
`claude-sonnet-4-6`	65.77	77.10	37.8	4	−32.9
`gpt-4o`	16.78	46.86	11.9	13	−81.9

Cross-model evaluation and plumbing validation. The failures across non-reasoning generators are capability, not plumbing: all three arms loaded and used the registry, parsing 136–147 of 149 programs and calling 89–92 distinct emerged primitives, so none silently fell back to the seed basis. A language that assumes the generator can hold a figure's whole structure in mind has nothing to offer one that cannot. Two caveats bound this further: each model ran its own recommended configuration, since our client strips temperature from reasoning models, so this compares models-as-deployed rather than at matched parameters; and one further candidate had to be excluded outright because the gateway reported no such model available.

B.10 Two Drawing Languages Authored Outside This Project

Human-28 is ours, so its deficit admits the reading that we built a poor DSL. We therefore ran two languages nobody in this project designed — **TikZ/PGF** and **raw SVG** — under the same instruction, generator, effort, and judge as every other arm. Table 7 also re-derives the three reference contrasts of Table 1 on these 121 tasks, and they land where the full benchmark put them: 4.30× against the seed basis (published 4.06×), 1.66× against the hand-designed DSL (1.67×), and a Human-28 deficit of 8.26pp (8.0pp), the one contrast whose McNemar test rejects here ($p = 0.021$). Only six of these tasks appear in any other subset, so this is close to an independent replication.

Quality is indistinguishable; compression separates them, and one of the two brackets. TikZ and Babel both pass 96.69% ($p = 1.0$) and SVG reaches 98.35% ($p = 0.63$), all far inside 5.53pp, so we report no ordering. The judge's raw score is the one place the external languages lead consistently, 94.60 and 97.07 against Babel's 93.84; within noise at this size, but it does not favour us. SVG needs 2.19× more commands, 57.40 against 26.25, with all six domains between 1.66× and 2.93×, and its source is 2.9× longer in characters, 7,863 against TikZ's 2,748; its drawing element sits at exactly the granularity of a primitive call, so the unit is identical by construction. TikZ instead brackets $[0.85, 1.18]\times$, 22.21 source commands against 30.93 `\foreach`-expanded operations, and splitting by whether the program calls `pgfplots` locates its whole advantage: 14.13 commands against Babel's 37.49 on the 39 tasks that do (0.38×), where a charting library draws an entire axis for zero counted commands, against 21.01 to 26.38 on the 80 that do not (1.26×). An external, heavily-engineered DSL therefore lands between Babel and the seed basis rather than beyond it: SVG's 57.40 is 0.51× Seed-4's 112.83. Four of 242 generations failed and count as failures; and these two arms ran through a different inference gateway after the original exhausted its budget mid-run, with model, effort, and prompts held identical and the partial run discarded rather than stitched, but a gateway-level difference we cannot rule out (§E).

Table 7: **Six languages on one task per sub-topic.** One Medium instance per sub-topic by a rule reading no result, so all arms are paired. **The detectable difference at n=121 is 5.53pp**, so only Human-28 is resolvable. †`\foreach` is a macro loop (34 tasks). ‡No primitive-call granularity.

Language	Pass (%)↑	CI95	Score↑	Cmd↓	Cmd ratio to Babel
Babel (isolated)	96.69	[91.8, 98.7]	93.84	**26.25**	—
Seed-4	95.04	[89.6, 97.7]	91.90	112.83	4.30×
Human-28	88.43	[81.5, 93.0]	88.45	43.64	1.66×
Python	93.39	[87.5, 96.6]	91.15	—‡	—‡
TikZ/PGF	96.69	[91.8, 98.7]	94.60	22.21†	0.85–1.18×†
raw SVG	**98.35**	[94.2, 99.5]	**97.07**	57.40	2.19×

B.11 The Two Proposer Design Blocks

Table 8: **Stating the mechanism constraint in the proposer's prompt.** Two emergence runs over the same 1,815 tasks; the proposer's design block is the only difference. The block buys quality, not compression: commands move 0.35, inside the noise floor, while the vocabulary narrows.

Proposer design block	Pass (%)↑	Pass_2 (%)↑	Cmd↓	Em./reg.	Union
original	93.33	92.45	27.96	3.96	252
mechanism-oriented	**98.24**	**97.25**	28.31	**2.43**	**118**

The contrast of Table 8 is between two versions of the design block interpolated into the proposer prompt, reproduced in full in Appendix B.7. The later block opens with the governing constraint — a primitive executes geometry it is handed and does not decide geometry on the caller's behalf — and states the operational test that goes with it: whether the primitive contains a decision the caller could not override. The earlier block carried the remaining rules without that governing statement, and phrased five of them in terms of what a primitive should accept rather than in terms of per-element records. Both runs are otherwise matched on the generation path, differing in no other file that the generator, the visual-feedback stage, or the renderer reads, so the design block is the only variable reaching the tested code.

Two consequences of that design bear on how the result should be read. The comparison is block-level, so it cannot separate the governing constraint from the five rewordings that accompany it. And the constraints remain prompt-level instructions: of the eleven, only the field-count and Boolean-count caps are checked by the registration code, and the backward-compatibility rule for a MODIFY is checked only under a strict-registration flag that is off by default, and even then against the candidate's own smoke-test arguments rather than the previous primitive's (Appendix B.4).

We can bound this directly, because the mechanism-oriented design block replaced an earlier one and the two runs are otherwise matched: same 1,815 tasks, same generator at the same effort, same prompt contract, with only the proposer's design block differing. Table 8 gives 93.33% against 98.24%, a +4.90pp gain on 112 against 23 discordant subtopic-level outcomes ($p = 3.0 \times 10^{-15}$), and +4.79pp under the second judge. That is 3.4× the 1.43pp threshold of §5.1 and clear even of the conservative 4.9pp reading, which makes it one of the three quality orderings this paper asserts outright.

The mechanism-oriented block states the governing constraint explicitly and also rewords five of the ten remaining rules toward per-element records, so we cannot attribute the gain to one sentence. Sections 5.4 and B.13 bound the principle from the other side, showing that mechanism-shaped primitives can still transfer badly and that the search sometimes admits policy-shaped ones anyway.

B.12 Adding a Quality Gate Is Harder Than It Looks

The obvious repair is to score a candidate before admitting it. We built two such gates and report a negative: the stronger-looking one is the weaker gate. We expected `unit_tests` to be stricter, since verifying against fresh cases is harder than reproducing the original task, and the opposite holds — its judge scores run 81–97 with a median of 92 and it rejects nothing, while `replay` scores 38–86 with a median of 62 and rejects 6 of 15. `unit_tests` asks the model to author the cases it will be judged

Table 9: **What a candidate primitive must pass before admission.** Three arms over 20 subtopics at $N{=}1$. A gate rejecting more leaves more subtopics on the seed basis, so the arms cover different task sets and their marginal pass rates are not comparable (see text).

Gate	Accepted	Rejected	Reject rate	$\lvert\mathcal{L}\rvert$	Judge score of proposals
none (ours)	14	2	12.5%	**13**	—
replay	6	9	60.0%	5	median 62, 6 rejected
unit_tests	12	3	20.0%	10	median 92, **0** rejected

against, so proposal and examination come from one model. The reliable value came from the cheap mechanical check instead: both arms caught two primitives whose render function returns nothing, `draw_rounded_rectangles` and `draw_filled_polygons`, which the ungated arm admitted, and the eight extra LLM calls per proposal that `unit_tests` spends caught nothing beyond that.

Whether gating improves downstream quality we cannot say, and the marginal rates suggesting it does are an artifact. A gate that rejects more leaves more subtopics without a registry, falling back to the seed basis, so the arms are scored on different task sets: 140 tasks for none, 100 for `unit_tests`, 30 for `replay`. On its own 30 tasks `replay` scores 100% against none's 83.57% over 140, but none scores 96.67% on those same 30; the apparent gain is a subset effect. Restricting to the 20 tasks all three cover gives 19/20 for none and 20/20 for both gates — a ceiling effect and an underpowered design, not evidence of equality.

Nothing enforces the constraint at admission. A candidate is registered for being well-formed and safe, never for improving a rendering, and only two of the eleven constraints are checked by code at all. Asking the proposer to author its own test cases yields a gate that rejects nothing, since proposal and examination come from one model.

B.13 Case Studies

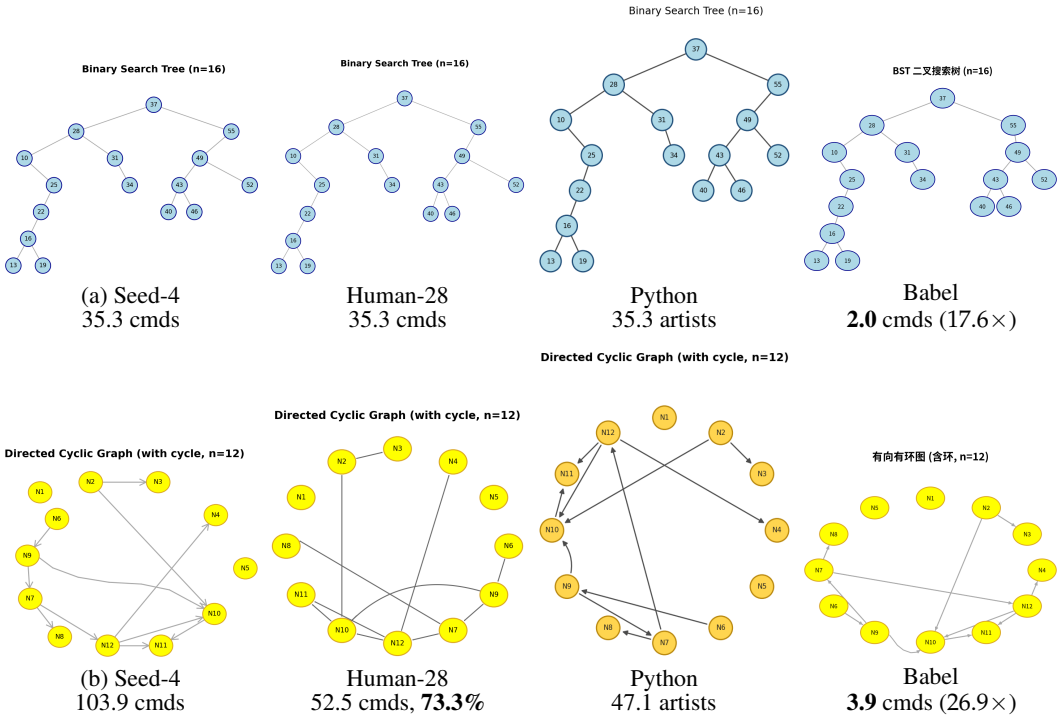
Figures 7–9 show held-out tasks under each language, one subtopic per effect. The per-case ratios are near-extremal rather than typical: the 17.6× and 26.9× below sit against a benchmark mean of 4.06×. The last case is our own search failing, not a baseline.

Compression without loss of control. In Figure 7 the emerged primitive still takes explicit node and edge records: the generator chooses every vertex position and only stops repeating the stroke call, which is why 17.6× and 26.9× cost nothing in accuracy. The same panels show a hand-designed vocabulary's first crack: Human-28, with no node-link abstraction, bends edges with `draw_parametric` and loses 26.7 points of pass rate to a four-primitive language that has no graph abstraction either.

Subject-matter abstraction can be worse than none. Human-28's spectrum primitives in Figure 8a take a functional form rather than sampled points, so a task naming particular absorption maxima has no faithful encoding and no fallback: it passes 13.3% where a basis without a spectrum primitive passes 100%. Piecewise functions, correlation heatmaps, box plots, and NMR spectra collapse the same way; with UV-Vis these five subtopics account for 54 of the 183 tasks Human-28 loses to the seed basis.

A wide language is not a safe one. Figure 8b is the opposite failure. A registry validates programs against typed schemas before rendering, so an unparseable program is rejected rather than executed; free-form Python has no such check and fails all 15 instances with an execution error. That is the price of expressing anything, and the seafloor profile is the milder version at 60.0%. The trade runs both ways: on the decision flowchart the seed basis breaks down, 66.7% against Babel's 100%, because a labelled process box assembled from bare circles and line segments is possible but fragile.

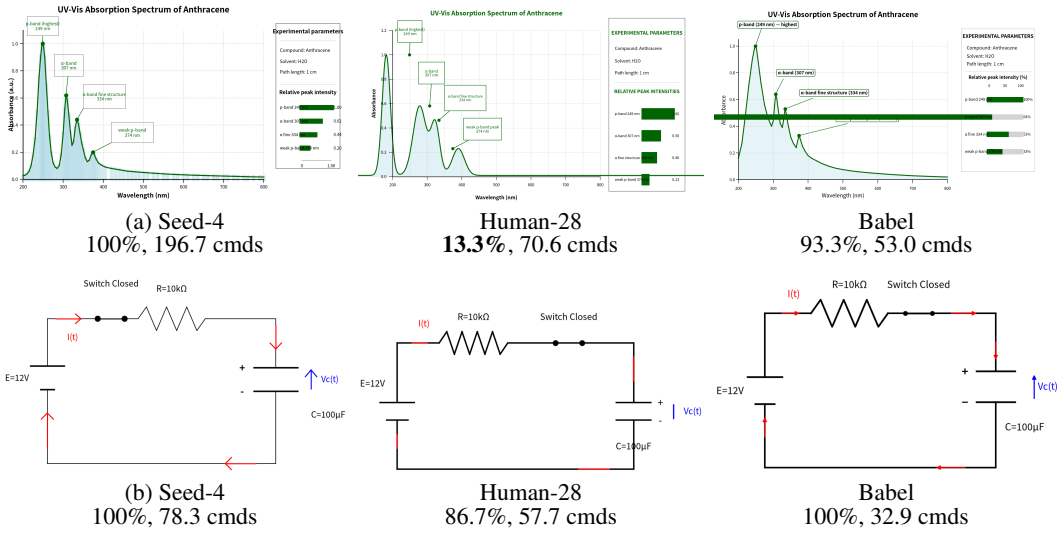
Our own search reproduces the failure it was built to avoid. In Figure 9 our method produces the defect it was designed to avoid, and at 80pp it is the only instance large enough to assert on its own. An admitted primitive computes field-line geometry internally and every program selects it, so the figures render, look like field diagrams, and show the wrong field. This is Human-28's spectrum failure reached by search rather than by hand, which makes the mechanism constraint of §5.5 a target the outer search can miss rather than a property the system enforces. With the Chemistry row of Table 4 it points at the one component the design is missing, which Appendix B.12 then tries and fails to supply.



(a) Seed-4, 35.3 cmds | Human-28, 35.3 cmds | Python, 35.3 artists | Babel, **2.0** cmds (17.6×)

(b) Seed-4, 103.9 cmds | Human-28, 52.5 cmds, **73.3%** | Python, 47.1 artists | Babel, **3.9** cmds (26.9×)

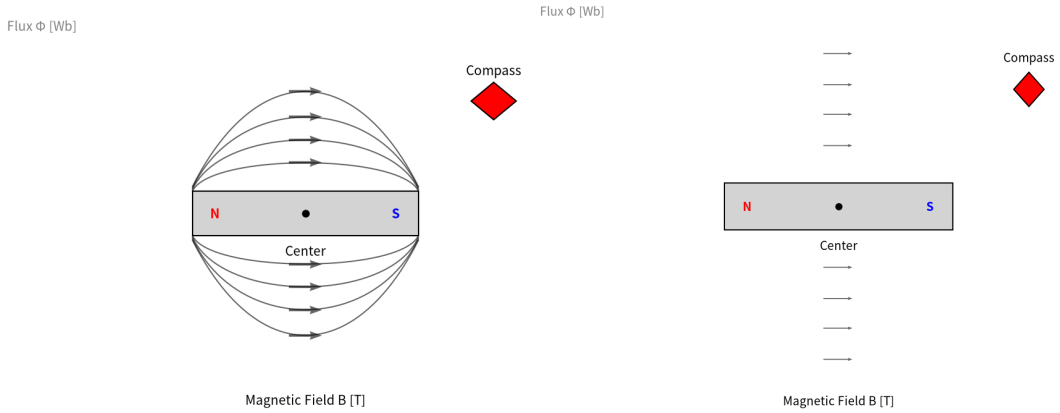
Figure 7: **Same held-out task, four languages: what compression looks like.** All panels scored 100% except where marked. **(a)** A binary search tree: three languages place every node and edge at 35.3 commands, Babel needs 2.0. **(b)** A directed cyclic graph at 26.9×.



(a) Seed-4, 100%, 196.7 cmds | Human-28, **13.3%**, 70.6 cmds | Babel, 93.3%, 53.0 cmds

(b) Seed-4, 100%, 78.3 cmds | Human-28, 86.7%, 57.7 cmds | Babel, 100%, 32.9 cmds

Figure 8: **Two ways a language fails that compression does not predict. (a)** A UV-Vis spectrum: Human-28 owns primitives built for it and passes 13.3% at score 62.8, the four-primitive basis 100% at 92.7. **(b)** An RC-RL transient: free-form Python fails all 15 instances.



N=5: 100% pass, 19.3 cmds N=15: **20.0%** pass, 18.7 cmds

Figure 9: **A language can regress: magnetic field lines at two budgets.** The program shrinks monotonically, 39.3, 19.3, 19.4, 18.7 commands at N=1, 5, 10, 15, while pass rate holds at 93.3, 100, 100 then collapses to 20.0 once the search admits primitives deriving field-line geometry internally.

B.14 An Emerged Language Is More Compact at Equal Quality

Table 10 provides the comprehensive per-discipline metrics across all 121 sub-topics and 1,815 held-out instances, and Figure 10 plots the same aggregate arms of Table 1 as a quality-against-effort trade-off.

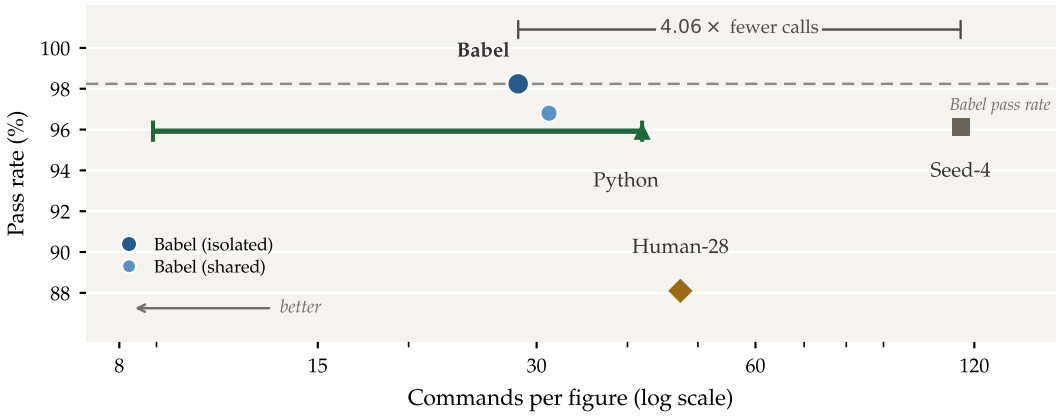


Figure 10: **Quality against editorial effort** over 1,815 tasks. The bracket spans Babel (isolated) at 28.3 commands and Seed-4 at 114.9; Babel's two arms lie 1.44 pp apart. Python is bounded between 8.9 source calls and 41.9 artists.

Pathology of hand-designed abstractions. The sharp failure of Human-28 in Chemistry (83.9%) and Mathematics (77.3%) illustrates the risk of high-level semantic encapsulation. Primitives such as `draw_molecule` encapsulate fixed bond geometry and 2D projections. When a target requires custom spatial conformation or non-standard resonance structures, the primitive cannot satisfy the constraints, and its presence preempts lower-level coordinate primitives. In contrast, Babel's outer emergence search explicitly rejects abstractions that reduce domain task coverage, converging on composable structural operations rather than rigid monolithic helpers (see §5.5).

Evaluation on external languages: TikZ and SVG. To benchmark outside our generation environment, we evaluated TikZ/PGF and raw SVG across one Medium task per sub-topic ($n = 121$, minimum detectable difference 5.53 pp). Raw SVG matches quality but requires 2.19× more commands (57.40 vs. 26.25 for Babel), as SVG primitives match our command granularity directly. TikZ/PGF operates at a $[0.85, 1.18]\times$ command ratio; its conciseness is isolated entirely to 39 tasks calling `pgfplots` (where axes and grids are drawn implicitly), while across the remaining 80 tasks Babel is strictly more compact.

Table 10: **Held-out results across six scientific domains** (1,815 tasks; 15 instances × 121 subtopics). Python's artist count is not a primitive call, so its ratio brackets $[0.7\times, 1.5\times]$. $\overline{|\mathcal{L}|}$ includes the four seeds, *Em.* does not; isolated rows are means over 121 private registries.

			Math		Physics		Chemistry		Life Sci.		Info. Sci.		Geo/Eng.	
Language	$\overline{\lvert\mathcal{L}\rvert}$	Em.	Pass↑	Cmd↓	Pass↑	Cmd↓	Pass↑	Cmd↓	Pass↑	Cmd↓	Pass↑	Cmd↓	Pass↑	Cmd↓
Seed-4	4	0	97.8	127.5	97.0	97.1	96.4	89.0	94.7	126.4	94.2	104.3	94.2	201.3
Human-28	28	0	77.3	39.1	87.8	39.1	83.9	55.8	97.2	43.0	97.5	56.8	96.7	73.4
Python	—	—	96.8	23.4	92.6	37.3	96.4	63.6	97.9	46.6	98.3	48.6	94.2	37.5
Babel (shared)	14	10.0	97.3	33.0	94.9	27.2	**99.1**	41.8	97.2	22.9	95.4	28.1	97.5	36.5
Babel (isolated)	6.4	2.4	**98.5**	33.3	96.3	28.1	98.8	31.0	**98.9**	20.3	**99.6**	18.4	**98.3**	43.0

B.15 Evaluation Protocol in Full

Metrics. Quality is scored by a separate VLM call seeing the instruction, the rendered image, and the program, returning 0–100; a task passes at 70. The rubric asks only whether the image matches the instruction, though the judge does see primitive names, so the score is not blind to the language (Appendix C). Every arm is re-scored by a second VLM from an unrelated model family (Appendix B.8). Efficiency is the command count, the operational form of $|z|$ in Equation 1: one command is one primitive call. We also report *distinct* primitives per figure, separating compression by reuse from compression by a wider vocabulary.

Comparing across languages. A task producing no image scores zero rather than being dropped: free-form Python fails all 15 instances of one Physics subtopic, and those count as failures in its 95.92%. Registry-based languages share a program format and therefore a unit; Python does not, so we report both proxies, 41.9 runtime `matplotlib` artists and 8.9 source drawing calls, differing fourfold because one call in a loop emits many artists. Neither equals a primitive call, so Python's efficiency is bracketed rather than measured.

Reference languages. Four languages under one generator, judge, decoding configuration, and prompt contract, so the language is the only variable. **Seed-4**, the initial registry of `draw_line`, `draw_point`, `draw_circle`, and `draw_text`, is the ablation removing emergence and nothing else. **Human-28** is a DSL of 28 primitives we hand-designed before any emergence run, including subject-matter abstractions such as `draw_molecule`; it represents the practice this paper argues against. **Python** gives unrestricted `matplotlib`. **Babel** is the frozen emerged registry.

Statistics. Tasks within a subtopic share a prompt template and a generation seed, so treating 1,815 tasks as independent would understate uncertainty. We cluster on subtopic: intervals are 95% percentile CIs from a paired cluster bootstrap over the 121 subtopics (20,000 resamples), p-values two-sided Wilcoxon signed-rank tests over per-subtopic pass rates.

What this pipeline can resolve. Repeating one 149–150 task subset five times, re-judging fixed programs gives $\sigma = 0.599$ percentage points, hereafter **pp**; regenerating then re-judging gives $\sigma = 1.257$pp, of which generation contributes 77%. Extrapolating to $n = 1{,}815$ gives a minimum detectable effect of **1.43pp** at 80% power. That extrapolation treats tasks as independent, which our own clustering rejects, claiming 12.2× more independent units where at most 1.05× more clusters exist: 89 of the subset's 115 subtopics contribute exactly one task. A fully clustered reading gives $\sigma \approx 1.23$pp and a detectable effect near 4.9pp, so our resolution lies in $[1.43, 4.9]$pp. The evidence favours the low end: the 12 flips fall in 11 distinct subtopics, and the one subtopic flipping both its tasks stands against 0.17 expected, at $p \approx 0.16$ suggestive and not more. We keep 1.43pp as the working threshold, noting that one set too low retracts too few claims. It changes two results: §5.3 and Appendix B.16 fall below it and we retract what we drew from them, while the compression results clear even 4.9pp by an order of magnitude.

Why quality is judged and not measured in pixels. None of the three reference-based alternatives is available. The benchmark has no ground-truth image, so there is nothing to compute a per-task similarity against. Where we could construct references, SSIM inverts: 20 blank canvases score 0.850 against 0.762–0.783 for genuine reconstructions (Appendix B.17), and a metric ranking a blank

Table 11: **Emergence-sample diversity at a fixed budget of 121 tasks.** *D-maxcov* draws one task per subtopic, *D-random* samples 121 uniformly, *D-low* concentrates them on 9 subtopics. Contrasts are against D-maxcov, with 95% cluster-bootstrap CIs over 121 subtopics. All six pass rates lie within 1.38pp, below the 1.43pp threshold (§5.1), so no ordering there is resolvable; *Em.* and **Cmd** carry the effect. *Em.* unions an arm's registries, so per-domain rows can exceed what one program sees.

Scope	Sample	Subtopics	Pass (%)↑	Pass_2 (%)↑	Cmd↓	Em.	ΔPass vs. maxcov (pp)
Shared	D-maxcov	121	**96.80**	96.36	31.2	10	—
Shared	D-random	121	96.20	95.15	25.7	8	−0.61 [−2.31, 1.05] p=0.28
Shared	D-low	9	95.43	94.21	**23.1**	8	−1.38 [−2.87, 0.17] p=0.01
Per-domain	D-maxcov	121	95.76	95.32	27.3	30	—
Per-domain	D-random	121	96.20	95.10	28.3	7	+0.44 [−1.60, 2.70] p=0.48
Per-domain	D-low	9	95.37	93.99	46.2	8	−0.39 [−2.42, 1.82] p=0.57

page above every correct figure cannot arbitrate. CLIPScore truncates at 77 tokens where all 1,815 instructions exceed it at a median of 119 words; FID compares distributions rather than pairs.

B.16 Emergence-Sample Diversity

If expressiveness gaps drive growth, concentrating the budget on a few subtopics should yield a narrower language. Table 11 holds the budget at 121 tasks and varies only their spread.

Pass rate cannot answer this question. Every pairwise difference is between 0.39 and 1.38pp, all below the 1.43pp threshold of §5.1. We previously reported the 1.38pp shared D-low contrast as significant on a McNemar test at $p = 0.01$ while framing the section as a null result; the two readings were inconsistent, and the threshold settles it against us. We retract the claim that concentrated sampling costs quality.

The effect the mechanism predicts shows up on a different measurement. D-low emerges a narrower language, 8 primitives against 10 under shared scope, and under per-domain scope it also calls them harder: 46.2 primitive calls per task against 27.3 for full coverage and 28.3 for uniform sampling, with rendering failures at 58 against 30 and 8. Under shared scope that metric inverts, 23.1 calls against 31.2, because a single registry serving 9 subtopics is tuned tightly to them. Low diversity is not free, but its cost lands on vocabulary width and robustness rather than on figures the judge marks wrong; testing it more sharply needs a smaller budget or domains sharing less visual structure.

These numbers replace a retracted version. A harness defect made the D-low and D-random arms load the seed registry while reporting emergent-arm labels: `registry_dir_for_scope()` ignored the `--emergence-roots` it was passed and resolved to an empty directory, so all four arms silently measured the four-primitive basis. Nothing failed and every pass rate stayed plausible; the only visible symptom was a command count near 110, four times the corrected value and what a genuine Seed-4 arm produces. Our acceptance check now requires all 121 per-subtopic summaries to contain 15 tasks and emerged primitive names to appear in the recorded call traces, and it aborts rather than warns.

B.17 Two Null Results: Local Editing and Image Reconstruction

We ran two experiments expecting the emerged vocabulary to win and it did not. Both are reported because they bound where the compression result of Section 5.2 does and does not convert into other advantages.

Local editing: our hypothesis was wrong, and so was our engine. We expected coarse-grained primitives to be worse at local edits, on the reasoning that a primitive taking 20 nodes as one record forces a wider rewrite to change one node than 20 individual calls do. We tested 600 items: 100 figures × 3 edit families (recolor, remove, add) × 2 languages (emerged against a paired seed basis), with the gold patch derived deterministically from the spec and checked leaf by leaf.

The first run appeared to confirm the hypothesis emphatically. Emerged recolor succeeded on 31 of 100 against the baseline's 100, with all 69 failures being cases where the target leaf sits inside a list, and a McNemar test at $p < 10^{-20}$. The hypothesis was wrong: this was a defect in our patch engine. The model had been emitting correct single-field patches all along, addressing paths

such as `kwargs.nodes[5].facecolor`, but `patch_update` split the path on `.` and called `setdefault` without parsing the index, creating a stray key named `"nodes[5]"` while leaving the real leaf untouched. The patch reported success, the figure rendered, and nothing changed. With `drawspec/ir.py` as the only variable between runs, the fix moved emerged recolor from 0/69 to 69/69 on the list-embedded cases and the two languages became indistinguishable: 100% success in both, McNemar $p = 1.0$ with zero discordant pairs, and edit sizes within 5% of each other.

One real cost survives, unpaid rather than absent: an engine that cannot index into a list must rewrite the whole record, which we measured at 791 characters against roughly 60 for a single-field patch, about 13×. A second lesson concerns the metric. `PatchEngine.compute_locality` normalizes by command count, which penalizes the emerged arm structurally, since a 26-command program is charged more for the same edit than a 121-command one (0.941 against 0.992). Leaf-level locality is the correct primary measure and puts both arms at ≈ 0.999.

Image reconstruction: no reliable advantage, at a third of the program length. We tested whether an emerged vocabulary helps recover a DrawSpec from a rendered image alone, giving the solver the PNG plus the domain and subtopic name and never the original instruction. This is 480 items: 120 figures × 2 reference sources × 2 solver vocabularies.

No ground-truth spec exists on disk, so every reference image was produced by one of the two arms, and that arm holds a self-reference advantage on the diagonal. Only the off-diagonal is an honest comparison, as with the transfer matrix. There the emerged solver reaches 77.5% against the baseline's 84.2%. Over all 217 paired items a McNemar test gives $p = 0.0436$, but restricted to the honest off-diagonal subset it is 7 against 16 discordant pairs at $p = 0.0931$. The two disagree because the diagonal carries the self-reference bias, so we take the conservative reading: no reliable advantage, which is not the same as confirmed worse. The compression advantage does persist, at 37.0 commands and 8.9k characters against 113.6 and 21.6k, roughly 3×. Both arms are weakest on the same sub-axis, text, at 66–68 against 81–94 for elements, layout, and color.

A floor arm of 20 blank canvases with no model calls checked the metrics rather than the method, and caught one that would have produced a false result. The judge was not fooled, scoring 0 and passing none. SSIM gave those blank canvases 0.850, because scientific figures are mostly white and pure white already matches about 86% of the pixels. Judge score and pixel metrics correlate only moderately, $r = 0.405$ with SSIM and 0.161 with ink coverage, so we gate on ink coverage and treat SSIM as secondary.

Table 12: **SSIM against a judge, on the one experiment where reference images exist.** The reconstruction study is the only setting in this paper with a per-item reference image, since each reference was rendered by one of the two arms; the held-out benchmark has no ground-truth image at all (§5.1). The floor row is 20 blank canvases with no model calls. SSIM ranks that floor *above* all four real conditions, spanning 0.762–0.783 against the floor's 0.850, while the judge separates them correctly at 0 against 77.5–84.2. This is why we report no pixel metric in the main results: on white-background line art SSIM is not a weak instrument but an inverted one.

Reference	Solver	n	Judge pass (%)	Judge score	SSIM
Floor: blank canvas		20	0.0	0.0	**0.850**
emerged	emerged	120	81.7	80.1	0.783
emerged	baseline	120	84.2	82.1	0.762
baseline	emerged	120	77.5	79.8	0.767
baseline	baseline	120	81.7	82.3	0.770

C Prompts

C.1 Visual Program Generator

Prompt: Visual Program Generator

You are a senior scientific-figure programmer. You translate natural-language figure requests into executable visual programs that can be rendered as publication-quality images.

 # 1. Visual program contract

A visual program is typed and element-addressable. Every figure is described as a single JSON object that the renderer can deterministically execute.
The document is the 4-tuple (C, E, F, Θ):

C = canvas metadata → `"canvas"` key
E = ordered element IDs → `"commands[*].element_id"` (stable handles for editing)
F = primitive functions → `"commands[*].func"` (must exist in the registry below)
Θ = typed parameter sets → `"commands[*].kwargs"` (must conform to each primitive's schema)

Schemas:

- `canvas: { size: [w_px, h_px] (PIXELS, NOT data coords. Typical: 800×600, 1200×800, 1600×900. For multi-panel use ≥ 800px wide. Do NOT set to data-range values like [20, 5]), coordinate_system: "cartesian" | "polar" | "3d", x_range: [xmin, xmax] (LOGICAL data coords, independent of size), y_range: [ymin, ymax] (LOGICAL data coords, independent of size), background: "white" | color }`
- `commands`: ordered list of `{ element_id, func, kwargs, meta? }`
  - `element_id`: globally unique, non-empty, snake_case (e.g. `"main_curve"`, `"y_axis_label"`)
  - `func`: must be one of the primitives registered below
  - `kwargs`: typed parameters, every key must conform to that primitive's schema
  - `meta`: optional dict (description, semantic_tag, z_order, etc.)

 # 2. Five-step workflow - follow it on every task
Step 1 - PARSE the request. List every visual element the user asks for (curves, points, labels, axes, auxiliary lines, arrows, annotations). Note every numeric value, color, and coordinate. If a numeric value is missing, choose one consistent with the task; never invent conflicting values.
Step 2 - CHOOSE the coordinate system.

`"cartesian"` → 2D with x/y axes, rectangles, parabolas, ellipses, labels in plane
`"polar"` → 2D with θ axis, requires elements that take angle/radius inputs
`"3d"` → any task mentioning depth, axes, surfaces, rotations, "view angle", elevation/azimuth, or 3D primitives

Pick `"3d"` whenever the user says "3D", "axes X/Y/Z", "viewing angle", or names any 3D primitive (sphere, cone, cube, helix, plane section, ...).
Step 3 - CHOOSE primitives from the registry. Prefer the FEWEST primitives that can express the request. If two primitives could represent the same element, pick the one whose `kwargs` best match the task's adjectives (e.g. for a labeled circle, prefer `draw_circle` over a parametric curve approximation; for "perpendicular bisector" prefer a single specialized primitive over three `draw_line` hacks.
Step 4 - FILL parameters carefully.

- **Colors**: use the exact CSS/named color the user specified; do not paraphrase.
- **Canvas boundaries & curve domains**: keep ∼10% inward padding from the canvas edges, so curves/points/surfaces never touch the borders. If canvas is $[-7, 7]$, use a sampling domain of $[-6.3, 6.3]$ - never match the canvas boundary exactly.
- **Coordinate fields - when to use which**:
  - 2D function-plot primitives (`draw_function_curve`, `draw_polar_grid`, `draw_step_function`, `draw_vertical_reference_lines`, etc.): use the

primitive's own math-domain fields (`x_min`/`x_max` or `r_max`). Do NOT add separate canvas-positioning `kwargs`; the primitive fills the canvas axes naturally.

- 3D primitives (`draw_surface_3d` and similar): use BOTH
  * `x_min`/`x_max`/`y_min`/`y_max` - math domain of $f(x, y)$, and
  * `x_range`/`y_range` - canvas position (e.g. $[-7, 7] \times [-5, 5]$).

  These two pairs are independent. Without `x_range`/`y_range`, the surface fills the entire canvas; with them, you can place multiple panels.
- Plain 2D primitives (`draw_line`, `draw_point`, `draw_text`, `draw_circle`) take coordinates directly in the canvas coordinate system - no domain or positioning `kwargs`.

- **3D**: set view elevation/azimuth when the user specifies them; otherwise $25° / 45°$.
- **Labels**: include the user's exact text, in the user's specified color and size.
- **Numeric precision**: keep 2–3 decimals (e.g. `0.58`, not `0.5773502691896257`).
- **COMPACT KWARGS** - every primitive schema below declares a `default` for each non-required field. To minimize token waste and keep programs readable, OMIT any `kwarg` whose value equals the field's default. The renderer treats a missing `kwarg` exactly the same as one whose value equals the default (all renderers use `kwargs.get(key, default)` internally), so omitting never breaks rendering or semantic equivalence - it only shrinks the JSON. Only emit a `kwarg` when you explicitly override the default.

  Examples (all valid; outputs render identically):

```
draw_line  {"start": [0,0], "end": [1,1]}
           # color/linewidth/linestyle omitted
           # defaults are "black"/1.0/"-"
draw_point {"center": [1,2], "color": "red"}
           # size/marker omitted (defaults: 5/"o")
draw_text  {"position": [0.5, 0.5], "text": "x"}
           # font/color/align omitted (defaults apply)
```

  Exception: if a field's default is `null` AND you also want `null`, emit `null` explicitly so the intent is self-documenting. Never leave a truly required (no-default) field blank - those must always be present. Note: this rule applies ONLY to `kwarg` values that equal the schema's declared default. Always emit values that override the default, even when they are visually similar (e.g. `linewidth: 1.0` equals the default and should be omitted; `linewidth: 2.5` must be kept).

Step 5 - ASSIGN element_ids. Use descriptive snake_case that reflects the element's role, NOT its order:

good: `"parabola_main"`, `"focus_F"`, `"directrix"`, `"vertex_V"`, `"tangent_at_P"`
bad: `"e1"`, `"elem_3"`, `""` (empty)

IDs must be unique across the document.

 # 3. Common mistakes to avoid

- Coordinate-system mismatch - labelling 3D primitives with `"cartesian"` loses the Z axis.
- Out-of-range coordinates - drawing a point at $(7, 4)$ when `x_range` is $[-3, 3]$ clips it.
- Empty `element_id` - every command MUST have a non-empty `element_id`.
- Duplicate `element_id` - IDs must be globally unique.
- Unknown primitive - only use `funcs` from the registry below; never invent names.

- Missing required `kwargs` - if a field is required (no default), you must provide it.
- Wrong `kwarg` type - strings need quotes (`"red"`), lists need brackets (`[0, 0]`), bools are `true`/`false`, numbers are bare (`3.5`, not `"3.5"`).
- Wrapping the JSON in markdown fences - output must be raw JSON, no ```` ```json ```` blocks.
- Adding prose outside the JSON - no preamble, no postamble, no comments.

 # 3.5 Writing safe math expressions (for `expression`, `x_expr`, `y_expr` ... kwargs)

Some primitives accept a user-supplied math expression that the renderer `eval()`s at draw time, inside a `safe` namespace. Practical guidance:

- Each expression field's `description` enumerates supported names - respect it. Do NOT invent functions that are not listed.
- Avoid bare denominators that hit zero inside the sampling domain (yields `NaN` strips). Prefer `np.arctan2(y, x)` over `np.arctan(y/x)` when both are exposed; otherwise shift the domain, or guard with `+ 1e-12` only if allowed.
- For tasks with poles (e.g. $1/(x-2)$), keep the function accurate - the renderer `NaN`s out singular cells. Do not also shift `x_range` to hide the pole off-canvas.
- Numeric constants: `0.5`, `pi`, `e`, `2*pi` etc. all fine.

 # 4. Self-check before emitting

- `canvas.coordinate_system` matches the task (cartesian / polar / 3d)
- coordinate-field choice matches the primitive type (see Step 4):

  2D function-plot: only the math-domain `kwargs`; no `x_range`/`y_range`.
  3D primitive: math-domain `kwargs` AND canvas-positioning `x_range`/`y_range`.
  plain 2D primitives: only canvas-coord `kwargs`.

- $\sim$10% inward padding between canvas `x_range`/`y_range` and any drawn sampling domain
- every command has a non-empty, unique, snake_case `element_id`
- every `func` is one of the registered primitives below
- every `kwarg` key exists in that primitive's schema, with a valid type and value
- no `kwarg` equals its schema `default` - these are redundant and should be omitted (see COMPACT KWARGS)
- output is a single JSON object (no markdown, no surrounding prose)

 # 5. Registered primitives

`__REGISTRY_TEXT__`

 # 6. Output format

Return ONLY a single JSON object that conforms to the visual program schema (`canvas + commands`). No explanation, no markdown fences, no commentary, no trailing whitespace or code-block markers. The response must START with { and END with }.

Figure 11: Visual program generator for JSON synthesis.

C.2 VLM-AS-JUDGE EVALUATOR

Prompt: VLM-as-Judge Image–Task Consistency Evaluator

You are a strict "image–task consistency" reviewer.

You will receive:

1. A drawing task described in Chinese (English).
2. A PNG image rendered by a visual program.
3. The predicted visual program JSON with every primitive's parameters.

Your task is to judge whether this PNG satisfies the task description.

Evaluation dimensions (consider jointly):

1. **Geometric elements**: are all the points, line segments, polygons, circles, arcs, ticks, annotations etc. mentioned in the task actually present?
2. **Positioning**: are the coordinates roughly accurate (slight visual deviations are allowed)?
3. **Visual attributes**: are color, line style (solid/dashed/dotted), linewidth, fill, and transparency correct?
4. **Labels and text**: do letter annotations, axis labels, legends, force-vector marks etc. appear and sit at reasonable positions?
5. **Overall structure**: is the image complete, with no major missing or extra elements; can the drawing content be seen?

Scoring rules:

- 90–100: fully satisfied; every key element is correct.
- 70–89: mostly satisfied; 1–2 minor elements missing or wrong.
- 50–69: partially satisfied; the main elements essentially exist but with clear defects.
- 30–49: only a few elements satisfied; the main content is missing or seriously wrong.
- 0–29: not satisfied at all; the image is almost unrelated to the task.

Strict requirements:

- Return STRICT JSON only; no explanation, no markdown code fences, no extra text.
- The `pass` field indicates whether the basic bar is met (`>=70` counts as pass).
- `issues` enumerates concrete deficiencies; if none, return an empty array.

Figure 12: VLM-as-judge image-task consistency evaluator.

C.3 PRIMITIVE PROPOSER

Prompt: Step-A Primitive Proposer

Role. You are a *Visual Language Designer* for a typed visual programming system used to compose publication-quality scientific figures. Your task is to inspect the registry and a current task result, then decide whether it warrants **adding** a brand-new primitive, **modifying** one existing primitive, or doing **nothing**. The goal is a compact, capable registry that produces correct figures with fewer primitive calls.

Registry Document Model. A visual program contains:

- `canvas: { size: [w_px, h_px] (PIXELS - not logical coordinates; typical: 800×600, 1200×800. Multi-panel: ≥800px wide. Do NOT set to data-range values like [20, 5].), coordinate_system, x_range, y_range, background }`

- `commands`: ordered list of { `element_id, func, kwargs, meta` }

Every `func` must be a primitive in the global registry and must respect that primitive's typed schema.

Coordinate System Convention.

- `coordinate_system` is metadata only (`cartesian` | `polar` | `3d`); it does not affect rendering.
- 2D primitives (`draw_line`, `draw_text`, `draw_point`, ...) use canvas coordinates.
- 3D primitives (`draw_surface_3d`, ...) use the primitive's own `x_min`/`x_max`/`y_min`/`y_max` domain, independent of canvas.
- 2D function-plot primitives (curves, grids, step graphs, asymptotes, ...) use `x_min`/`x_max` for the math domain; no separate canvas-positioning fields are needed - the curve fills the canvas axes naturally.

Primitive Design Principles. *The eleven-rule design block of Appendix B.7 is interpolated verbatim at this position.* It opens with the governing rule, MECHANISM, NOT POLICY: a primitive draws what it is told and does not decide. It then states single responsibility, the schema-size targets, composition over enumeration and over Boolean flags, YAGNI, the strategy pattern for function families, default handling, open-closed extension, backward compatibility for MODIFY, and naming the mechanism rather than the subject. The refiner prompt below receives the identical block.

Design Goal. Every primitive is judged on two criteria:

- **Rendering Correctness**: does it produce the right visual output for the target task? A primitive that achieves the correct geometry with fewer, simpler `kwargs` is always better.
- **Editorial Efficiency**: how many primitive calls would a user need? The best primitives are general-purpose enough to subsume multiple `draw_line` / `draw_point` constructions.

Allowed Actions. Pick exactly one.

- **ADD**: invent a brand-new primitive. Use only when no existing primitive can be adapted without violating the principles (e.g. it would push some primitive past the 10-field hard cap).
- **MODIFY**: take ONE existing primitive and adjust its schema/renderer to handle this task. Use ONLY when (a) the change is small ($\leq$2 new optional fields, $\leq$10 LoC of new render logic) AND (b) the target primitive's responsibility legitimately covers this case.
- **NONE**: the existing registry already solves this task, or no valid and reusable change is warranted.

Pre-Bound Names Inside Render Bodies.

- `ax` : `matplotlib.axes.Axes`
- `np` : numpy (`import numpy as np`)
- `plt` : `matplotlib.pyplot`
- `kwargs` : the dictionary of named parameters from the visual program

You may also import inside the body: `math`, `matplotlib.patches`, `matplotlib.lines`. You MUST call `kwargs.get(key, default)` for all parameters.

Coordinate Field Contract (Rule CF-1). Coordinates (`start`, `end`, `center`, `position`, `tip`, `origin`, `point`, ...) are ALWAYS expressed as JSON arrays such as `["start", [x, y]]`; they are NEVER strings.

- **Wrong**: `start: str = Field(description='Arrow start coordinate as [x, y].')` (schema says `str`!)

- **Right**: `start: List[float] = Field(description='Arrow start point. JSON array [x, y] in canvas coords.')`
- **In render body**: `start = kwargs.get('start', [0, 0]); ax.plot([start[0]], [start[1]], ...)` (index into list).
- NEVER emit stringified arrays like `"start": "[-2, 3]"` - always use proper JSON array syntax `"start": [-2, 3]`.

Renderer Safety Contract.

- Every expression-type field's `description` MUST list all supported function/constant names.
  - *Good*: `"Expression with x. Supported: sin, cos, tan, exp, log, sqrt, abs, pi, x."`
  - *Bad*: `"A mathematical expression."` (forces callers to guess)
  - *Bad*: `"Expression using np.sin, np.cos. Supported: np.sin, np.cos."` (using `np`-prefixed names requires the `safe` dict to also contain the bare name `np` - prefer unprefixed names).
- Field `description` and the renderer's `safe` dict keys MUST match exactly. If your renderer uses `eval()` on the expression, build a `safe` namespace dict mapping each supported name to a numpy implementation; do NOT rely on bare builtins. Mask non-finite results with `NaN` before plotting.
- Schema/render consistency: the `safe` dict keys MUST be exactly the names appearing in the `eval()` expression. For example, if the default expression is `"sin(x)"` then `safe` must contain `'sin'`; if the default is `"np.sin(x)"` then `safe` must contain `'np'`. NEVER write `safe = dict(np.sin=np.sin, ...)` together with `eval("np.sin(x)", globals, safe)` - the `np` lookup fails first. Prefer unprefixed names (`sin`, `cos`, ...) to keep the `safe` dict simple.

Compositing Renderer Contract (CRITICAL - read carefully).
Background: the layered renderer renders each primitive into its own temporary matplotlib figure, then alpha-blends all layers onto a single canvas. Before each `render_fn` is called, the renderer executes:

```
x_range, y_range = self._get_command_range(cmd, canvas)
ax.set_xlim(x_range)   # canvas math coordinate system
ax.set_ylim(y_range)
ax.axis("off")         # matplotlib set_axis_off
ax.set_position([0, 0, 1, 1])
```

- **Rule CS-1 (Coordinate System)**: the renderer has already set `ax.set_xlim` and `ax.set_ylim`. Your primitive's job is to draw artists using those coordinates - NOT to redefine them. **FORBIDDEN** inside `render_fn`: `ax.set_xlim(...)`, `ax.set_ylim(...)`, `ax.set_aspect(...)`. These override the canvas coordinate system and break alignment in the final alpha-composited image. All coordinate data you need (`x_min`, `x_max`, ...) is already available in `kwargs` - use those values in `plot`/`scatter`/`text` calls directly.
- **Rule CS-2 (Axison)**: `ax.axis("off")` (`axison=False`) silently skips every axis decoration, and calling `set_visible(True)` afterwards cannot rescue it. Specifically, the following are completely invisible in the final composite: `ax.spines['left'].set_position/set_color/set_visible(True)`; `ax.spines['bottom']` likewise; `ax.grid(True, ...)`; `ax.set_xticks / set_yticks / set_xticklabels / set_yticklabels`; `ax.set_xlabel / set_ylabel / set_title`; `ax.tick_params(...)`; `ax.set_xlim / set_ylim / set_aspect`.

- **What DOES render** (independent `Line2D / Patch / Text` artists, not bound to axison): `ax.plot(...); ax.axvline(...); ax.axhline(...); ax.scatter(...); ax.text(...); ax.fill_between(...); ax.add_patch(...)` (Rectangle, Circle, Polygon, ...); `ax.arrow(...); ax.annotate(...)`.
- **Rule CS-3**: every primitive that needs axes, grid, origin cross, or tick marks MUST use `axvline / axhline / plot / add_patch`. NEVER use `ax.spines[...], ax.grid(...), ax.set_xticks(...)`, or any `set_xlim / set_ylim / set_aspect`.

Concrete patterns:

- **(a) Drawing the x=0 / y=0 main axes (replacement for `ax.spines` positioning):** `ax.axvline(0, color=axis_color, linewidth=axis_linewidth, zorder=...); ax.axhline(0, color=axis_color, linewidth=axis_linewidth, zorder=...)` - NOT `ax.spines['left'].set_position(('data', 0))`.
- **(b) Drawing a uniform grid (replacement for `ax.grid(True, ...)`):** `for x in xs: ax.axvline(x, color=grid_color, linewidth=grid_linewidth, alpha=alpha)` and similarly for `y` - NOT `ax.grid(True, color=..., linewidth=...)`.
- **(c) Drawing tick labels (since `ax.set_xticklabels` is ignored):** `for x in xs: ax.text(x, label_offset, str(x), ha='center', va='top', fontsize=...)`.
- **(d) Suppressing the default frame:** `for name in ('top', 'right', 'left', 'bottom'): ax.spines[name].set_visible(False)` (note: `set_visible(False)` is allowed but has no effect under `axison=False`; the key rule is never to rely on `set_visible(True)` or `set_position` for visibility).

Returning an Artist. The `render_fn` MUST return a matplotlib Artist (`Line2D`, `Patch`, `Text`, ...). Capture the return value of `plot / axvline / axhline` and return it. Without `return`, the downstream composite logic cannot track the artist and silent fall-back rendering issues may occur.

Inputs.

- Task description: {`TASK_PROMPT`}
- Predicted visual program: {`PREDICTED_PROGRAM`}
- Rendered PNG or failure note: {`RENDERED_PNG_OR_FAILURE`}
- Registry schema: {`REGISTRY_SCHEMA`}
- Provenance: {`PROVENANCE`}

Output Requirements (must be strictly followed).
Reply ONLY with a single JSON object. Always emit every top-level key. For ADD, fill `new_primitive` and set the MODIFY fields to `null`. For MODIFY, set `new_primitive` to `null` and fill the MODIFY fields. For NONE, set all action-specific fields to `null`.

```
{
  "analysis": "1--3 sentences describing the observed gap and chosen action",
  "action": "ADD | MODIFY | NONE",
  "new_primitive": {
    "name": "draw_<snake_case>",
    "family": "scientific | geometric | data_viz | domain_specific",
    "description": "one visual responsibility",
```

```
    "schema_dict": {"fields": [
      {"name": "<field>", "type": "string|number|integer|bool|array[number]",
       "required": false,
       "description": "field semantics and constraints",
       "default": <value>}
      ]},
    "editable_fields": ["<fields renderer reads>"],
    "rationale": "which principle supports ADD, why MODIFY
won't work",
    "provenance_tasks": ["<task_id>"],
    "smoke_test_kwargs": {"<field>": <value>},
    "render_code": "kwargs line 1\nkwargs line 2\n..."
   },
  "modify_target": "<exact primitive name>",
  "modify_hint": "Specific change needed (≤2 new fields, ≤10
LoC)",
  "modify_principle": "which principle permits this (e.g. OCP:
refinement of fill mode)"
}
```

Figure 13: Step-A primitive proposer for the dynamic registry.

C.4 Primitive Refiner

Prompt: Step-B MODIFY-Only Primitive Refiner

Role. You are refining a **MODIFY** action on an existing primitive. You are given a single target primitive together with the triggering task, its current schema, and its current render code. Produce a new version of this primitive that addresses the task without growing the schema past 10 fields or 2 Boolean fields (aim for ≤8 fields) and without changing the primitive's core responsibility.

Coordinate System Convention.

- `coordinate_system` in canvas is metadata only; it does NOT affect rendering.
- 2D primitives use canvas coordinates; 3D primitives (`draw_surface_3d`, ...) use their own `x_min`/`x_max`/`y_min`/`y_max` domain independent of canvas.
- 2D function-plot primitives use `x_min`/`x_max` for the math domain - no separate canvas-positioning fields are needed; the curve fills the canvas axes naturally.

Renderer Body Skeleton. The renderer body runs inside:

```
def render_<name>(ax, **kwargs):
    # your code here, must return a matplotlib Artist
    return artist
```

Pre-Bound Names Inside Render Bodies.

- `ax` : `matplotlib.axes.Axes`
- `np` : numpy
- `plt` : `matplotlib.pyplot`
- `kwargs` : dict of named parameters

You may also import inside the body: `math`, `numpy as np`, `matplotlib.pyplot as plt`, `matplotlib.patches`, `matplotlib.lines`. You MUST call `kwargs.get(key, default)` for all parameters.

Coordinate Field Contract (Rule CF-1). Coordinates (`start`, `end`, `center`, `position`, `tip`, `origin`, `point`, ...) are ALWAYS expressed as JSON arrays: `__START_ARRAY__` or `__END_ARRAY__`. They are NEVER strings.

- **Wrong**: `start: str = Field(description='Arrow start coordinate as [x, y].')` with `ax.arrow(start[0], start[1], ...)` (TypeError when JSON has `"start": "[-2,3]"`).
- **Right**: `start: List[float] = Field(description='Arrow start point. JSON array [x, y] in canvas coords.')` with `start = kwargs.get('start', [0, 0]); ax.arrow(start[0], start[1], ...)`.
- NEVER emit stringified arrays like `"start": "[-2, 3]"` - always use proper JSON array syntax.

Renderer Safety Contract.

- Every expression-type field's `description` MUST list all supported names.
  - *Good*: `"Expression with x. Supported: sin, cos, tan, exp, log, sqrt, abs, pi, x."`
  - *Bad*: `"Expression with x. Supported: np.sin, np.cos, np.exp, np.pi, x."` (forces `np.sin(x)` which requires `np` in safe dict - prefer unprefixed names).
  - *Bad*: `"A mathematical expression."` (forces callers to guess).
- Field `description` and the renderer's `safe` dict keys MUST match exactly. If the renderer uses `eval()` on the expression, build a `safe` namespace dict mapping each supported name to a numpy implementation; do NOT rely on bare builtins. Mask non-finite results with `NaN` before plotting.
- Schema/render consistency: the `safe` dict keys MUST be exactly the names appearing in the `eval()` expression. For example, if the default expression is `"sin(x)"` then `safe` must contain `'sin'`; if the default is `"np.sin(x)"` then `safe` must contain `'np'`. NEVER write `safe = dict(np.sin=np.sin, ...)` together with `eval("np.sin(x)", globals, safe)` - the `np` lookup fails first. Always prefer unprefixed names (`sin`, `cos`, ...) to keep the `safe` dict simple.

Compositing Renderer Contract (CRITICAL - applies to MODIFY).
Background: the layered renderer renders each primitive into its own temporary matplotlib figure, then alpha-blends layers onto a single canvas. Before each `render_fn` is called, the renderer executes:

```
x_range, y_range = self._get_command_range(cmd, canvas)
ax.set_xlim(x_range)   # canvas math coordinate system
ax.set_ylim(y_range)
ax.axis("off")         # matplotlib set_axis_off
ax.set_position([0, 0, 1, 1])
```

- **Rule CS-1 (Coordinate System)**: the renderer has already set `ax.set_xlim` and `ax.set_ylim`. Your primitive's job is to draw artists using those coordinates - NOT to redefine them. **FORBIDDEN** inside `render_fn`: `ax.set_xlim(...)`, `ax.set_ylim(...)`, `ax.set_aspect(...)`. These override the canvas coordinate system and break alignment in the final alpha-composited image. All coordinate data you need (`x_min`, `x_max`, ...) is already available in `kwargs` - use those values directly. (This was the root cause of `function_L2_038`: `draw_function_curve` called `ax.set_xlim(0, 4.5)` on a figure sized for canvas coords $(-5, 5)$, and the red curve $(x+2)$ visibly missed the red point at $(0, 2)$.)
- **Rule CS-2 (Axison)**: `ax.axis("off")` (`axison=False`) silently skips every axis decoration; calling `set_axis_on()` or `set_visible(True)` afterwards cannot rescue it. The following are completely invisible in the final composite: `ax.spines[...].set_position / set_color / set_visible(True)`; `ax.grid(True, ...)`; `ax.set_xticks / set_yticks`

/ set_xticklabels / set_yticklabels; ax.set_xlabel / set_ylabel / set_title; ax.tick_params(...); ax.set_xlim / set_ylim / set_aspect.

- **What DOES render**: ax.plot(...); ax.axvline(...); ax.axhline(...); ax.scatter(...); ax.text(...); ax.annotate(...); ax.fill_between(...); ax.add_patch(...) (Rectangle, Circle, Polygon, ...); ax.arrow(...).
- **Rule CS-3**: every primitive that needs axes, grid, origin cross, or tick marks MUST use axvline / axhline / plot / add_patch. NEVER use ax.spines[...], ax.grid(...), ax.set_xticks(...), or any set_xlim / set_ylim / set_aspect.

When MODIFYing a primitive that draws axes / grid / origin cross / tick marks, REWRITE the offending lines to use axvline / axhline / plot / text:

Before (broken under axison=False):

```
ax.spines['left'].set_position(('data', 0))
ax.spines['bottom'].set_position(('data', 0))
ax.spines['left'].set_color(axis_color)
ax.spines['bottom'].set_color(axis_color)
ax.grid(True, color=grid_color, linewidth=grid_linewidth)
```

After (renders correctly):

```
ax.axvline(0, color=axis_color, linewidth=axis_linewidth, zorder=z)
ax.axhline(0, color=axis_color, linewidth=axis_linewidth, zorder=z)
for x in xs:
    ax.axvline(x, color=grid_color, linewidth=grid_linewidth,
               alpha=alpha, zorder=z)
for y in ys:
    ax.axhline(y, color=grid_color, linewidth=grid_linewidth,
               alpha=alpha, zorder=z)
```

Returning an Artist. Returning a matplotlib Artist (Line2D, Patch, Text, ...) is mandatory. Without a return value, the downstream composite loses the artist handle. Capture the return of axvline / axhline / plot and return it.

Primitive Design Principles. *The same design block interpolated into the proposer prompt is interpolated verbatim at this position* (Appendix B.7). Rule 9, backward compatibility, and rule 0, MECHANISM, NOT POLICY, are the two that bind a MODIFY: new fields must default to the previous behavior, and a refinement may not move a layout decision from the caller into the primitive.

Inputs.

- The triggering task is: __TASK_PROMPT__
- The existing primitive being modified is: __TARGET_NAME__
- Current schema:

  __SCHEMA_TEXT__

- Current render code:

  __OLD_RENDER_CODE__

Output Requirements (must be strictly followed).

Return ONLY a single JSON object. The four lines below show the EXACT output shape you must produce - copy the structure and substitute the primitive name from __TARGET_NAME__ (which is given as a plain string earlier in this prompt):

```
{
  "name": "<__TARGET_NAME__>",
  "family": "scientific | geometric | data_viz | domain_-
specific",
  "description": "1--2 sentences - responsibility (unchanged or
clarified)",
  "schema_dict": {"fields": [
    {"name": "<field>", "type": "...|array[number]",
"required": false,
     "description": "If math expression: list supported
names (e.g. sin, cos, tan, exp, log, sqrt, abs, pi, x). For
coordinate/array fields: write JSON array syntax [x, y] (NOT a
string). Otherwise describe normally.",
     "default": <value>}
    ]},
  "editable_fields": ["<fields renderer reads>"],
  "rationale": "which principle authorises this MODIFY; how
many new fields / LoC",
  "principle_check": "Confirm: schema \leq 8 fields (hard cap
10), \leq 2 bools, responsibility unchanged, every expression
field description lists supported names, coordinate/array
fields use JSON array syntax [x, y] NOT strings (cf. Rule
CF-1), the safe dict keys match the names used in the eval()
expression exactly (prefer unprefixed names like sin/cos - if
using np-prefixed names, the safe dict MUST contain the bare
name np). COORDINATE SYSTEM (Rule CS-1): render_fn does NOT
call set_xlim/set_ylim/set_aspect.",
  "provenance_tasks": ["__TASK_ID__"],
  "smoke_test_kwargs": {"<field>": <value>},
  "render_code": "kwargs line 1\nkwargs line 2\n..."
}
```

Concrete Example with `target_name = draw_line`. Emit EXACTLY this (the `name` is the LITERAL string `draw_line`, not `__TARGET_NAME__`, not `draw_line_v2`):

```
{
  "name": "draw_line",
  "family": "geometric",
  "description": "Draws a straight line segment between two
canvas-coordinate points.",
  "schema_dict": { "fields": [ { "name": "start", "type":
"array[number]", "required": false, "description": "Start
point [x, y].", "default": [0, 0] } ] },
  "editable_fields": ["start"],
  "rationale": "example",
  "principle_check": "example",
  "provenance_tasks": ["function_L1_003"],
  "smoke_test_kwargs": { "start": [0, 0] },
  "render_code": "start = kwargs.get('start', [0, 0])\nartist,
= ax.plot([start[0]], [start[1]])\nreturn artist"
}
```

Warning - Absolute Rule for the `name` Field. The `name` JSON value MUST be the EXACT primitive name `__TARGET_NAME__` - copy it verbatim.

- **WRONG literal strings that have appeared in past LLM outputs (each corrupts the registry):** `__TARGET_NAME__` (unresolved placeholder), `target_name` (bare word), `draw_line_v2` (MODIFY must REPLACE, not duplicate), `draw_line_new` (similar v2 / new variant).

- **RIGHT**: emit the resolved primitive name as a plain JSON string, e.g. `draw_line` (the resolved value of `__TARGET_NAME__`, copied verbatim).
- A wrong `name` causes MODIFY to silently register a NEW primitive instead of replacing the original - corrupting the registry.

Prefer `"required": false` with `"default"` values.

Key Rules.

- **Backward compatibility**: previous smoke-test kwargs MUST still render sensibly.
- **No new responsibility**: do NOT add a new responsibility; if needed, abandon MODIFY and propose ADD.
- **Default preservation**: new fields must default to reproduce the OLD behaviour.
- **Expression names**: update field `description` AND `safe` dict in lockstep.
- **Coordinate Field (Rule CF-1)**: coordinate/array fields (`start`, `end`, `center`, `position`, `tip`, ...) MUST use JSON array syntax `[x, y]` in the JSON output - NEVER stringified arrays like `"[-2, 3]"`. The schema type should reflect this (`List[float]`, `array[number]`, ...), not `str`.
- **Coordinate System (Rule CS-1)**: `render_fn` must NEVER call `set_xlim`, `set_ylim`, or `set_aspect` - the renderer already sets the canvas math coordinate system before `render_fn` runs. Calling these rescaling the data-to-pixel mapping and breaks alignment in the final alpha-composited image.
- **Axes Rendering (Rule CS-2/CS-3)**: any MODIFY that touches axes / grid / origin cross / tick-mark rendering MUST replace `ax.spines[...]`, `ax.grid(...)`, `ax.set_xticks/yticks/xticklabels/yticklabels`, `ax.set_xlabel`, `ax.set_ylabel`, `ax.set_title` with `ax.axvline / ax.axhline / ax.plot / ax.text / ax.add_patch`. Returning a matplotlib Artist is mandatory.

Design Goal. Judge on RENDERING CORRECTNESS and EDITORIAL EFFICIENCY with fewer primitive calls. Be conservative. Only modify primitives that genuinely need a tightly scoped change.

Figure 14: Step-B MODIFY-only primitive refiner for the dynamic registry.

D Detailed Related Work

This section provides an expanded contextualization of Babel relative to prior work in visual program generation, program abstraction synthesis, and self-evolving agent architectures.

Visual Program Generation with Predefined Languages. Generating programs to produce or reason about visual artifacts has long relied on predefined representation formats. Existing pipelines predominantly prompt or fine-tune models to emit code in standard vector formats such as SVG (20; 41; 44; 29; 39), LaTeX-based TikZ (16; 6), declarative visualization grammar like Vega-Lite (34), or imperative Python plotting packages such as Matplotlib (28; 11; 45) and 3D web visualizers like Viser (47). Other efforts design specialized, human-curated visual DSLs for compositional reasoning (17; 36). Across all these approaches, the vocabulary of primitives is treated as an immutable, *a priori* design constraint. Consequently, when confronted with multi-domain shifts or dense structural patterns, fixed languages either suffer syntactic bloat or fail to express novel visual semantics. Babel removes this constraint by treating the visual language itself as a dynamic variable to be grown and optimized from a minimal seed basis.

Library Learning and Abstraction Discovery. Library learning methods aim to automate vocabulary creation by identifying and compressing common programmatic subroutines. Foundational systems such as DreamCoder (13), LAPS (12), Stitch (7), and Babble (9) mine repetitive syntax subtrees from corpus programs to compress search spaces and construct hierarchical primitive libraries. Recent neuro-symbolic systems such as LILO (14) and Voyager (40) integrate foundation models for

semantic search and open-ended embodied skill accumulation. Unlike traditional library learning, which compresses syntactic redundancy retrospectively across program corpora, Babel generates new primitives prospectively in response to execution tracebacks and task expressiveness gaps, analyzing whether the resulting abstractions settle on domain-specific policies or domain-general drawing mechanisms.

Tool Use, Creation, and Self-Evolving Agents. Expanding LLM capabilities via tool synthesis is an active frontier. Early systems like LATM (8), Code as Policies (23), CREATOR (31), and TroVE (42) explore dynamic, problem-dependent tool generation; however, these mechanisms are largely ephemeral, generating ad-hoc utilities that are discarded once the immediate task concludes. More recently, self-evolving agent architectures and adaptive meta-harness systems (22; 49; 10; 37; 48; 50) have demonstrated the benefits of cumulative, cross-task skill consolidation. Babel brings this lifelong accumulation to the visual domain: rather than maintaining a disconnected bag of isolated helper scripts, Babel validates, filters, and accumulates primitives into a typed, unified registry that functions as a structured, composable visual DSL.

E LIMITATIONS

Both efficiency baselines are contestable, in opposite directions. Human-28's weakness admits two readings, that policy-shaped primitives are harmful or that we built a poor DSL; we report its failure pattern (§5.2) so the reader can judge, and the argument rests on Seed-4 instead. The external languages qualify rather than confirm: SVG needs 2.19× more commands with an identical unit, while TikZ is comparable in length and its whole advantage sits in the 39 tasks calling `pgfplots`. Python is bracketed for the same reason, its proxies differing by 4.7×. The judge also sees primitive names, so it is not blind to the language, though a second judge reverses no ordering we assert.

Gating is unsolved, and the scope is narrow. A candidate is admitted for being well-formed and safe, never for improving a rendering; Appendix B.12 reports two attempts to close this and neither settles it. The seed basis is an initialization rather than an invariant, three of the four seeds having been rewritten in several arms, and 2,350 of 5,445 English prompts, 43.2%, contain Chinese label literals, so that split is not usable standalone. All results are scientific figures in six domains under one judge family.