Lyra: Lifelong Chart-to-Code Generation via LLM-Driven Multimodal Data Synthesis

Anonymous ACL submission

Abstract

Current vision-language models struggle to parse the dense visual content in charts. The chart-to-code task addresses this by converting visual data into executable rendering code. Previous approaches rely on static and synthetic datasets, but these collections often lack diversity in chart types and underlying data distributions. This limitation contrasts with real-world charts that evolve continuously across domains. To bridge this gap, we propose Lyra, a fully automated lifelong learning framework for chart-to-code generation. We employ a large language model as a strategic data generator that generates chart-code pairs to train the vision-language model. The vision-language model assesses these samples to identify challenging instances and guides the language model to optimize its generation strategy while simultaneously refining its own charting capabilities through training on the synthesized data. Our experiments demonstrate that Lyra progressively improves performance across three out-of-domain benchmarks while successfully mitigating catastrophic forgetting and achieving superior sample efficiency compared to prior work.

1 Introduction

Large Multimodal Models (LMMs) have demonstrated strong capabilities in general visual understanding (Yue et al., 2025; Lu et al., 2024). However, accurately interpreting the dense visual content and structural relationships within charts remains a persistent challenge (Masry et al., 2022). Charts are critical for communicating quantitative information, but require models to perform precise perception of fine-grained elements and complex multi-step reasoning.

To address this, recent approaches (Yang et al., 2024; Wu et al., 2025) decompose chart reasoning into two distinct stages comprising chart-to-code translation and subsequent code reasoning. In this paradigm, the multimodal model focuses exclusively on translating the visual chart into a code representation while a language model performs complex reasoning over the code to derive answers. This separation allows systems to exploit the robust code reasoning capabilities of modern LLMs. Previous studies primarily rely on static datasets crawled from the web or generated by proprietary language models (Xia et al., 2025; Zhao et al., 2025). However, these static collections fail to capture the continuous evolution of real-world visualization styles and domains. Consequently, models trained on fixed corpora struggle to generalize to novel chart types or underlying data distributions. Moreover, existing machine-generated data ignores the current state of the model on the chart-to-code task. This leads to an inefficient training data distribution where some examples are trivial and redundant while others remain too challenging. We therefore raise the following research question:

> *How can we continually improve the chart-to-code capabilities of multimodal models as data and visualization styles evolve over time?*

We answer this by proposing Lyra, a framework for fully automated lifelong chart-to-code learning. Lyra establishes a dynamic teacher-student loop. A large language model serves as the teacher that devises improvement plans and generates training data, while a vision-language model acts as the student that learns to generate code from the given chart image. Unlike prior work targeting imperative libraries like Matplotlib, our student predicts Vega-Lite JSON specifications. We utilize Vega-Lite because its declarative grammar explicitly encodes visualization priors, which significantly reduces the output search space and simplifies the learning objective. The core of Lyra lies in its ability to adaptively regulate training based on the student's current performance. Specifically, the

teacher generates diverse and complex charting tasks that target instances where the student previously struggled. To identify these failure cases efficiently, we introduce SpecSim, a metric that evaluates the syntactic and semantic alignment of Vega-Lite specifications without the prohibitive computational costs associated with VLM-as-a-Judge. For lifelong learning, we employ Prioritized Experience Replay (PER) (Schaul et al., 2015) to retain previously acquired skills and mitigate catastrophic forgetting.

We evaluated Lyra on three out-of-domain benchmarks comprising Plot2Code, ChartX, and ChartMimic. Experimental results demonstrate that Lyra progressively improves chart-to-code performance and exhibits better sample efficiency compared to baselines. Furthermore, our analysis reveals that expanding the replay memory more frequently with fewer training examples per round leads to better performance than less frequent, larger per-step data volumes. This finding validates the effectiveness of the language model-guided curriculum in driving continual improvement. As the first framework to apply lifelong learning to the chart-to-code task using declarative grammar, we believe Lyra offers a robust foundation for future research in adaptive multimodal systems.

2 Related Work

2.1 Lifelong Learning in LLM

Lifelong learning addresses the challenge of knowledge retention without catastrophic forgetting (Zheng et al., 2025b,a; Chen and Liu, 2018). While classic approaches utilize rehearsal and regularization (Rolnick et al., 2019; Kirkpatrick et al., 2017; Mallya and Lazebnik, 2018), recent work extends these methods to LLMs for tasks like instruction following and knowledge editing (Jin et al., 2022; Luo et al., 2025; Jang et al., 2021). Our work specifically aligns with LLM-based agents in dynamic environments where the model functions as a cognitive core (Xi et al., 2025). These agents employ modular architectures that integrate perception for multimodal grounding (Zhou et al., 2023; Jaegle et al., 2021), hierarchical memory for storing interaction histories (Chevalier et al., 2023; Vitter, 1985; Qin et al., 2023), and action modules for reasoning (Sumers et al., 2023; Yao et al., 2022). In this context, lifelong learning occurs through continuous memory updates and policy refinements to adapt to evolving multimodal tasks.

2.2 LLM-Driven Data Synthesis

LLMs have emerged as effective tools for synthetic data generation due to their broad world knowledge and instruction-following capabilities (Long et al., 2024; Ding et al., 2023a; Eldan and Li, 2023). These properties allow models to produce data that mimics real-world distributions while targeting specific formats or difficulty levels. Previous studies utilize prompt engineering (Liu et al., 2024b; Gunasekar et al., 2023) and multi-step generation pipelines (Ding et al., 2023b; Honovich et al., 2023) to generate faithful natural language data. Recent studies extend these capabilities to generate multimodal data using text-based language models. ReachQA (He et al., 2025) synthesizes chart rendering code and image pairs with self-instruction and evolutionary strategies starting from seed code in the Matplotlib gallery. ChartCoder-160k (Zhao et al., 2025) adopts a similar approach using GPT-4 to generate a large-scale dataset of approximately 160k samples. These approaches demonstrate significant performance improvements when models are trained on their synthetic data. However, their data covers a comparatively narrow range of charting tasks with unstructured difficulty levels because they do not account for the current proficiency of the vision-language model. Our proposed Lyra ensures that generated data is grounded in the current state of the vision-language model and targets diverse chart styles and previously unseen data.

3 Proposed Method

In this section, we introduce Lyra, an LLM-driven data synthesis and training framework for fully automated lifelong learning of chart-to-code task.

3.1 Lyra

Lyra adopts a teacher-student framework to enhance chart-to-code generation. A Large Language Model acts as the teacher to design a curriculum based on the current capabilities of the student. It subsequently generates synthetic training data aligned with this plan. The Vision-Language Model functions as the student. It learns the task while providing feedback regarding the difficulty of specific samples. We iteratively generate data and finetune the Vision-Language Model using this guidance. Since iterative finetuning is prone to catastrophic forgetting, we use replay memory (Lin, 1992) and prioritized experience replay (PER) (Schaul et al., 2015). Specifically, Lyra

Algorithm 1: `Lyra`, Lifelong Chart-to-Code Learning

Input: VLM π_0, LLM ϕ, initial replay memory $M_0 = \{(d_i, s_i)\}_{i=1}^{|M_0|}$, $d_i = (I_i, C_i, T_i)$, total iterations R, number of training examples N_{tr}.

Output: Trained VLM π_R, replay memory M_R.

for $t = 0, 1, \ldots, R-1$ **do**

$M_t \leftarrow \texttt{Add}(M_{t-1}, \phi)$;

$M_t \leftarrow \texttt{Update}(M_t, \pi_{t-1})$;

`// Sample training subset using prioritized replay`

$\mathcal{I}_t^{train} \sim P(i \mid M_t)$ with $|\mathcal{I}_t^{train}| = N_{tr}$;

$\mathcal{D}_t^{train} \leftarrow \{d_i : (d_i, s_i) \in M_t,\ i \in \mathcal{I}_t^{train}\}$;

$\pi_t \leftarrow \arg\min_\pi\ \mathcal{L}(\pi_{t-1}; \mathcal{D}_t^{train})$;

$M_t \leftarrow \texttt{Update}(M_t, \pi_t,)$

return (π_R, M_R).

stores all generated data points $d_i = (I_i, C_i, T_i)$ in a replay memory M, together with their scores s_i. Here, I_i is a chart image, C_i is the corresponding Vega-Lite specification, and T_i is a one sentence description of what I_i depicts. We choose Vega-Lite as our code representation because its declarative grammar significantly reduces the complexity of the code generation task compared to imperative plotting libraries, as detailed in Appendix A.1. The score s_i reflects the vision-language model's chart-to-code performance on that data point and can be updated at different stages. At every iteration of vision-language model training, we sample N_{tr} training examples from the replay memory with probabilities defined over the inverse ranks of their scores in M.

Concretely, we define the probability of sampling the i-th example as,

$$P(i|M) = (1-\epsilon)\frac{r_i^{-\alpha}}{\sum_{k=1}^{|M|} r_k^{-\alpha}} + \epsilon\frac{1}{|M|}, \quad (1)$$

where r_i is the rank of the i-th sample in memory M, α and ϵ are smoothing parameters of the probability distribution. Since examples with lower ranks are more likely to be sampled, the vision-language model is encouraged to learn from its previous failures. At the same time, high-ranked examples retain a non-zero probability of being sampled, so the model is still regularly exposed to examples it previously succeeded on. We provide pseudocode for `Lyra` in Algorithm 1 to support a more detailed understanding of the overall process. In the following sections, we introduce two key memory operations, Add and Update, which expand and maintain the replay memory to enable lifelong learning of the chart-to-code task.

3.2 Add Operation

The Add operation introduces new entries into the replay memory. It consists of two sequential processes: task generation and Vega-Lite specification generation. We present the pseudocode of the Add operation in Algorithm 2.

Task Generation Voyager (Wang et al., 2023) has demonstrated that large language model agents can design plans to progressively improve the skills required to solve a task by considering their current state. Inspired by this, we generate new tasks by first sampling easy-to-translate and difficult-to-translate examples from the memory at time t, M_t. We categorize an example as easy if its score exceeds a threshold τ and as difficult otherwise. The threshold τ is set either to a fixed value or to a statistic of the scores in M_t, such as the mean or median. We then partition the memory into two disjoint sets, M_t^{easy} and M_t^{hard}, such that $s_i > \tau$ for $d_i \in M_t^{\text{easy}}$ and $s_i \leq \tau$ for $d_i \in M_t^{\text{hard}}$. We sample hard tasks from M_t^{hard} with probabilities defined in Equation 1. For easy tasks, we rank examples in descending order of their scores so that higher-scored examples are more likely to be sampled. Here, our scheme suppresses sampling near the partition boundary. Examples in this region often provide ambiguous signals, as they are not clearly classified as easy or hard. After sampling easy and hard examples, we feed them to the large language model together with the task-generation prompt P_T shown in Figure 9. The output of the

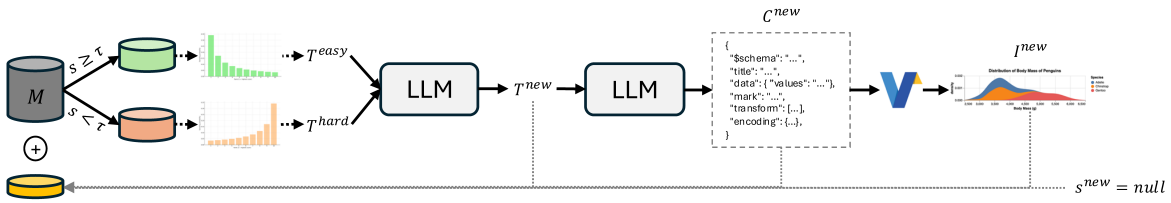

Figure 1: Overview of the Add operation. This operation introduces new entries into the replay memory.

language model is a new task and its accompanying reasoning, which explains why training and evaluating the model on this task is necessary for mastering the chart-to-code translation task.

Vega-Lite Specification Generation The newly generated tasks from the task generator are not yet paired with actual chart images and Vega-Lite specifications. Therefore, we convert each task into a Vega-Lite specification and subsequently render it to obtain a chart image. We pass the generated tasks and their associated reasoning from the task generator as input to the large language model with the Vega-Lite specification generation prompt P_S in Figure 8. When generating Vega-Lite specifications, we impose several constraints motivated by empirical observations. For example, we respect Vega-Lite defaults to leverage its built-in prior knowledge, and we generate synthetic datasets instead of downloading presets from the Vega-Lite ecosystem to ensure diversity in the replay memory. The output of the language model is a Vega-Lite JSON specification, which we convert into a chart image using the Vega rendering engine. We then store the triple (I, C, T) into the replay memory with a *null*-valued score and denote the replay memory after adding new entries as M_{t+1}.

3.3 Update Operation

The Update operation refreshes the scores of entries stored in the replay memory. We update an entry when its score is *null*-valued or when it has been used to train the vision-language model. Similar to the Add operation, Update consists of two subsequent processes: chart-to-code translation and evaluation. We present the pseudocode of the Update operation in Algorithm 3.

Chart-to-Code Translation This task is the only process in `Lyra` in which the vision-language model participates directly, and it is also the primary objective of lifelong learning. The vision-language model receives a chart image together with the Vega-Lite specification generation prompt P_G in Figure 7 as input and outputs a JSON specification. Although both P_S and P_G produce JSON specifications, their inputs and purposes differ. While P_S is designed to expand the replay memory by generating diverse and task-consistent examples, P_G aims to generate code that perfectly reproduces a given chart image.

Evaluation Recent studies have used a vision-language model-as-judge (VLM-as-judge) as a primary similarity metric, since it can capture semantic similarity between images that conventional metrics such as NRMSE and SSIM cannot (Xia et al., 2025; Wu et al., 2025). However, VLM-as-judge is computationally expensive because it relies on large models, making it a major bottleneck if used for every Update operation. Unlike test benchmarks for chart-to-code, our data generation pipeline produces not only chart images but also ground-truth Vega-Lite JSON specifications. This enables a lightweight, model-free metric, SpecSim, that measures both syntactic and semantic similarity between two Vega-Lite specifications. For the syntactic component, we simply flatten each JSON specification into a string and measure the edit distance between the two strings. For the semantic component, we decompose each specification into a small set of visualization components, mark, encoding, data, and transform, and compute a component-wise similarity score. Further details of how SpecSim computes semantic similarity are provided in Appendix A.2. The overall SpecSim score S is a weighted average of syntactic and semantic similarity,

$$S(s_1, s_2) = \gamma \cdot S_{syn}(s_1, s_2) + (1 - \gamma) \cdot S_{sem}(s_1, s_2),$$

where the semantic term itself is a weighted sum of the component scores. This yields a bounded score in $[0, 1]$ that closely tracks semantic similarity between specifications while remaining lightweight enough to be used for every Update operation. For further details, we present the pseudocode of SpecSim in Algorithm 4.

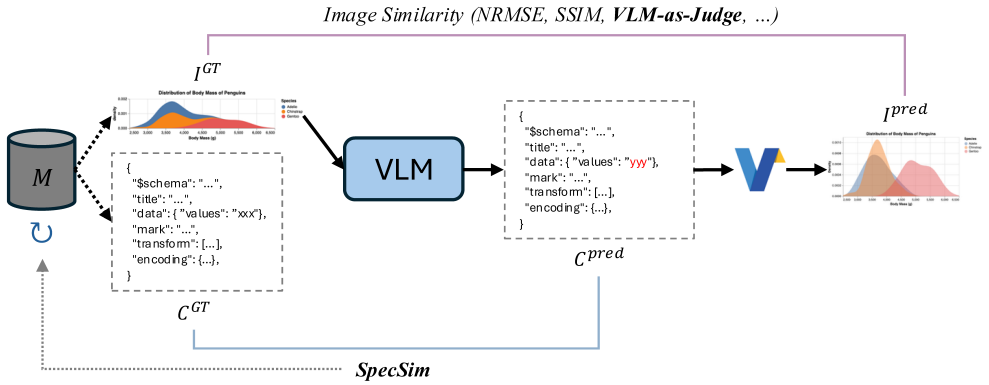

Figure 2: Overview of the Update operation. This operation refreshes the scores of entries stored in the replay memory.

4 Experiments

4.1 Experimental Setup

Data Although Lyra can generate training data without human intervention, we observed that initializing the replay memory with high-quality, human-crafted data improves the diversity of generated samples in subsequent iterations. To this end, we collected official Vega-Lite examples and filtered out near-duplicate charts. Specifically, we first computed the NRMSE and SpecSim S between all pairs of Vega-Lite specifications in the collection. We then manually inspected chart images forming near-similar pairs, defined as NRMSE > 0.9 or $S > 0.9$, and remained only the chart image with the simplest specification. We also excluded interactive charts because we do not consider interaction in our evaluation metrics. This filtering process yielded 437 distinct pairs of Vega-Lite JSON specifications and images which served as seed entries for the replay memory.

To assess out-of-domain generalization, we evaluated performance on three chart-to-code datasets comprising Plot2Code, ChartMimic, and ChartX. Plot2Code (Wu et al., 2025) is a manually filtered collection of 368 scientific plots categorized into six types such as Pie, Statistics, and Contour. ChartMimic (Yang et al., 2024) consists of 600 figures from arXiv that were manually filtered and annotated. ChartX (Xia et al., 2025) contains 1,125 machine-generated chart–code pairs that emphasize chart diversity despite a prevalence of bar charts. We utilized NRMSE, SSIM, and VLM-as-judge as evaluation metrics and adopted the same judge prompt proposed in ChartMimic.

Training and Inference For main experiments, we finetuned three vision-language models ranging from 1B to 9B parameters comprising InternVL-3.5 1B (Wang et al., 2025), Qwen2.5-VL-8B (Bai et al., 2025), and MiniCPM-V4.5 9B (Yu et al., 2025). For all subsequent analyses, we utilized InternVL-3.5 1B only. We executed 20 iterations of the data synthesis and training loop. At each iteration, we added 2,048 new entries to the replay memory and sampled 2,048 training examples. We set $\alpha = 0.7$ and $\epsilon = 0.1$ for all rank-based sampling. We employed the Adam optimizer with a learning rate of 1×10^{-6} and a cosine decay schedule with 0.1 epochs of warmup. The batch size was set to 16 with training lasting for 1 epoch per iteration. The Qwen3-Next-80B-A3B mixture-of-experts (MoE) model (Yang et al., 2025) served as the data generator while the GLM-4.5V-102B-A10B MoE model (Hong et al., 2025) functioned as the judge. For inference, we used a temperature of 0.7 and top-p sampling with $p = 0.95$.

4.2 Results

Main Experiments Figure 3 demonstrates that Lyra's iterative training loop drives substantial performance gains for MiniCPM-V 4.5 9B and InternVL-3.5 1B. Both models exhibit a sharp increase in metrics across all datasets during the early rounds before saturating. The improvements are most pronounced on ChartX where scores roughly double after the initial rounds. This indicates that the generated Vega-Lite specifications provide effective supervision for learning the chart-to-code task. Performance on the out-of-domain benchmarks ChartMimic and Plot2Code also trends upward though with more modest margins compared to ChartX. InternVL-3.5 1B shows the largest relative gains given its weaker initialization and improves almost monotonically. The results suggest

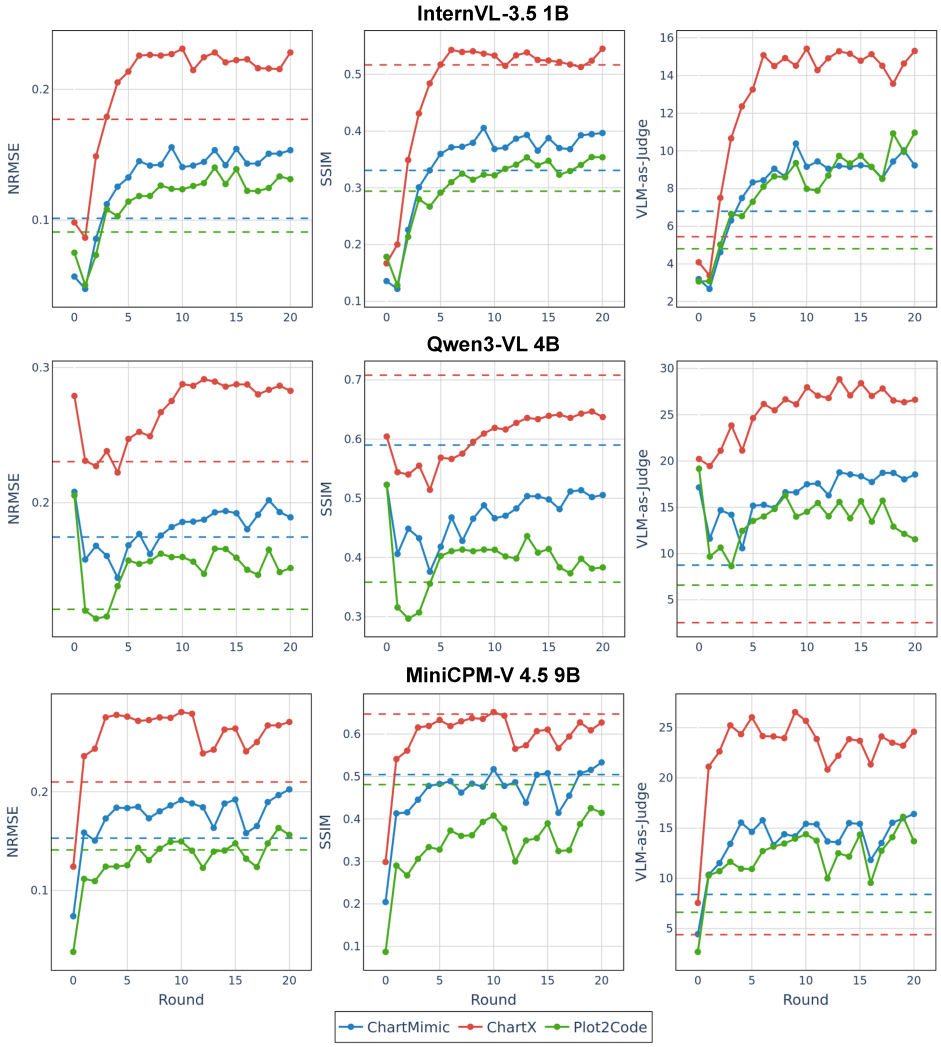

Figure 3: We evaluate performance improvements over 20 rounds on three out-of-domain datasets, Plot2Code, ChartMimic, and ChartX, using image-based metrics: Normalized Root Mean Squared Error (NRMSE), Structural Similarity Index Measure (SSIM), and a vision-language model as judge (VLM-as-Judge). Solid lines represent the performance of `Lyra` with different backbones, while dashed horizontal lines indicate the baseline performance trained on the static ChartCoder-160k dataset.

that the prioritized experience replay effectively prevents overfitting to the synthetic data from any single round while enabling continual learning.

Qwen3-VL 4B begins with strong zero-shot chart-to-code capabilities and consequently exhibits unique training dynamics. Specifically, we observe a transient performance drop on all datasets during the first few rounds before the model recovers. Unlike other models, Qwen3-VL does not demonstrate substantial monotonic growth. This trend indicates that models with strong pre-existing priors are more sensitive to distributional shifts between their original training data and our synthetic data. Although `Lyra` successfully adapts Qwen3-VL to the target Vega-Lite tasks represented by ChartX, maintaining robust generalization to diverse out-of-distribution styles remains a challenge for already-capable models.

As a baseline, we trained the model on the chart-to-Python code task using a subset of the ChartCoder-160k dataset (Zhao et al., 2025). Further details of dataset construction and training are provided in Appendix A.3. We indicate these baseline results using dashed horizontal lines in Figure 3. We observe that `Lyra` outperforms baselines across all datasets, metrics, and models using fewer training examples, with the exception of SSIM on Qwen3-VL and MiniCPM-V 4.5. Moreover, the VLM-as-judge scores surpass the baseline within just one or two rounds. These results demonstrate the sample efficiency of the proposed method and suggest that abstract visualization grammars more

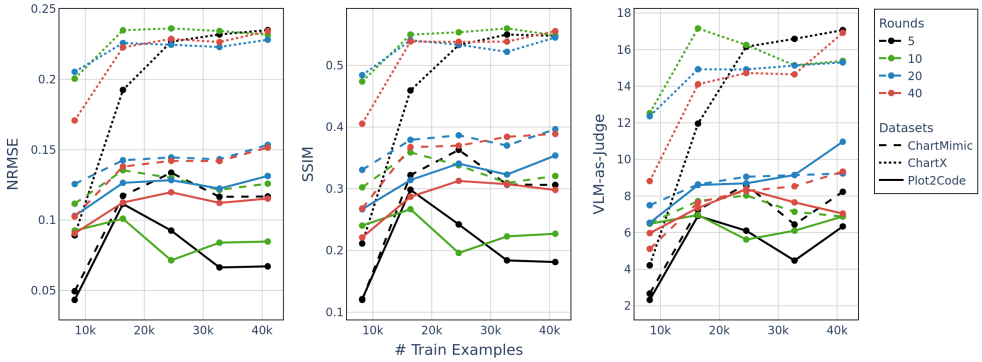

Figure 4: The trade-off between the number of rounds R and the number of training examples per round N_{tr} while fixing the total budget to 40,960 examples.

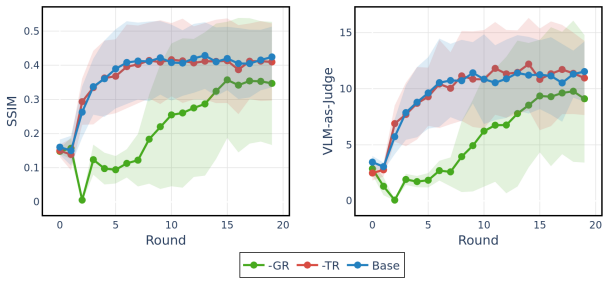

Figure 5: Ablation study on the two sampling processes in Lyra. The *-GR* and *-TR* replace the sampling in the Add operation and training data selection, respectively, with a top-k selection. We report the mean and standard deviation averaged across the three datasets.

Figure 6: Effect of the number of new examples added to the memory per round. N_{tr} is fixed at 2,048.

effectively capture the semantic features of complex charts. Regarding the discrepancy in dataset rankings within the baseline results, pixel-based metrics disproportionately penalize the visual noise in real-world datasets compared to cleaner synthetic data. In contrast, semantic metrics prioritize data fidelity over visual style. This distinction explains why VLM-as-judge scores are highest on ChartMimic, whereas ChartX leads in pixel-based metrics due to its visual uniformity. Additionally, we observe that the Qwen3-VL baseline achieves higher SSIM than Lyra by utilizing its strong reasoning to replicate structural layouts via imperative code, yet incurs higher NRMSE due to fine-grained rendering mismatches inherent to non-standardized plotting libraries.

Trade-off between N_{tr} and R The total number of training examples exposed to the vision-language model is determined by the product of the examples per round N_{tr} and the number of Lyra loops R. To investigate the trade-off between these variables, we fix the total budget to 40,960 examples and evaluate four configurations of $(N_{tr}, R) \in \{(1024, 40), (2048, 20), (4096, 10), (8192, 5)\}$. As shown in Figure 4, increasing R yields better performance than increasing N_{tr} for Plot2Code and ChartMimic. With a higher R, Lyra performs more Add operations, which increases the chance where the model encounters novel data. Additionally, more frequent Add operations allow the LLM to guide generation toward the complex charting practices required for real-world benchmarks like Plot2Code and ChartMimic. However, we observe diminishing returns at $R = 40$, suggesting that excessively small number of training examples can destabilize training. In contrast, results on ChartX are similar across all settings. Since ChartX consists of simpler machine-generated charts, the model performance saturates once it acquires basic charting skills regardless of the specific training schedule.

Ablation Study We demonstrate the impact of our rank-based sampling strategy by comparing Lyra against two deterministic variants as shown in Figure 5. The *-GR* replaces the probabilistic sampling in the Add operation with a greedy top-k selection that exclusively selects the examples with the most extreme scores for task generation. The *-TR* applies a similar top-k strategy to the se-

PEFT	*Plot2Code*			*ChartMimic*			*ChartX*		
	NRMSE	SSIM	Judge	NRMSE	SSIM	Judge	NRMSE	SSIM	Judge
-	**0.13**	**0.35**	**10.97**	**0.15**	**0.40**	**9.23**	0.23	0.55	15.3
Freeze	0.12	0.31	7.51	**0.15**	0.39	8.36	0.23	0.53	14.41
LoRA	**0.13**	0.33	**10.97**	0.12	0.31	8.14	**0.26**	**0.61**	**18.93**

Table 1: Performance comparison between parameter-efficient finetuning and full finetuning across out-of-domain benchmarks.

lection of training data from the replay memory. The results show that *-TR* exhibits a performance trajectory almost identical to the full `Lyra` framework. This suggests that the prioritization of high-error samples is key regardless of whether the selection is probabilistic or deterministic. In contrast, *-GR* suffers a significant performance drop during the initial rounds and subsequently recovers at a much slower rate. This aligns with findings from DQN (Mnih et al., 2015) that learning from strongly correlated samples is inefficient. By restricting task generation to extreme cases, *-GR* amplifies these correlations and concentrates the generated samples within a narrow, biased region.

Replay Memory Expansion Increasing the number of new examples added per round N_a accelerates memory expansion, allowing the model to cover a wider range of charting tasks. However, an excessively large N_a poses two potential risks: it may dilute the sampling probability of high-scored examples, and it increases the number of memory entries with stale scores that do not reflect the model's current state. To examine this trade-off, we compare performance with $N_a \in \{256, 1024, 2048\}$ while fixing the number of training examples N_{tr} to 2,048. As shown in Figure 6, performance generally improves as N_a increases. Notably, this gain is most pronounced on the challenging Plot2Code benchmark, whereas performance on the simpler ChartX benchmark saturates at $N_a = 1024$ with negligible further improvement. This indicates that aggressive memory expansion is particularly beneficial for mastering complex, real-world charting tasks where diverse exploration is critical. Taken together, we suggest that setting $N_a \approx N_{tr}$ is effective, whereas smaller N_a are insufficient for discovering new charting skills.

Parameter-Efficient Finetuning Recent studies suggest that parameter-efficient finetuning methods, such as low-rank adapters (LoRA), can mitigate catastrophic forgetting in continual learning (Liu et al., 2024a). We investigate whether this finding extends to `Lyra` by applying LoRA to the language backbone. In this configuration, we freeze the pre-trained weights and train only the inserted adapters using $\alpha = 8$ and $\mathrm{rank} = 32$. For comparison, we also evaluate a full fine-tuning baseline where the entire language model is updated while other parameters remains frozen. We use the same training and evaluation settings as in the main experiments and report the results in Table 1. As shown in Table 1, LoRA achieves performance comparable to full fine-tuning on Plot2Code and significantly outperforms it on ChartX. This improvement suggests that LoRA effectively mitigates catastrophic forgetting, preserving the model's ability to handle simpler, near in-distribution charts. However, full finetuning retains a clear advantage on the more challenging ChartMimic benchmark, resulting in higher SSIM and VLM-as-judge scores. This indicates that while LoRA is efficient, updating the full parameter set provides the necessary plasticity to master the complex, out-of-distribution visual cues found in real-world replication tasks.

5 Conclusion

In this work, we introduced `Lyra`, a fully automated and lifelong learning framework for the chart-to-code task. By training a vision-language model to predict Vega-Lite specifications rather than imperative code, our approach leverages the structural priors of declarative visualization grammars to constrain the output space. `Lyra` utilizes a large language model to synthesize an adaptive curriculum that generates progressively diverse chart-code pairs. To ensure continual improvement, the framework maintains a replay memory via Prioritized Experience Replay and employs SpecSim, a specification-level similarity metric, to focus training on the most challenging examples. Experiments across three out-of-domain benchmarks demonstrate that `Lyra` progressively enhances chart-to-code performance while effectively mitigating catastrophic forgetting.

Limitations

Although we demonstrate Lyra on the chart-to-code task, the framework is generalizable to the lifelong learning of other complex visual tasks. However, a major challenge in extending this approach is that chart images possess a unique advantage: they can be described nearly perfectly by formal languages. In contrast, natural images generally lack such precise descriptors. Despite advances in diffusion models (Cao et al., 2024), text-to-image mappings do not guarantee that the generated image perfectly reflects the semantic information of the text. Similar information loss affects model-based mappings in other domains, including text-to-speech (Zhang et al., 2023) and text-to-video (Sun et al., 2024). A potential solution is to develop abstract grammars for natural data, analogous to how Vega provides a high-level grammar for charts. Incorporating prior knowledge into such grammars could constrain variability in textual descriptions and mitigate information loss in model-based generation.

References

Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, and 1 others. 2025. Qwen2. 5-vl technical report. *arXiv preprint arXiv:2502.13923*.

Hanqun Cao, Cheng Tan, Zhangyang Gao, Yilun Xu, Guangyong Chen, Pheng-Ann Heng, and Stan Z Li. 2024. A survey on generative diffusion models. *IEEE transactions on knowledge and data engineering*, 36(7):2814–2830.

Zhiyuan Chen and Bing Liu. 2018. *Lifelong machine learning*. Morgan & Claypool Publishers.

Alexis Chevalier, Alexander Wettig, Anirudh Ajith, and Danqi Chen. 2023. Adapting language models to compress contexts. *arXiv preprint arXiv:2305.14788*.

Bosheng Ding, Chengwei Qin, Linlin Liu, Yew Ken Chia, Boyang Li, Shafiq Joty, and Lidong Bing. 2023a. Is GPT-3 a good data annotator? In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11173–11195, Toronto, Canada. Association for Computational Linguistics.

Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. 2023b. Enhancing chat language models by scaling high-quality instructional conversations. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3029–3051.

Ronen Eldan and Yuanzhi Li. 2023. Tinystories: How small can language models be and still speak coherent english? *arXiv preprint arXiv:2305.07759*.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, and 1 others. 2023. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*.

Wei He, Zhiheng Xi, Wanxu Zhao, Xiaoran Fan, Yiwen Ding, Zifei Shan, Tao Gui, Qi Zhang, and Xuanjing Huang. 2025. Distill visual chart reasoning ability from LLMs to MLLMs. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 3224–3250, Suzhou, China. Association for Computational Linguistics.

Wenyi Hong, Wenmeng Yu, Xiaotao Gu, Guo Wang, Guobing Gan, Haomiao Tang, Jiale Cheng, Ji Qi, Junhui Ji, Lihang Pan, and 1 others. 2025. Glm-4.1 v-thinking: Towards versatile multimodal reasoning with scalable reinforcement learning. *arXiv preprint arXiv:2507.01006*.

Or Honovich, Thomas Scialom, Omer Levy, and Timo Schick. 2023. Unnatural instructions: Tuning language models with (almost) no human labor. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14409–14428, Toronto, Canada. Association for Computational Linguistics.

John D Hunter. 2007. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(03):90–95.

Andrew Jaegle, Felix Gimeno, Andy Brock, Oriol Vinyals, Andrew Zisserman, and Joao Carreira. 2021. Perceiver: General perception with iterative attention. In *International conference on machine learning*, pages 4651–4664. PMLR.

Joel Jang, Seonghyeon Ye, Sohee Yang, Joongbo Shin, Janghoon Han, Gyeonghun Kim, Stanley Jungkyu Choi, and Minjoon Seo. 2021. Towards continual knowledge learning of language models. *arXiv preprint arXiv:2110.03215*.

Lingjie Jiang, Shaohan Huang, Xun Wu, Yixia Li, Dongdong Zhang, and Furu Wei. 2025. Viscodex: Unified multimodal code generation via merging vision and coding models. *arXiv preprint arXiv:2508.09945*.

Xisen Jin, Dejiao Zhang, Henghui Zhu, Wei Xiao, Shang-Wen Li, Xiaokai Wei, Andrew Arnold, and Xiang Ren. 2022. Lifelong pretraining: Continually adapting language models to emerging corpora. In *Proceedings of BigScience Episode #5 – Workshop on Challenges & Perspectives in Creating Large Language Models*, pages 1–16, virtual+Dublin. Association for Computational Linguistics.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu,

Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, and 1 others. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526.

Bingxuan Li, Yiwei Wang, Jiuxiang Gu, Kai-Wei Chang, and Nanyun Peng. 2025. METAL: A multi-agent framework for chart generation with test-time scaling. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 30054–30069, Vienna, Austria. Association for Computational Linguistics.

Long-Ji Lin. 1992. *Reinforcement learning for robots using neural networks*. Carnegie Mellon University.

Jiajun Liu, Wenjun Ke, Peng Wang, Jiahao Wang, Jinhua Gao, Ziyu Shang, Guozheng Li, Zijie Xu, Ke Ji, and Yining Li. 2024a. Fast and continual knowledge graph embedding via incremental lora. *arXiv preprint arXiv:2407.05705*.

Ruibo Liu, Jerry Wei, Fangyu Liu, Chenglei Si, Yanzhe Zhang, Jinmeng Rao, Steven Zheng, Daiyi Peng, Diyi Yang, Denny Zhou, and 1 others. 2024b. Best practices and lessons learned on synthetic data for language models. *CoRR*.

Lin Long, Rui Wang, Ruixuan Xiao, Junbo Zhao, Xiao Ding, Gang Chen, and Haobo Wang. 2024. On llms-driven synthetic data generation, curation, and evaluation: A survey. *arXiv preprint arXiv:2406.15126*.

Pan Lu, Hritik Bansal, Tony Xia, Jiacheng Liu, Chunyuan Li, Hannaneh Hajishirzi, Hao Cheng, Kai-Wei Chang, Michel Galley, and Jianfeng Gao. 2024. Mathvista: Evaluating mathematical reasoning of foundation models in visual contexts. In *International Conference on Learning Representations (ICLR)*.

Yun Luo, Zhen Yang, Fandong Meng, Yafu Li, Jie Zhou, and Yue Zhang. 2025. An empirical study of catastrophic forgetting in large language models during continual fine-tuning. *IEEE Transactions on Audio, Speech and Language Processing*.

Arun Mallya and Svetlana Lazebnik. 2018. Packnet: Adding multiple tasks to a single network by iterative pruning. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 7765–7773.

Ahmed Masry, Do Xuan Long, Jia Qing Tan, Shafiq Joty, and Enamul Hoque. 2022. ChartQA: A benchmark for question answering about charts with visual and logical reasoning. In *Findings of the Association for Computational Linguistics: ACL 2022*, pages 2263–2279, Dublin, Ireland. Association for Computational Linguistics.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, and 1 others. 2015. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, and 1 others. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*.

David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy Lillicrap, and Gregory Wayne. 2019. Experience replay for continual learning. *Advances in neural information processing systems*, 32.

Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2016. Vega-lite: A grammar of interactive graphics. *IEEE transactions on visualization and computer graphics*, 23(1):341–350.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2015. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*.

Theodore Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas Griffiths. 2023. Cognitive architectures for language agents. *Transactions on Machine Learning Research*.

Rui Sun, Yumin Zhang, Tejal Shah, Jiahao Sun, Shuoying Zhang, Wenqi Li, Haoran Duan, Bo Wei, and Rajiv Ranjan. 2024. From sora what we can see: A survey of text-to-video generation. *arXiv preprint arXiv:2405.10674*.

Jeffrey S Vitter. 1985. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*.

Weiyun Wang, Zhangwei Gao, Lixin Gu, Hengjun Pu, Long Cui, Xingguang Wei, Zhaoyang Liu, Linglin Jing, Shenglong Ye, Jie Shao, and 1 others. 2025. Internvl3. 5: Advancing open-source multimodal models in versatility, reasoning, and efficiency. *arXiv preprint arXiv:2508.18265*.

Chengyue Wu, Zhixuan Liang, Yixiao Ge, Qiushan Guo, Zeyu Lu, Jiahao Wang, Ying Shan, and Ping Luo. 2025. Plot2Code: A comprehensive benchmark for evaluating multi-modal large language models in code generation from scientific plots. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 3006–3028, Albuquerque, New Mexico. Association for Computational Linguistics.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, and 1 others. 2025. The

rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 68(2):121101.

Renqiu Xia, Hancheng Ye, Xiangchao Yan, Qi Liu, Hongbin Zhou, Zijun Chen, Botian Shi, Junchi Yan, and Bo Zhang. 2025. Chartx & chartvlm: A versatile benchmark and foundation model for complicated chart reasoning. *IEEE Transactions on Image Processing*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Cheng Yang, Chufan Shi, Yaxin Liu, Bo Shui, Junjie Wang, Mohan Jing, Linran Xu, Xinyu Zhu, Siheng Li, Yuxiang Zhang, and 1 others. 2024. Chartmimic: Evaluating lmm's cross-modal reasoning capability via chart-to-code generation. *arXiv preprint arXiv:2406.09961*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.

Tianyu Yu, Zefan Wang, Chongyi Wang, Fuwei Huang, Wenshuo Ma, Zhihui He, Tianchi Cai, Weize Chen, Yuxiang Huang, Yuanqian Zhao, and 1 others. 2025. Minicpm-v 4.5: Cooking efficient mllms via architecture, data, and training recipe. *arXiv preprint arXiv:2509.18154*.

Xiang Yue, Tianyu Zheng, Yuansheng Ni, Yubo Wang, Kai Zhang, Shengbang Tong, Yuxuan Sun, Botao Yu, Ge Zhang, Huan Sun, Yu Su, Wenhu Chen, and Graham Neubig. 2025. MMMU-pro: A more robust multi-discipline multimodal understanding benchmark. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15134–15186, Vienna, Austria. Association for Computational Linguistics.

Chenshuang Zhang, Chaoning Zhang, Sheng Zheng, Mengchun Zhang, Maryam Qamar, Sung-Ho Bae, and In So Kweon. 2023. A survey on audio diffusion models: Text to speech synthesis and enhancement in generative ai. *arXiv preprint arXiv:2303.13336*.

Xuanle Zhao, Xianzhen Luo, Qi Shi, Chi Chen, Shuo Wang, Zhiyuan Liu, and Maosong Sun. 2025. Chart-Coder: Advancing multimodal large language model for chart-to-code generation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7333–7348, Vienna, Austria. Association for Computational Linguistics.

Junhao Zheng, Shengjie Qiu, Chengming Shi, and Qianli Ma. 2025a. Towards lifelong learning of large language models: A survey. *ACM Computing Surveys*, 57(8):1–35.

Junhao Zheng, Chengming Shi, Xidi Cai, Qiuke Li, Duzhen Zhang, Chenxing Li, Dong Yu, and Qianli Ma. 2025b. Lifelong learning of large language model based agents: A roadmap. *arXiv preprint arXiv:2501.07278*.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, and 1 others. 2023. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*.

A Appendix

A.1 Background: Vega-Lite

To render chart images, we use Vega-Lite instead of conventional plotting libraries such as matplotlib (Hunter, 2007). We briefly summarize Vega-Lite here to make this section self-contained. Vega-Lite is a high-level, declarative grammar of interactive graphics that allows users to specify visualizations as concise JSON objects rather than imperative drawing code (Satyanarayan et al., 2016). A basic specification consists of a data source, a mark type, and a set of encoding definitions that map data fields to visual channels such as position, color, size, and shape. Encodings can include common data transformations such as filtering, binning, aggregation, and sorting. Vega-Lite provides smart defaults for scales, axes, and color schemes, so that many details can be omitted in the specification and are inferred automatically by the compiler. Beyond single plots, Vega-Lite defines an algebra for composing multiple views, including layering, horizontal and vertical concatenation, faceting over a partitioning field, and repetition over sets of fields. This enables the succinct construction of dashboards, trellis plots, and scatterplot matrices while automatically sharing or aligning scales and axes when appropriate. In addition, Vega-Lite integrates a grammar of interaction based on selections, which abstract user input as named parameters that can drive conditional encodings, filters, and scale domains. Vega-Lite specifications are compiled to the lower-level Vega language, which handles dataflow, event processing, and rendering, so that users can describe the desired visual and interactive structure at a relatively high level while still obtaining expressive, interactive visualizations.

The use of Vega-Lite offers several advantages over the visualization tools adopted in prior work (Zhao et al., 2025; Li et al., 2025; Jiang et al., 2025) for training and inference with multimodal language models. First, Vega-Lite affords a much lower degree of freedom in the code than general-purpose languages such as Python. The structure and values related to the chart layout must strictly follow the grammar defined by the Vega-Lite specification. In contrast, there are typically many different code snippets in other languages that can render the same chart image, so the chart-to-code task is inherently one-to-many. By reducing the degrees of freedom, Vega-Lite effectively shrinks the target space of the chart-to-code problem, allowing a generative model to learn this mapping from less training data. Second, Vega-Lite encodes prior knowledge about human chart design through default values for properties that are left unspecified in the specification. In many other visualization tools, users must manually specify most design details to obtain a chart that best describes their data. For chart-to-code, this implies that the model must not only map pixels to code fragments, but also infer the design logic that yields an appropriate visualization. With Vega-Lite, part of this logic is delegated to carefully hand-crafted rules, which reduces the burden on the model. Although this sacrifices some flexibility and expressiveness, Vega-Lite remains sufficiently powerful to render almost all general-purpose charts. Taken together, these characteristics reduce problem complexity and enable more sample-efficient training of multimodal models.

A.2 Semantic Score in SpecSim

The semantic score in SpecSim is computed based on four key elements in Vega-Lite specifications: mark, encoding, data, and transformation. Mark similarity compares the mark type and its visual properties, assigning partial credit to closely related mark types and to overlapping mark properties. Encoding similarity measures both the overlap of encoding channels and the agreement of their definitions. Each channel is normalized into one or more field definitions, and we average property-level matches across key Vega-Lite encoding attributes. Data similarity compares inline data values using Jaccard similarity. Transform similarity compares the list of transforms via a combination of multiset Jaccard similarity and longest common subsequence over normalized transform objects, capturing both the multiset of operations and their ordering.

SpecSim also supports composite views such as multi-layer, faceting, concatenation, and repetition. For these cases, it recursively compares corresponding sub-specifications and aggregates their semantic scores with weights proportional to the number of leaf views. When the two specifications have different composite structures, SpecSim extracts representative base specifications and applies a structural penalty based on nesting depth and numbers of leaf views, thus rewarding semantically similar visual encodings even under mismatched layouts.

A.3 Details on ChartCoder-160k Baselines

The ChartCoder baseline utilizes the Chart2Code-160k dataset, which serves as the first large-scale, diverse dataset specifically designed for the chart-to-code generation task. Unlike previous datasets that rely on web-crawled images or simple synthetic pairs, Chart2Code-160k comprises approximately 160,000 high-quality chart-code pairs spanning 27 distinct chart types.

The dataset construction follows a fully automated pipeline powered by LLMs. It begins with the generation of domain-specific keywords and simulated data values to ensure semantic diversity. To guarantee structural variety, the framework utilizes 79 manually crafted code templates covering standard libraries such as Matplotlib and Seaborn. The LLM populates these templates with the generated data to produce executable Python code, which is subsequently rendered to create the corresponding chart images. Rigorous filtering is applied to remove execution errors or low-quality renderings, ensuring a high-fidelity training corpus. To further enhance the reasoning capabilities of the baseline model, the dataset employs a *Snippet-of-Thought* (SoT) strategy. For a subset of 50,000 examples, the direct code generation is decomposed into a step-by-step reasoning format that mimics human programming logic:

1. **Layout Analysis:** Determining the overall chart type and figure structure (e.g., `plt.bar`, `plt.subplots`).

2. **Data Definition:** Extracting and formatting the numerical data and color schemes.

3. **Refinement:** Applying stylistic details such as hatching patterns or legend locations.

4. **Final Code:** Assembling the complete executable script.

.

Following manual inspection, we observed significant redundancy among the code samples in ChartCoder-160k. To eliminate semantically similar duplicates, we first computed the BERTScore between all code pairs using RoBERTa. We then filtered out samples with a similarity $sim(c_1, c_2) > 0.9$, retaining only a single representative from any cluster of overlapping examples. This process reduced the dataset to 42,684 examples, from which we allocated 40,960 for training and 1,724 for validation. Training followed the same hyperparameters as our main experiments, with the exception that we continued training until the validation loss began to increase.

A.4 Prompt Templates

We present the prompt templates for Chart-to-Code Translation (Figure 7), Vega-Lite Specification Generation (Figure 8), and Task Generation (Figure 9).

A.5 Pseudocode

We present pseudocode for the Add and Update operations in Algorithms 2 and 3, respectively. Additionally, Algorithm 4 details the computation of the SpecSim metric.

Chart Image to Vega-Lite JSON Spec

You are a helpful assistant that analyzes chart images and generates Vega-Lite JSON specifications to reproduce them. You will receive:

- Image: A visualization/chart image to analyze and reproduce.

Your job is to generate a single, complete Vega-Lite JSON specification that:
1. Accurately reproduces the chart shown in the input image.
2. Extracts and reconstructs the data visible in the chart as faithfully as possible.
3. Matches the visual design including chart type, colors, axis labels, titles, and styling.
4. Follows all constraints below.

Analysis and Generation Process:
1. Chart Type Identification:
   - Identify the chart type (bar, line, scatter, pie, heatmap, etc.)
   - Note if it's a single chart or composite/layered visualization
2. Data Extraction:
   - Extract data points visible in the chart as accurately as possible
   - Infer approximate values from axis scales and visual positions
   - Reconstruct the dataset structure with meaningful field names and types
3. Visual Properties:
   - Colors: Only specify colors explicitly if they differ from Vega-Lite defaults or use a specific color scheme. Only use explicit color values when matching specific brand colors or non-standard palettes
   - Titles and Labels: Always include chart title and axis titles when present in the image
   - Mark Properties: Only specify mark properties when they visibly differ from defaults
   - Scales: Let Vega-Lite auto-generate scales unless specific domains, ranges, or scale types are needed
   - Legends: Vega-Lite generates legends automatically; only customize when position, format, or visibility differs from defaults
4. Layout and Formatting:
   - Match axis formatting, tick values, and grid lines
   - Reproduce any annotations or text elements

Constraints:
1. Generate Vega-Lite JSON Only.
   - The entire output must be a single, valid JSON object.
2. Leverage Vega-Lite Defaults.
   - Only specify properties when they visibly differ from defaults or are essential to match the image.
3. Use Inline Data.
   - The dataset must be embedded directly within the JSON using the "data": {"values": [...]} property. Do not use the url property or link to any external data files. The entire visualization must be self-contained.

Output Format: Respond with a single code block containing one complete Vega-Lite JSON specification object. For example,

```JSON
{
  "$schema": "https://vega.github.io/schema/vega-lite/v6.json",
  ...
}
```.

Figure 7: Prompt for Chart-to-Code Translation.

Vega-Lite JSON Spec Generation

You are a helpful assistant that creates Vega-Lite JSON specifications for data visualizations. Each task will be defined by the user and includes:

- Task: A one-sentence description of the chart to render.
- Reasoning: The rationale behind why this chart is being generated.

Your job is to generate a single, complete Vega-Lite JSON specification that:
1. Implements the chart exactly as described in the task.
2. Respects the reasoning when making design decisions (e.g., choosing appropriate colors, titles, and encodings).
3. Follows all constraints below.

Code Generation Constraints:
1. Generate Vega-Lite JSON Only.
   - The entire output must be a single, valid JSON object.
2. Specify All Chart Details.
   - Fully define the title, mark, encoding, and any other necessary properties like width and height to create a complete, polished chart.
3. Never Assign Tasks.
   - Do not suggest or create new tasks. Focus solely on executing the provided task.
4. Use Inline Data.
   - The dataset must be generated by you and embedded directly within the JSON using the "data": {"values": [...]} property. Do not use the url property or link to any external data files. The entire visualization must be self-contained.
5. Leverage Vega-Lite Defaults.
   - Prefer concise specifications by omitting properties that already match Vega-Lite's built-in defaults, only overriding defaults when needed to satisfy the Task or Reasoning so that charts remain minimal and readable.

Output Format: Respond with a single code block containing one complete Vega-Lite JSON specification object. For example,

```JSON
{
  "$schema": "https://vega.github.io/schema/vega-lite/v5.json",
  ...
}
```.

Figure 8: Prompt for Vega-Lite Specification Generation.

Task Generation

You are a helpful assistant that tells me the next immediate coding task to do in order to master chart rendering.
My ultimate goal is to discover as many diverse visualization techniques as possible, render charts from a variety of data sets, and become an expert chart programmer.

I will give you the following information:
Completed Task (string): A charting task I have already successfully finished.
Failed or Challenging Task (string): A charting task I struggled with or could not complete.

Follow these criteria:
- Recommend exactly one new task that best advances my current skills.
- Help me steadily grow charting fluency by gradually increasing complexity, data variety, and visualization techniques.
- Always create the chart definition, dataset, layout, encodings, and axes yourself. Never ask me to supply them.

Output Format:
You should only respond in the format as described below:

Reasoning: Based on the information I listed, do reasoning about what the next task should be.
Task: Single sentence explanation of the chart.

Figure 9: Prompt for Task Generation.

Algorithm 2: Add

Input: Replay memory M_t, LLM ϕ, task-generation prompt P_T, spec-generation prompt P_S, threshold $\tau(\cdot)$, number of new entries N_a.
Output: Updated replay memory M_{t+1}.

// Partition memory
$\tau_t \leftarrow \tau(\{s_i\})$;
$M_t^{easy} \leftarrow \{(d_i, s_i) \in M_t \mid s_i > \tau_t\}$, $M_t^{hard} \leftarrow \{(d_i, s_i) \in M_t \mid s_i \leq \tau_t\}$.

// Generate specs, render, and add to memory
$M_{t+1} \leftarrow M_t$
for $j = 1, \ldots, N_a$ **do**
  $T^{hard} \sim P(i|M_t^{hard})$ where $r_i \leftarrow rank(s_i, M_t^{hard})$; // Sample hard task
  $T^{easy} \sim P(i|M_t^{easy})$ where $r_i \leftarrow rank(-s_i, M_t^{easy})$; // Sample easy task
  $T_j \leftarrow \phi(P_T, T^{hard}, T^{easy})$; // Task generation
  $C_j \leftarrow \phi(P_S, T_j)$; // Vega-Lite specification generation
  $I_j \leftarrow \text{Render}(C_j)$; // Image rendering with Vega-Lite engine
  $M_{t+1} \leftarrow M_{t+1} \cup \{((I_j, C_j, T_j), null)\}$
return M_{t+1}.

Algorithm 3: Update

Input: Replay memory M_t, VLM π_t, generation prompt P_G, similarity metric S.
Output: Updated replay memory M_t.

// Select entries whose scores should be refreshed
$\mathcal{U}_t \leftarrow \{\, i \mid s_i = null \text{ or } d_i \in D_t^{train}\}$

for $i \in \mathcal{U}_t$ **do**
  $(I_i, C_i, T_i) \leftarrow d_i$;
  $\widehat{C}_i \leftarrow \pi(P_G, I_i)$; // Chart-image-to-code translation
  $s_i^{new} \leftarrow S(C_i, \widehat{C}_i)$; // Evaluation
  $M_t \leftarrow \big(M_t \setminus \{(d_i, s_i)\}\big) \cup \{(d_i, s_i^{new})\}$
return M_t.

Algorithm 4: SpecSim

Input: Vega-Lite specs c_1 and c_2.
Output: $S(c_1, c_2) \in [0, 1]$.

`// Syntactic similarity on linearized JSON`
$\tilde{c_1} \leftarrow \texttt{Linearize}(c_1)$, $\tilde{c_2} \leftarrow \texttt{Linearize}(c_2)$
$d \leftarrow \texttt{EditDistance}(\tilde{c_1}, \tilde{c_2})$, $L_{\max} \leftarrow \max(|\tilde{c_1}|, |\tilde{c_2}|)$
$S_{\text{syn}} \leftarrow \begin{cases} 1 - d/L_{\max}, & L_{\max} > 0 \\ 1, & \text{otherwise} \end{cases}$

`// Semantic similarity`
$S_{\text{sem}} \leftarrow \texttt{RecursiveSemSim}(c_1, c_2)$

`// Final score`
$S(c_1, c_2) \leftarrow w_{\text{syn}} S_{\text{syn}} + w_{\text{sem}} S_{\text{sem}}$
return $S(c_1, c_2)$.

Function `RecursiveSemSim`(c_1, c_2)**:**

`// Single-view semantics`
$s_{\text{mark}} \leftarrow \texttt{CompareMarks}(c_1, c_2)$
$s_{\text{enc}} \leftarrow \texttt{CompareEncodings}(c_1, c_2)$
$s_{\text{data}} \leftarrow \texttt{CompareData}(c_1, c_2)$
$s_{\text{trans}} \leftarrow \texttt{CompareTransforms}(c_1, c_2)$
$S^{\text{single}} \leftarrow w_{\text{mark}} s_{\text{mark}} + w_{\text{enc}} s_{\text{enc}} + w_{\text{data}} s_{\text{data}} + w_{\text{trans}} s_{\text{trans}}$

`// Composite structure (layer, facet, concat, repeat)`
$t_1 \leftarrow \texttt{CompositeType}(c_1)$, $t_2 \leftarrow \texttt{CompositeType}(c_2)$
if $t_1 = t_2 \neq \varnothing$ **then**
  $S^{\text{comp}} \leftarrow \texttt{CompareComposite}(c_1, c_2, t_1)$
else if $t_1 \neq t_2$ ***or*** *exactly one is composite* **then**
  $\text{base}_1 \leftarrow \texttt{ExtractBaseSpec}(c_1)$, $\text{base}_2 \leftarrow \texttt{ExtractBaseSpec}(c_2)$
  if *$base_1$, $base_2$ exist* **then**
    $p \leftarrow \texttt{StructuralPenalty}(c_1, c_2)$
    $S^{\text{comp}} \leftarrow p \cdot \texttt{RecursiveSemSim}(\text{base}_1, \text{base}_2)$
  else
    $S^{\text{comp}} \leftarrow 0$
else
  $S^{\text{comp}} \leftarrow S^{\text{single}}$; `// No composite structure`

return $\frac{1}{2} S^{\text{single}} + \frac{1}{2} S^{\text{comp}}$.