VARIABLE LENGTH DIFFUSE EVERYTHING: ARBITRARY LENGTH GENERATIVE MODELING ON ARBITRARY STATE SPACES

Anonymous authors
Paper under double-blind review

ABSTRACT

Diffusion and flow-based generative models have achieved remarkable results across diverse applications, but most assume data of fixed-dimension/length. Meanwhile, real-world problems such as layout generation often involve variable-length, multimodal data, such as continuous bounding box coordinates paired with discrete types. Existing methods typically address variable length by padding data to a fixed size or by introducing specialized objectives to model lengths explicitly, adding complexity and limiting generality. We propose Variable Length Diffuse Everything (VL-DE), a simple and general framework for variable-length multimodal generation on arbitrary state spaces. Instead of padding, VL-DE handles variable length through an insertion-based mechanism grounded in a principled theoretical framework, yielding a tractable training objective. In addition, by utilizing the fact that VL-DE is based on a generator with an additive form, we develop a higher-order sampler by appropriately handling the time dependence. Extensive experiments on layout generation demonstrate the effectiveness of our approach.

1 INTRODUCTION

Generative models have led to significant advances across a wide range of scientific and practical applications. Diffusion and flow-based models have been established across a broad range of settings, including Euclidean spaces (Song et al., 2020; Ho et al., 2020; Albergo et al., 2023; Lipman et al., 2022; Liu et al., 2022), curved manifolds (De Bortoli et al., 2022; Huang et al., 2022; Chen and Lipman, 2024; Zhu et al., 2025), discrete spaces (Campbell et al., 2022; Lou et al., 2023; Ou et al., 2024; Sahoo et al., 2024; Shi et al., 2024), and more recently, multimodal data (e.g., Holderrieth et al., 2024; Campbell et al., 2024; Rojas et al., 2025; Benton et al., 2024; Ren et al., 2025b). Variable-length generation has likewise received substantial attention, but almost exclusively in single-modality settings (Campbell et al., 2023; Kim et al., 2025; Havasi et al., 2025). Multimodal generation, on the other hand, has been studied predominantly at fixed length. Only a handful of works sit at the intersection of these two lines (Billera et al., 2025; Nguyen et al., 2025), and even these impose strong structural constraints; such as tying all modalities to a single length.

Variable-length multimodal data arises naturally in many domains. One such example is layout generation, where a layout is a variable-length sequence of discrete box types paired with continuous box coordinates, and both the number of boxes and their arrangement must be generated jointly. A key challenge in such settings is that length is rarely known a priori: much like a language model does not know how many tokens it will produce before generation begins, the appropriate length only becomes apparent as the content unfolds. This makes length an implicit, content-dependent quantity, and suggests that decoupled approaches — in which one first samples a length and then samples data conditioned on it — are unlikely to capture the joint structure faithfully.

A standard approach to variable-length generation relies on auto-regressive models, which generate sequences through a series of insertions. While effective in many settings, auto-regressive generation imposes a left-to-right factorization that is not always natural for the underlying data distribution. This has motivated work to extend diffusion and flow-based models to support variable-length data. *Unimodal* methods such as *Transdimensional Diffusion* (Campbell et al., 2023), *FlexMDM* (Kim et al., 2025), and *Edit Flows* (Havasi et al., 2025) extend diffusion or flow-based models with insertion

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.

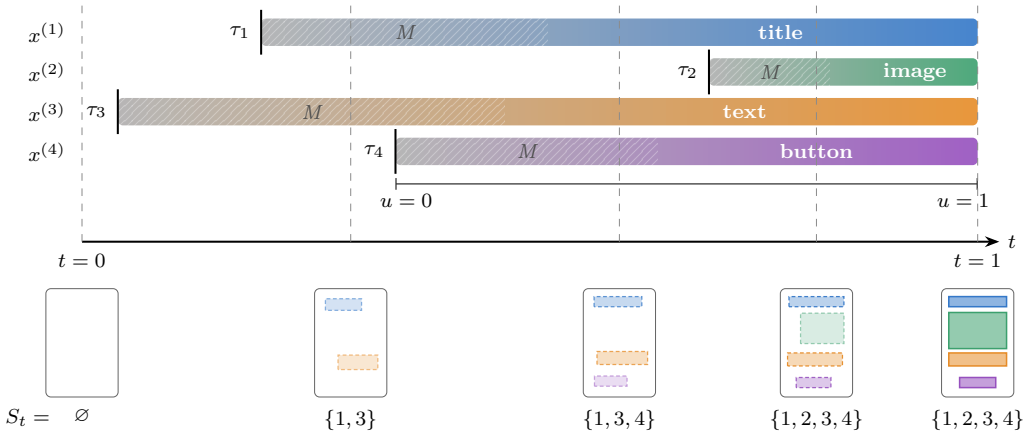


Figure 1: Overview of **VL-DE** on a layout example. Variable-length data is generated by a joint process (X_t, S_t), where S_t is an *auxiliary process* that tracks the active coordinates and X_t carries their values. Each coordinate i receives an insertion time τ_i, an insertion distribution and is denoised on its own clock u, so that all elements are clean simultaneously at $t = 1$.

or editing mechanisms, but do not address the multimodal case. *Interleaving-based multimodal* methods such as *Transfusion* (Zhou et al., 2025) and *OneFlow* (Nguyen et al., 2025) handle the specific pairing of discrete tokens with continuous values by interleaving them in a single sequence; this construction is tied to that particular pairing and does not extend to general orderings in general state spaces. *General-state-space multimodal* frameworks such as *Branching Flows* (Billera et al., 2025) allow arbitrary state spaces, but their duplication dynamics may not suit all data, and they impose structural restrictions like having all modalities share a length.

In this work, we introduce *Variable Length Diffuse Everything* (VL-DE), a framework whose construction rests on two frameworks: *Generator Matching* (Holderrieth et al., 2024) and *Diffuse Everything* (Rojas et al., 2025). Generator Matching offers a general recipe to learn generators of continuous-time Markov processes on arbitrary state spaces, while Diffuse Everything shows how to model multimodal data by combining per-modality generators under decoupled time variables. Neither, however, accommodates variable-length data: Generator Matching applied directly leads to intractable training objectives (as in Campbell et al. (2023)), and Diffuse Everything inherits these issues.

VL-DE addresses both limitations. We extend Generator Matching by augmenting the state space with an auxiliary variable that records the latent operations producing a state. Marginalizing this variable yields a generator on the original variable-length data space, while training on the joint augmented process keeps the objective tractable. As a concrete instantiation, we specify the training process directly as a *variable-length interpolant*: each coordinate of a data point is assigned an insertion time, then as in Diffuse Everything, each coordinate is denoised on its own clock, so that all coordinates are clean simultaneously at the end (see Figure 1). The resulting generator combines insertion and denoising on arbitrary state spaces, and recovers many variable length approaches such as (Kim et al., 2025; Havasi et al., 2025; Nguyen et al., 2025).

Beyond the design of the generative process, efficient *sampling* is essential for practical deployment. To this end, we leverage the fact that our approach is based on the generator of a Markov process that is naturally the sum of multiple terms, and develop higher-order samplers. Notably, traditional operator splitting techniques (McLachlan and Quispel, 2002) do not directly apply because each term is time dependent, and we introduce a novel non-autonomous operator splitting scheme for general Markov processes. These samplers improve sample quality per step at small step counts.

We evaluate VL-DE on variable-length multimodal generation on both unconditional and conditional layout generation. Experimental evidence demonstrates that VL-DE can perform on par with even oracle-length models without sacrificing sample quality and beats other variable length methods.

Our contributions are summarized as follows:

- For generatively modeling multimodal data with variable dimensions, we develop a principled construction based on augmenting state space with auxiliary variables, and show that the resulting dynamics can be marginalized to obtain generators defined on the original state space.

- We instantiate this construction as a variable-length interpolant with insertion and per-coordinate clocks, which is defined on arbitrary state spaces, is trained without marginalizing over latent insertion histories, and can handle multiple lengths.
- We develop an efficient sampling method based on a non-autonomous operator splitting for general Markov processes, improving accuracy per step in the large step size regime.

2 BACKGROUND

Generator Matching Many generative models construct samples by interpolating between a simple reference distribution p_{simple} and the data distribution p_{data}. This includes diffusion, flow-based, and related models across Euclidean, discrete, and multimodal state spaces. A unifying perspective is to view these models as continuous-time Markov processes. Concretely, given a state space $\mathbb{X}$ and data distribution p_{data}, one specifies a *conditional probability path* $p_{t|1}(dx \mid x_1)$ for $t \in [0,1]$ satisfying

$$p_{1|1}(dx \mid x_1) = \delta_{x_1}(x), \qquad p_{0|1}(dx \mid x_1) = p_{\text{simple}}.$$

Averaging over $x_1 \sim p_{\text{data}}$ yields the marginal path $p_t(dx) = \mathbb{E}_{x_1 \sim p_{\text{data}}}\big[p_{t|1}(dx \mid x_1)\big]$, which interpolates between $p_0 = p_{\text{simple}}$ and $p_1 = p_{\text{data}}$. If $(p_t)_{t\in[0,1]}$ is generated by a (possibly time-inhomogeneous) Markov process with generator $\mathcal{L}_t$ (refer to Appendix C.1 for a formal introduction), then its evolution satisfies the weak Fokker-Planck equation

$$\partial_t \langle p_t, f\rangle = \langle p_t, \mathcal{L}_t f\rangle, \tag{1}$$

for suitable test functions f. Thus, learning the generator $\mathcal{L}_t$ is sufficient to specify the generative process. Generator Matching (Holderrieth et al., 2024) provides a general framework for learning such generators. Suppose that for each data point x_1, there exists a conditional generator $\mathcal{L}_{t,x_1}$ generating the path $p_{t|1}(\cdot \mid x_1)$. Then the generator of the marginal path admits the factorization

$$\mathcal{L}_t f(x) = \mathbb{E}_{x_1 \sim p_{1|t}(\cdot|x)}[\mathcal{L}_{t,x_1} f(x)]\,, \tag{2}$$

where $p_{1|t}(\cdot \mid x)$ denotes the posterior over data points given the intermediate state x at time t. Furthermore, for a broad class of Markov processes, the generator admits a linear parameterization

$$\mathcal{L}_t f(x) = \langle Kf(x), F_t(x)\rangle, \tag{3}$$

where K is a fixed linear operator and $F_t(x)$ characterizes the dynamics. Generator Matching aims at learning a neural approximation F_t^θ by minimizing

$$L_{\text{gm}}(\theta) = \mathbb{E}_t\, \mathbb{E}_{x_t \sim p_t}\big[D\big(F_t(x_t), F_t^\theta(x_t)\big)\big], \tag{4}$$

where D is a Bregman divergence. Since this objective is generally intractable, one instead minimizes the *conditional generator matching* objective

$$L_{\text{cgm}}(\theta) = \mathbb{E}_t\, \mathbb{E}_{x_1 \sim p_{\text{data}}}\, \mathbb{E}_{x_t \sim p_{t|1}(\cdot|x_1)}\big[D\big(F_{t,x_1}(x_t), F_t^\theta(x_t)\big)\big], \tag{5}$$

where F_{t,x_1} parameterizes $\mathcal{L}_{t,x_1}$. A key result in Holderrieth et al. (2024) shows that minimizing this objective recovers the generator of the marginal path.

Diffuse Everything Rojas et al. (2025) models multimodal data on arbitrary state spaces. It was originally formulated in the Denoising Markov Models framework (Benton et al., 2024; Ren et al., 2025b), where the generative process is reversing a forward noising process; however, its core construction applies more broadly to any number of Markov processes specified by their generators, including those that do not arise from a forward process. We adopt this more general view here.

Given two modalities $x \in \mathbb{X}$ and $y \in \mathbb{Y}$, each is equipped with its own generator, $\mathcal{L}^x$ and $\mathcal{L}^y$, defined on its respective space. These per-modality generators are then combined into a joint generator using *decoupled time variables* $(t_1, t_2) \in [0,1]^2$, one per modality:

$$\mathcal{L}_{t_1,t_2} = \mathcal{L}^x_{t_1} + \mathcal{L}^y_{t_2}.$$

Under this construction each modality evolves independently under its own clock. The decoupled times allow a single model to support multiple generation modes: setting $t_1 = t_2$ corresponds to joint generation of (x, y), while holding one clock fixed (e.g., $t_1 = 1$) and running the other yields conditional generation (e.g., $y \mid x$). The framework extends to any number of modalities and state spaces. A key restriction, however, is that Diffuse Everything assumes each modality lives in a *fixed-length* state space; lifting this restriction is the central goal of VL-DE. Appendix A discusses this view of Diffuse Everything and the effect of decoupled times on training.

3 METHODOLOGY

In principle, generator matching can model variable length data by defining a Markov process over $\mathbb{X} = \bigcup_{d=1}^{D} \tilde{\mathbb{V}}^d$, where $\tilde{\mathbb{V}}$ is an arbitrary space that may itself be multimodal (e.g., $\tilde{\mathbb{V}} = \mathbb{R}^4 \times [N]$). If this process changes dimensions (e.g., through insertions), it can model variable-length data — as in Trans-dimensional Diffusion (Campbell et al., 2023). However, the difficulty lies in the conditional probability path: many latent operations (e.g., different insertion paths) can produce the same state, so evaluating the loss function requires summing over all such possibilities. This is generally intractable, and Campbell et al. (2023) relies on a practical simplification to make training feasible.

To sidestep this issue, *Edit Flows* (Havasi et al., 2025) and *FlexMDM* (Kim et al., 2025) concurrently introduced an auxiliary process that explicitly tracks the latent operations producing each state, removing the ambiguity and making training tractable. However, both constructions restrict the data and the auxiliary process to discrete spaces, which limits their applicability to multimodal data.

We address this in two steps. Section 3.1 generalizes the auxiliary-process idea to arbitrary state spaces, yielding *Generator Matching with an Auxiliary Process*; this already covers multimodal data with a common length. Section 3.2 then instantiates it as a variable-length interpolant in which every coordinate is denoised on its own clock; since a coordinate may itself be a variable-length object, this also covers modalities with *independently varying lengths*, a regime not handled by prior work.

3.1 GENERATOR MATCHING WITH AN AUXILIARY PROCESS

Notation. We use $\mathcal{L}$ to denote generators acting on augmented state spaces (e.g., including auxiliary variables), and $\mathcal{K}$ to denote generators acting on the marginal data space.

Setup: We start by considering a distribution p_{data} over a space $\mathbb{X}$ that we want to generate new samples from. Unlike standard approaches that define a generator on $\mathbb{X}$, we work on a joint state space $\mathbb{X} \times \mathbb{S}$, where the auxiliary space $\mathbb{S}$ records the latent operations that produced a corresponding state in $\mathbb{X}$ — for example, in Edit Flows the auxiliary process keeps track of the sequence of edit operations used. We require this generator to satisfy the same principle as in Equation 2:

$$\mathcal{L}_t f(x,s) = \mathop{\mathbb{E}}_{x_1,s_1}[\mathcal{L}_{t,x_1,s_1} f(x,s) | (X_t,S_t) = (x,s)] \tag{6}$$

Furthermore, we assume that both $\mathcal{L}_t$ and $\mathcal{L}_{t,x_1,s_1}$ can be parametrized linearly as:

$$\mathcal{L}_t f(x,s) = \langle Kf(x,s), F_t(x,s)\rangle, \quad \mathcal{L}_{t,x_1,s_1} f(x,s) = \langle Kf(x,s), F_{t,x_1,s_1}(x,s)\rangle \tag{7}$$

are the same conditions as in Generator Matching (Holderrieth et al., 2024), reviewed in (2) and (3), except now on the extended state space. These conditions allow defining a generative model over $\mathbb{X} \times \mathbb{S}$. Naively doing so would force us to model S, leading to an unnecessary computational overhead. We propose a better alternative, while keeping the benefits of the auxiliary process.

The marginalized Process: The key insight is that the role of S is to make training tractable by removing the ambiguity of many auxiliary sequences producing the same x. We therefore construct a generator on $\mathbb{X}$ that *marginalizes* S while still letting us exploit it during training, as formalized by the following theorem:

Theorem 3.1. *Let $\mathcal{L}_t f(x,s)$ satisfy equations* (6) *and* (7) *as well as the weak Fokker-Planck equation* (1)*, and let $\varphi_f(x,s) := f(x)$ denote the lift of f to $\mathbb{X} \times \mathbb{S}$. Define the generator $\mathcal{K}_t$ on $\mathbb{X}$ by*

$$\mathcal{K}_t f(x) = \mathop{\mathbb{E}}_{s_t}[\mathcal{L}_t \varphi_f(x,s_t) \mid X_t = x]. \tag{8}$$

Then $\mathcal{K}_t$ generates the marginal probability path $p_t(x)$. Furthermore, the generator admits a factorization given by:

$$\mathcal{K}_t f(x) = \mathop{\mathbb{E}}_{x_1,s_1,s_t}[\mathcal{K}_{t,x_1,s_1,s_t} f(x) | X_t = x] \tag{9}$$

where the conditional generator $\mathcal{K}_{t,x_1,s_1,s_t} f(x) = \mathcal{L}_{t,x_1,s_1} \varphi_f(x,s_t)$. Additionally, assume that the linear operator from (7) *satisfies that $K\varphi_f(x,s) = K_x f(x)$ for some linear operator K_x, then the generator admits a linear parametrization of the form:*

$$\mathcal{K}_t f(x) = \langle K_x f(x), F_t(x)\rangle \tag{10}$$

where $F_t(x)$ is a continuous function.

Algorithm 1 Generator Matching with an Auxiliary Process: recipe.

Step 1. Construct a conditional generator $\mathcal{L}_{t,x_1,s_1}$ generating $p_{t|1}(x_t, s_t|x_1, s_1)$ and satisfying (6);
Step 2. Parametrize $\mathcal{L}_{t,x_1,s_1}$ through $F_{t,x_1,s_1}(x_t, s_t)$, this is the *training target*.
Step 3. Parametrize $\mathcal{K}_t$ (Theorem 3.1) through $F_t^\theta(x)$
Step 4. Train F_t^θ by minimizing the loss $L_{\text{cgm-aux}}(\theta)$ in (11)
Step 5. Sample by simulating the marginalized generator $\mathcal{K}_t$ on $\mathbb{X}$ driven by F_t^θ.

This theorem specifies that given a generator $\mathcal{L}_t$ generating $p_t(x, s)$ and acting on functions of $\mathbb{X} \times \mathbb{S}$, one can define a new generator $\mathcal{K}_t$ acting on functions over $\mathbb{X}$ that generates the marginals $p_t(x)$. Furthermore, it provides us with a principled condition in which the resulting generator can be linearly parametrized. A discussion on which processes satisfy such conditions, as well as a comparison to other related auxiliary-process constructions, is provided in Appendix C.2.

The loss function: Naively one could use the generator $\mathcal{K}_t$ in the conditional generator matching objective in Equation (5). However, doing so would once again result in having to consider all the possible values of S_t. This would complicate the training as in Campbell et al. (2023) and may be computationally intractable. To address this we use the following theorem:

Theorem 3.2. *(Conditional Generator Matching with Auxiliary Process) For a generator satisfying equations* (6)*, we have that for any Bregman divergence, the loss given by*

$$L_{\text{cgm}-\text{aux}}(\theta) = \mathbb{E}_t\, \mathbb{E}_{x_1,s_1\sim p_{data}}\, \mathbb{E}_{x_t,s_t\sim p_{t|1}(\cdot|x_1,s_1)} \big[D\big(F_{t,x_1,s_1}(x_t, s_t), F_t^\theta(x_t)\big) \big], \tag{11}$$

satisfies that $\nabla_\theta L_{\text{gm}} = \nabla_\theta L_{\text{cgm}-\text{aux}}$.

This theorem yields a tractable training objective and is central to our method. Evaluating the loss only requires sampling from the conditional probability paths of the joint process, which is often straightforward. Training on samples from the joint process—rather than the marginalized one—avoids explicitly summing over the many auxiliary configurations that can correspond to the same observed state. We summarize the framework for training with auxiliary processes in Algorithm 1.

3.2 Designing a variable length generative model

We now instantiate step 1 of Algorithm 1: a conditional generator $\mathcal{L}_{t,x_1,s_1}$ for variable-length data over an arbitrary state space $\tilde{\mathbb{V}}$, which may itself be multimodal (e.g. $\tilde{\mathbb{V}} = \mathbb{R}^4 \times [N]$). We take an *interpolant* perspective (Albergo et al., 2023): we first describe what a sample x_t looks like given a data point x_1, and then give the generator that produces it. The description has two steps, illustrated in Figure 1: every coordinate of x_1 receives an *insertion time*, at which it is inserted with a value drawn from an *insertion distribution*, and from then on it is *denoised* toward its value.

The variable-length interpolant: Let $x_1 \in \tilde{\mathbb{V}}^L$. Fix an *insertion distribution* $q(\cdot \mid v)$ on $\tilde{\mathbb{V}}$, and a single-coordinate conditional probability path $p_u(\cdot \mid v)$, $u \in [0, 1]$, from $q(\cdot \mid v)$ to δ_v with conditional generator $\mathcal{A}_{u,v}$, as in standard Generator Matching. For example, one might have a Gaussian prior, $q(\cdot \mid v) = \pi$, with the flow matching path $uv + (1-u)\epsilon$. The insertion distribution then does not depend on x_1 and need not be learned. Alternatively, one may insert *noisy data*: take $q(\cdot \mid v)$ to be the Gaussian path at an intermediate noise level and p_u the same path from that level to the clean value. Now q depends on x_1, and the model must additionally learn the insertion distribution; in the extreme case $q(\cdot \mid v) = \delta_v$ clean values are inserted directly and no denoising is needed.

(i) Insertion times. Fix an increasing schedule γ_t with $\gamma_0 = 0$ and $\gamma_1 = 1$. Each coordinate $i \in [L]$ receives an independent insertion time τ_i with $\mathbb{P}(\tau_i > t) = 1 - \gamma_t$. The auxiliary variable is the ordered set of inserted coordinates $S_t = \{i : \tau_i \le t\}$, so that $|S_t| = \text{len}(x_t)$ and $S_1 = [L]$.

(ii) Insertion and denoising. At time τ_i, coordinate i is inserted with a value drawn from $q(\cdot \mid x_1^{(i)})$. From then on it is denoised toward $x_1^{(i)}$ over the interval $[\tau_i, 1]$, which we view as the coordinate's own *denoising clock* $u(t;\tau) := (t-\tau)/(1-\tau) \in [0, 1]$, running from 0 at insertion to 1 at $t = 1$ irrespective of τ. Hence $x_t^{(i)} \sim p_{u(t;\tau_i)}(\cdot \mid x_1^{(i)})$ independently for each inserted coordinate: a coordinate inserted late is simply denoised faster, and all are clean at the end. Since every coordinate

is denoised on its own time, the construction resembles Diffuse Everything (Rojas et al., 2025), where every modality has its own clock, and from which VL-DE takes its name.

The state is the sequence of inserted coordinates together with their insertion times, $x_t = ((x_t^{(i)}, \tau_i))_{i \in S_t} \in \mathbb{X} := \bigcup_{d \le D} (\tilde{\mathbb{V}} \times [0,1])^d$. Insertion times are observed at sampling time and the denoiser needs them to know the noise level of each coordinate, so they belong to the data state rather than to the auxiliary variable.

The conditional generator: The process above is Markov, and its generator has two terms mirroring the two steps:

$$\mathcal{L}_{t,x_1,s_1} = \mathcal{L}^{\text{insert}}_{t,x_1,s_1} + \mathcal{L}^{\text{denoise}}_{t,x_1}. \tag{12}$$

Insertion is a jump process. Writing $s^{(k)}$ for the index in x_1 of the k-th entry of x and $\text{ins}_k(x, v, t)$ for the state obtained by inserting the entry (v, t) at position k, its jump kernel is

$$R_t\big((x, S) \to (\text{ins}_k(x, v, t),\, S \cup \{i\})\big) = \frac{\dot{\gamma}_t}{1 - \gamma_t}\, \mathbb{1}\{s^{(k-1)} < i < s^{(k)}\}\, q\big(v \mid x_1^{(i)}\big), \tag{13}$$

with the conventions $s^{(0)} = 0$ and $s^{(\text{len}(x)+1)} = L + 1$; it inserts in between existing coordinates as in Kim et al. (2025). The kernel has a *rate* part, which depends on the state only through S, and an *initialization* part q, which is known in closed form when it is a prior and must be learned otherwise. Denoising runs the base generator of every inserted coordinate on its clock, whose speed is $\partial_t u(t;\tau) = 1/(1-\tau)$:

$$\mathcal{L}^{\text{denoise}}_{t,x_1} f(x,s) = \sum_{k=1}^{\text{len}(x)} \frac{1}{1 - \tau^{(k)}}\, \mathcal{A}^{(k)}_{u(t;\tau^{(k)}),\, x_1^{(s^{(k)})}} f(x,s), \tag{14}$$

with $\mathcal{A}^{(k)}$ acting on the k-th value only. Since the rate depends on the state only through S and denoising never changes S, the two steps decouple:

Theorem 3.3. *Let $\mathcal{L}_{t,x_1,s_1}$ be the generator in* (12)*. Then the conditional probability path factorizes as*

$$p_t(x_t, S_t \mid x_1, S_1) = p_t(S_t, \tau \mid S_1) \prod_{i \in S_t} p_{u(t;\tau_i)}\big(x_t^{(i)} \mid x_1^{(i)}\big), \tag{15}$$

where $\tau = (\tau_i)_{i \in S_t}$ are the insertion times contained in x_t. In particular, (x_t, S_t) is sampled by steps (i)–(ii) above, without simulating the process.

The loss function: By Theorem 3.1, at sampling time the model observes only x_t, i.e. values and insertion times, and outputs the denoising parameters of every coordinate, the expected number of missing coordinates in each gap j between consecutive entries, and, when q is not a prior, a distribution $q_t^\theta(\cdot \mid x_t, j)$ for the value of the next coordinate inserted in gap j. Theorem 3.2 then yields the objective.

Lemma 3.4. *For the generator in* (12)*, the training objective is*

$$\mathbb{E}\Big[\underbrace{D\Big(F^{\textit{denoise}}_{t,x_1,s_1}(x_t,s_t),\, F_t^{\theta,\textit{denoise}}(x_t)\Big)}_{\textit{denoise}} + \underbrace{\sum_j \phi\Big(s_t[j] - s_t[j-1] - 1,\; F_t^{\theta,\textit{insert}}(x_t, j)\Big)}_{\textit{insert}} + \underbrace{\sum_j \sum_{s_t[j-1] < i < s_t[j]} \mathbb{E}_{v \sim q(\cdot \mid x_1^{(i)})} \big[-\log q_t^\theta(v \mid x_t, j)\big]}_{\textit{initialization}}\Big],$$

where $t \sim \mathcal{U}[0,1]$, $x_1 \sim p_{\text{data}}$ and $(x_t, s_t) \sim p_{t|1}(\cdot \mid x_1, s_1)$ as in Theorem 3.3. The k-th entry of the denoising target is the base target of $\mathcal{A}$ at clock time $u(t;\tau^{(k)})$ toward $x_1^{(s^{(k)})}$, and $\phi(x,y) = y - x \log y$ is the standard Bregman divergence for jump rates (Kim et al., 2025). When q is a prior, the initialization term is constant in θ and is dropped. See Appendix C.3 for a proof.

All targets are read off from the sampled insertion times, with no marginalization over the latent insertion orders that could have produced x_t. The initialization term is the cross-entropy of the learned insertion distribution against the true one; it plays the role of the unmasking loss in masked diffusion and is only needed when coordinates are inserted as noisy data.

Extensions: The recipe is modular. Other insertion kernels, e.g. auto-regressive or block insertion, may replace (13) (Appendix D.1). Inserting mask tokens recovers FlexMDM (Kim et al., 2025), and inserting clean values with substitutions recovers Edit Flows (Havasi et al., 2025). *Deletions* follow by augmenting x_1 with junk coordinates whose lifetime ends before $t = 1$, and Theorem 3.3 and Lemma 3.4 extend naturally (Appendix D.2). Finally, $\tilde{\mathbb{V}}$ may itself be a variable-length space, so modalities with *independently varying lengths* are handled with one clock per modality as in Diffuse Everything, unlike previous approaches which require equal lengths.

3.3 Higher-Order Sampling via Autonomized Operator Splitting

Higher-order integrators can substantially reduce the number of sampling steps in diffusion models (e.g., Karras et al., 2022; Zhang et al., 2022; Lu et al., 2022; Ren et al., 2025a). In multimodal or variable-length settings, however, operations like insertion alter the dynamics and make standard higher-order samplers hard to apply directly. Existing methods therefore often rely on Euler discretizations with many steps (e.g., Billera et al. (2025) uses 1000).

To address this, we design a sampler based on *autonomized operator splitting*. While motivated by our framework, the method applies more generally to time-dependent Markov generators, so we present the method in its full generality. Consider a time-dependent generator of the form $\mathcal{K}_t = \mathcal{K}_t^A + \mathcal{K}_t^B$.

If there were no time dependence, high-order integration could be accomplished via operator splitting (McLachlan and Quispel, 2002) based on $\exp(\mathcal{K}^{A,*}h)$ and $\exp(\mathcal{K}^{B,*}h)$. However, the non-autonomy, or more precisely, the lack of commutativity in time ($[\mathcal{K}_{t_1}^A, \mathcal{K}_{t_2}^A] \neq 0$, $[\mathcal{K}_{t_1}^B, \mathcal{K}_{t_2}^B] \neq 0$), is what makes accurate numerical integration nontrivial as the exact flow maps induced by $\mathcal{K}_t^{A,*}$ and $\mathcal{K}_t^{B,*}$ are no longer tractable. We address this by introducing auxiliary time variables $t_1(t)$ and $t_2(t)$ in addition to p_t and split the dynamics in a very specific way:

$$\partial_t \begin{pmatrix} p_t(x) \\ t_1 \\ t_2 \end{pmatrix} = \begin{pmatrix} \left(\mathcal{K}_{t_1}^{A,*} + \mathcal{K}_{t_2}^{B,*}\right) p_t(x) \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} \mathcal{K}_{t_1}^{A,*} p_t(x) \\ 0 \\ 1 \end{pmatrix} + \begin{pmatrix} \mathcal{K}_{t_2}^{B,*} p_t(x) \\ 1 \\ 0 \end{pmatrix}. \tag{16}$$

Note the split systems $\partial_t \begin{pmatrix} p_t(x) \\ t_1 \\ t_2 \end{pmatrix} = \begin{pmatrix} \mathcal{K}_{t_1}^{A,*} p_t(x) \\ 0 \\ 1 \end{pmatrix}$ and $\partial_t \begin{pmatrix} p_t(x) \\ t_1 \\ t_2 \end{pmatrix} = \begin{pmatrix} \mathcal{K}_{t_2}^{B,*} p_t(x) \\ 1 \\ 0 \end{pmatrix}$ both became autonomous system because t_1 is a constant in A dynamics and t_2 is a constant in B dynamics, and their analytical solutions are respectively given by the following flow maps:

$$\varphi_A^h(p_t, t_1, t_2) = \begin{pmatrix} \exp(\mathcal{K}_{t_1}^{A,*}h) p_t(x) \\ t_1 \\ t_2 + h \end{pmatrix}, \qquad \varphi_B^h(p_t, t_1, t_2) = \begin{pmatrix} \exp(\mathcal{K}_{t_2}^{B,*}h) p_t(x) \\ t_1 + h \\ t_2 \end{pmatrix}. \tag{17}$$

Second-order sampling. Applying Strang splitting yields the second-order scheme

$$X_{t+h} \sim \varphi_A^{h/2} \circ \varphi_B^h \circ \varphi_A^{h/2}\, \delta_{(X_t, t_1, t_2)}. \tag{18}$$

Naively, this requires three neural network evaluations per step. However, consecutive steps contain adjacent $\varphi_A^{h/2}$ operations, which can be merged using the semigroup property:

$$\varphi_A^{h/2} \circ \varphi_B^h \circ \underbrace{\varphi_A^{h/2} \circ \varphi_A^{h/2}}_{\varphi_A^h} \circ \varphi_B^h \circ \underbrace{\varphi_A^{h/2} \circ \varphi_A^{h/2}}_{\varphi_A^h} \circ \varphi_B^h \cdots \tag{19}$$

This reduces the cost from three to two neural network evaluations per step. Pseudocode for both variants is given in Listings 1 and 2 in Appendix D.3.

Applying this to variable length generation: In our setting we set $\mathcal{K}_t^A$ to be the denoising operator, while $\mathcal{K}_t^B$ corresponds to the insertion operator. This decouples the insertion of new elements from their refinement. In mixed-length multimodal settings we follow this same convention.

4 EXPERIMENTS

We evaluate VL-DE on layout generation, where each element is a discrete category and a continuous bounding box and the number of elements varies. Prior layout models use heuristics for both of these challenges, such as quantization of the coordinates or embedding the categories into the Euclidean space (Zhang et al., 2023; Guerreiro et al., 2024), and receive the number of elements as input or pad to a fixed number of slots. On the other hand, VL-DE generates the length and keeps each modality in its native space. We evaluate it unconditionally on RICO (Deka et al., 2017) and PubLayNet (Zhong et al., 2019), and on CGL (Zhou et al., 2022), where the length must be inferred from the canvas; a demonstration where each box includes variable length text is included in Appendix E.5, other experimental details are in Appendix E.

4.1 LAYOUT GENERATION ON RICO AND PUBLAYNET

We set $\tilde{\mathbb{V}} = [C] \times [0,1]^4$ with a mask prior for the category and Gaussian noise as the box prior (Figure 1), and use an insertion schedule that only inserts elements during $t \in [0, 0.5]$; ablations of both choices can be found in Appendix E.2.

Setup. We follow the training and evaluation protocol of LayoutFlow (Guerreiro et al., 2024) on RICO (Deka et al., 2017), 66K mobile UI layouts with 25 categories, and PubLayNet (Zhong et al., 2019), 360K document layouts with 5 categories. We report FID in the feature space of Zhang et al. (2023), together with the geometric statistics Alignment and Overlap, whose ideal value is that of the validation data. The standard conditional tasks, generation given the categories (Gen-Type) or the categories and sizes (Gen-TypeSize), completion and refinement, are generation modes of Section 3.2 and need no task-specific training. Datasets, metrics, tasks and baselines are detailed in Appendix E.1.

VL-DE matches fixed-length approaches. Table 1 (left) compares VL-DE with the layout generation literature on RICO and PubLayNet. VL-DE neither receives the length nor pads, yet it matches the best of these models on RICO, reaching a new state-of-the-art FID of 2.32 against 2.37 for LayoutFlow. Unselected samples of every method are shown in Figure 4 (Appendix E.1).

Comparison against other variable-length methods. A natural question is how existing variable-length approaches perform on this task. We test this by running FlexMDM and Edit Flows on the tokenized problem of Chai et al. (2023), and OneFlow and Branching Flows on the multimodal setup; details are in Appendix E.1. Table 1 (right) shows that VL-DE outperforms all of them, with OneFlow close behind. We attribute this to the native multimodal dynamics of our method, in which each box evolves jointly with its category rather than separately as in OneFlow. Branching Flows severely underperforms here, as its duplication dynamics may not be well suited to layout generation.

Our samplers improve generation at low steps: Figure 2 compares the samplers of Section 3.3 on the learned-initialization model for 5 to 100 steps. Per step, the direct split reaches an FID of 9.3 at 5 steps and 4.9 at 10, against 26.2 and 7.4 for Euler, and the two match from 50 steps on. The staggered variant does not retain this gain. Per network evaluation, Euler is best at every budget, so the splitting samplers pay off when the number of steps rather than the number of evaluations is the bottleneck. We report all other results with Euler for a fair comparison.

Table 1: **Unconditional layout generation. Left:** layout models grouped by how they handle the number of elements (numbers from Guerreiro et al. (2024); $*$ retrained there). **Right:** methods that generate the number of elements, on RICO, grouped by how they represent boxes. $\rightarrow$: closer to the validation data is better. Protocol and sampling steps in Appendix E.1.

	RICO			PubLayNet		
Model	FID↓	Ali→	Ove→	FID↓	Ali→	Ove→
Fixed number of slots, padded with an empty class						
LayoutDM*	4.43	0.143	0.584	36.85	0.180	0.132
LayoutDiffusion	2.49	0.069	0.502	8.63	0.065	0.003
Number of elements given as input						
DLT*	13.02	0.271	0.571	12.70	0.117	0.036
LayoutFlow	2.37	0.150	0.498	8.87	0.057	0.009
Number of elements generated						
VL-DE (ours)	2.32	0.253	0.547	9.23	0.132	0.014
Validation data	2.10	0.093	0.466	8.10	0.022	0.003

	RICO		
Method	FID↓	Ali→	Ove→
Discrete: boxes quantized to tokens			
Edit Flows	6.43	0.191	0.570
FlexMDM	8.41	0.188	0.606
Multimodal: continuous boxes			
OneFlow	3.31	0.293	0.499
Branching Flows	12.09	0.247	0.643
Multimodal, arbitrary state space			
VL-DE (ours)	2.32	0.253	0.547
Validation data	2.10	0.093	0.466

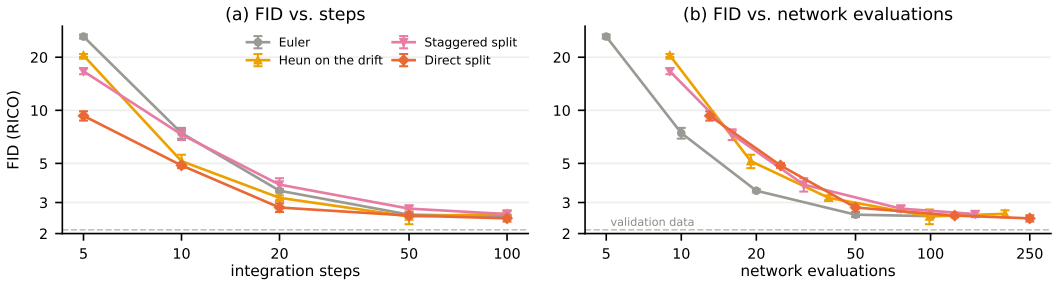


Figure 2: **Operator-splitting samplers on RICO** (Mean over 5 sampling seeds, error bars one standard deviation). (a) Per step, the direct split is far ahead of Euler below 20 steps. (b) Per network evaluation, Euler is the most economical sampler at every budget.

Length mechanism	FID↓	Occ↓	Und$_s$ ↑	Ove↓
Histogram (canvas-blind)	1.63	0.318	0.877	0.006
Padding + empty class	4.02	0.144	0.774	0.013
Autoregressive (RALF w/o retr.)	2.89	0.125	0.917	0.011
Insertion (VL-DE, ours)	1.53	0.148	0.950	0.0097
RALF + retrieval (Horita et al., 2024)	1.32	0.126	0.978	0.0042
Test data	0.80	0.125	0.988	0.0003

Table 2: **Content-aware layout generation on CGL** VL-DE outperforms other base methods in conditional generation. Full metrics, in Table 9; samples in Figure 5.

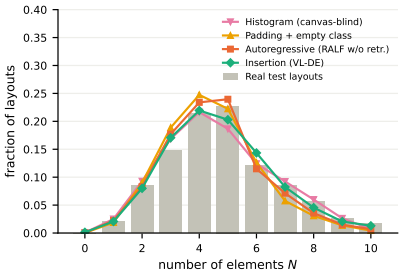


Figure 3: **Generated count distributions on CGL** VL-DE better models the length distribution.

4.2 Content-Aware Layout Generation on CGL

Unconditionally, the length distribution can be estimated from the data itself; the conditional setup does not, for instance, there may be a single example per condition, which makes the length much harder to model. We study this on content-aware layout generation, where the input is a poster canvas and the output its layout. Our goal is not the best content-aware generator but a comparison of how length can be modeled: sampling it from the training histogram, which ignores the canvas; *padding* to a fixed number of slots with an empty class (Zhou et al., 2022; **?**); *autoregressive* decoding with an end token (Horita et al., 2024); and our *insertion* based approach.

Setup. We use CGL (Zhou et al., 2022) with the splits, inpainted canvases, saliency maps and evaluator released by RALF (Horita et al., 2024), and report their metrics on the test split: FID, saliency occlusion (Occ), strict underlay effectiveness (Und$_s$) and overlay (Ove) in Table 2, with unreadability and loose underlay in Appendix E.4. Experimental details are described in Appendix E.4.

VL-DE better models the length and the sample distribution. Table 2 shows the results for the given task, among the existing methods VL-DE obtains the lowest FID and ranks highly in other metrics. Furthermore, Figure 3 shows the histograms for each method where we observe that VL-DE is the best at modeling the length, proving the robustness of our method on the conditional setting.

5 Limitations and Future Work

We highlight a few limitations of the current work. First, VL-DE shows that when compared to oracle-length baselines there might still be some extra error as seen for PubLayNet, suggesting that better optimization or architectures might be needed. Second, while the framework supports arbitrarily many modalities, all of our experiments use two; scaling to substantially more modalities is an interesting direction. Beyond these, VL-DE points to several broader directions: richer auxiliary-process designs (e.g., kernels that mix insertion with reordering or substitutions), and applying the autonomized operator-splitting sampler to non-autonomous generative models beyond the variable-length setting. On layouts, the sampler's gain per step does not carry over to a gain per network evaluation; settings where one operator is cheap to evaluate, such as a separate insertion network, or caching mechanisms, could turn it into a significant gain in cost.

REFERENCES

Michael S Albergo, Nicholas M Boffi, and Eric Vanden-Eijnden. Stochastic interpolants: A unifying framework for flows and diffusions. *arXiv preprint arXiv:2303.08797*, 2023.

Marianne Arriola, Aaron Gokaslan, Justin T Chiu, Zhihan Yang, Zhixuan Qi, Jiaqi Han, Subham Sekhar Sahoo, and Volodymyr Kuleshov. Block diffusion: Interpolating between autoregressive and diffusion language models. *arXiv preprint arXiv:2503.09573*, 2025.

Joe Benton, Yuyang Shi, Valentin De Bortoli, George Deligiannidis, and Arnaud Doucet. From denoising diffusions to denoising markov models. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 86(2):286–301, 2024.

Lukas Billera, Hedwig Nora Nordlinder, Jack Collier Ryder, Anton Oresten, Aron Stålmarck, Theodor Mosetti Björk, and Ben Murrell. Branching flows: Discrete, continuous, and manifold flow matching with splits and deletions. *arXiv preprint arXiv:2511.09465*, 2025.

Andrew Campbell, Joe Benton, Valentin De Bortoli, Thomas Rainforth, George Deligiannidis, and Arnaud Doucet. A continuous time framework for discrete denoising models. *Advances in Neural Information Processing Systems*, 35:28266–28279, 2022.

Andrew Campbell, William Harvey, Christian Weilbach, Valentin De Bortoli, Thomas Rainforth, and Arnaud Doucet. Trans-dimensional generative modeling via jump diffusion models. *Advances in Neural Information Processing Systems*, 36:42217–42257, 2023.

Andrew Campbell, Jason Yim, Regina Barzilay, Tom Rainforth, and Tommi Jaakkola. Generative flows on discrete state-spaces: Enabling multimodal flows with applications to protein co-design. *arXiv preprint arXiv:2402.04997*, 2024.

Shang Chai, Liansheng Zhuang, and Fengying Yan. Layoutdm: Transformer-based diffusion model for layout generation. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023.

Ricky TQ Chen and Yaron Lipman. Flow matching on general geometries. In *ICLR*, 2024.

Valentin De Bortoli, Emile Mathieu, Michael Hutchinson, James Thornton, Yee Whye Teh, and Arnaud Doucet. Riemannian score-based generative modelling. *Advances in neural information processing systems*, 35:2406–2422, 2022.

Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. Rico: A mobile app dataset for building data-driven design applications. In *ACM Symposium on User Interface Software and Technology (UIST)*, 2017.

Julian Jorge Andrade Guerreiro, Naoto Inoue, Kento Masui, Mayu Otani, and Hideki Nakayama. Layoutflow: Flow matching for layout generation. In *European Conference on Computer Vision (ECCV)*, 2024.

Marton Havasi, Brian Karrer, Itai Gat, and Ricky TQ Chen. Edit flows: Flow matching with edit operations. *arXiv preprint arXiv:2506.09018*, 2025.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020.

Peter Holderrieth, Marton Havasi, Jason Yim, Neta Shaul, Itai Gat, Tommi Jaakkola, Brian Karrer, Ricky TQ Chen, and Yaron Lipman. Generator matching: Generative modeling with arbitrary markov processes. *arXiv preprint arXiv:2410.20587*, 2024.

Daichi Horita, Naoto Inoue, Kotaro Kikuchi, Kota Yamaguchi, and Kiyoharu Aizawa. Retrieval-augmented layout transformer for content-aware layout generation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024.

Chin-Wei Huang, Milad Aghajohari, Joey Bose, Prakash Panangaden, and Aaron C Courville. Riemannian diffusion models. *Advances in Neural Information Processing Systems*, 35:2750–2761, 2022.

Zhaoyun Jiang, Jiaqi Guo, Shizhao Sun, Huayu Deng, Zhongkai Wu, Vuksan Mijovic, Zijiang James Yang, Jian-Guang Lou, and Dongmei Zhang. Layoutformer++: Conditional graphic layout generation via constraint serialization and decoding space restriction. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023.

Tero Karras, Miika Aittala, Timo Aila, and Samuli Laine. Elucidating the design space of diffusion-based generative models. *Advances in neural information processing systems*, 35:26565–26577, 2022.

Jaeyeon Kim, Lee Cheuk-Kit, Carles Domingo-Enrich, Yilun Du, Sham Kakade, Timothy Ngotiaoco, Sitan Chen, and Michael Albergo. Any-order flexible length masked diffusion. *arXiv preprint arXiv:2509.01025*, 2025.

Yaron Lipman, Ricky TQ Chen, Heli Ben-Hamu, Maximilian Nickel, and Matt Le. Flow matching for generative modeling. *arXiv preprint arXiv:2210.02747*, 2022.

Xingchao Liu, Chengyue Gong, and Qiang Liu. Flow straight and fast: Learning to generate and transfer data with rectified flow. *arXiv preprint arXiv:2209.03003*, 2022.

Aaron Lou, Chenlin Meng, and Stefano Ermon. Discrete diffusion modeling by estimating the ratios of the data distribution. *arXiv preprint arXiv:2310.16834*, 2023.

Cheng Lu, Yuhao Zhou, Fan Bao, Jianfei Chen, Chongxuan Li, and Jun Zhu. Dpm-solver: A fast ode solver for diffusion probabilistic model sampling in around 10 steps. *Advances in neural information processing systems*, 35:5775–5787, 2022.

Robert I McLachlan and G Reinout W Quispel. Splitting methods. *Acta Numerica*, 11:341–434, 2002.

John Nguyen, Marton Havasi, Tariq Berrada, Luke Zettlemoyer, and Ricky TQ Chen. Oneflow: Concurrent mixed-modal and interleaved generation with edit flows. *arXiv preprint arXiv:2510.03506*, 2025.

Maxime Oquab, Timothée Darcet, Théo Moutakanni, Huy Vo, Marc Szafraniec, Vasil Khalidov, Pierre Fernandez, Daniel Haziza, Francisco Massa, Alaaeldin El-Nouby, et al. DINOv2: Learning robust visual features without supervision. *Transactions on Machine Learning Research*, 2024.

Jingyang Ou, Shen Nie, Kaiwen Xue, Fengqi Zhu, Jiacheng Sun, Zhenguo Li, and Chongxuan Li. Your absorbing discrete diffusion secretly models the conditional distributions of clean data. *arXiv preprint arXiv:2406.03736*, 2024.

Yinuo Ren, Haoxuan Chen, Yuchen Zhu, Wei Guo, Yongxin Chen, Grant M Rotskoff, Molei Tao, and Lexing Ying. Fast solvers for discrete diffusion models: Theory and applications of high-order algorithms. *NeurIPS*, 2025a.

Yinuo Ren, Grant M Rotskoff, and Lexing Ying. A unified approach to analysis and design of denoising markov models. *arXiv preprint arXiv:2504.01938*, 2025b.

Kevin Rojas, Yuchen Zhu, Sichen Zhu, Felix X-F Ye, and Molei Tao. Diffuse everything: Multimodal diffusion models on arbitrary state spaces. *arXiv preprint arXiv:2506.07903*, 2025.

Subham S Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin T Chiu, Alexander Rush, and Volodymyr Kuleshov. Simple and effective masked diffusion language models. *Advances in Neural Information Processing Systems*, 37:130136–130184, 2024.

Jiaxin Shi, Kehang Han, Zhe Wang, Arnaud Doucet, and Michalis Titsias. Simplified and generalized masked diffusion for discrete data. *Advances in neural information processing systems*, 37: 103131–103167, 2024.

Yang Song, Jascha Sohl-Dickstein, Diederik P Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-based generative modeling through stochastic differential equations. *arXiv preprint arXiv:2011.13456*, 2020.

Kota Yamaguchi. Canvasvae: Learning to generate vector graphic documents. In *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 5461–5469. IEEE, 2021.

Junyi Zhang, Jiaqi Guo, Shizhao Sun, Jian-Guang Lou, and Dongmei Zhang. Layoutdiffusion: Improving graphic layout generation by discrete diffusion probabilistic models. In *IEEE/CVF International Conference on Computer Vision (ICCV)*, 2023.

Qinsheng Zhang, Molei Tao, and Yongxin Chen. gddim: Generalized denoising diffusion implicit models. In *The Eleventh International Conference on Learning Representations*, 2022. URL https://arxiv.org/abs/2206.05564.

Xu Zhong, Jianbin Tang, and Antonio Jimeno Yepes. Publaynet: largest dataset ever for document layout analysis. In *International Conference on Document Analysis and Recognition (ICDAR)*, 2019.

Chunting Zhou, LILI YU, Arun Babu, Kushal Tirumala, Michihiro Yasunaga, Leonid Shamis, Jacob Kahn, Xuezhe Ma, Luke Zettlemoyer, and Omer Levy. Transfusion: Predict the next token and diffuse images with one multi-modal model. In *The Thirteenth International Conference on Learning Representations*, 2025.

Min Zhou, Chenchen Xu, Ye Ma, Tiezheng Ge, Yuning Jiang, and Weiwei Xu. Composition-aware graphic layout GAN for visual-textual presentation designs. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence (IJCAI)*, 2022.

Xudong Zhou, Guozheng Li, and Chi Harold Liu. Layoutgd: Content-aware layout generation via graph-enhanced diffusion model. In *Proceedings of the 2026 International Conference on Multimedia Retrieval*, pages 1832–1841, 2026.

Yuchen Zhu, Tianrong Chen, Lingkai Kong, Evangelos Theodorou, and Molei Tao. Trivialized momentum facilitates diffusion generative modeling on lie groups. In *ICLR*, 2025.

AI USE STATEMENT

Generative AI was utilized for verifying proofs, formulating mathematical claims, assisting in the writing of proofs, suggesting edits to mathematical formulations, and designing or providing feedback on research methodology and experiments. All theorems and proofs originated as constructs by the authors, with AI-suggested edits and changes added after the fact. Proofs are verified as correct by the authors and also independently verified by AI checks.

For the experimental code, the authors first wrote a base implementation of the estimators, samplers, and evaluation pipeline. Generative AI was subsequently used to write the scripts that launch and organize experimental runs (e.g., sweeps over hyperparameters, seeds, and problem dimensions), to revise the existing code base, and to produce the plotting code. All AI-generated or AI-modified code was reviewed by the authors. Generative AI was also used as a writing aid, offering edits to improve the clarity and readability of the text.

A DIFFUSE EVERYTHING WITHIN GENERATOR MATCHING

Diffuse Everything was formulated in the Denoising Markov Models framework (Benton et al., 2024; Ren et al., 2025b), which only admits generative processes that reverse a forward noising process. Generator Matching (Holderrieth et al., 2024) imposes no such requirement, so it covers a strictly larger class of generators, and the construction of Diffuse Everything carries over to it unchanged: each modality receives its own generator and its own time variable, and the joint generator is their sum.

The decoupled time variables only change how training times are sampled. Drawing a separate time (t_1, t_2) per modality on paired data trains a single model for both unconditional and conditional generation, since any modality can be held at $t = 1$ at sampling time. Drawing a shared time $t_1 = t_2$ gives up conditional generation but trains the model only on the diagonal it uses for unconditional sampling. Unconditionally, a model trained with a shared time therefore behaves like one trained directly with Generator Matching, whereas decoupled times trade some unconditional quality for the additional generation modes.

B RELATED WORKS

We organize related work into unimodal variable-length methods, interleaving-based multimodal methods, and general-state-space multimodal methods, and discuss how each relates to VL-DE. Table 3 summarizes the comparison.

Branching Flows The work most closely related to ours is Branching Flows Billera et al. (2025). They propose a general framework, similar in spirit to ours, for modeling variable-length data. Unlike our approach, variable length is handled through explicit splits during the generation process. Each resulting branch is then denoised independently until the data distribution is recovered. A key advantage of this strategy is that it avoids the need to explicitly model the distribution of marginals at intermediate times t. However, this comes at the cost of requiring more aggressive velocity fields, which can become problematic when neighboring elements of the data distribution differ significantly in magnitude. In contrast, this approach performs particularly well in settings where neighboring points are close to each other, such as molecule generation. Like our method, Branching Flows are agnostic to the data modality (e.g., discrete, Euclidean, or manifold-valued data) and can be paired with arbitrary flow models.

Flex MDM Another relevant work that inspired our approach is Flex MDM Kim et al. (2025), which addresses variable-length discrete data using mask diffusion models. Their method introduces masked token insertions, which are subsequently unmasked following the standard mask diffusion framework. Their approach corresponds to a special case of our setting when $\mathcal{L}^{\text{denoise}}$ is trivial and random insertions are performed on discrete data. From this perspective, our method recovers the benefits of FlexMDM while extending it to other data modalities as well as more general classes of flow models.

Transdimensional Diffusion Campbell et al. (2023) is one of the earlier works on modeling variable length data using diffusion models. It models variable length euclidean data by using jump diffusions.

Table 3: Comparison of methods for modeling variable-length data. *Inserted object*: what is added when the dimensionality grows. *Modalities*: what state spaces are supported. *Arbitrary denoiser*: whether the per-coordinate dynamics can be any generator on $\tilde{\mathbb{V}}$ (as opposed to a fixed mask-diffusion or noisy-data scheme). *Different per-modality length*: whether each modality can carry its own independent length.

Method	**Inserted Object**	**Modalities**	**Arbitrary denoiser**	**Different per-modality length**
Flex MDM	Mask tokens	Discrete	×	×
Transdimensional Diffusion	Noisy data	Euclidean	×	×
Edit Flows	Noisy data	Discrete	×	×
OneFlow	Noisy data	Euclidean + Discrete	×	×
Transfusion	Clean Tokens / Noisy Images	Euclidean + Discrete	×	×
Branching Flows	Duplicates	General	✓	×
VL-DE (Ours)	Arbitrary	General	✓	✓

A forward process is designed to delete dimensions, while the backwards process is learned to insert dimensions. A key difference from our approach is that this method does not leverage an auxiliary process, resulting in an objective that involves a complex mixture over the data distribution. This significantly complicates training and, in practice, necessitates simplifying assumptions that introduce discrepancies between the theoretical objective and the implemented training procedure. Moreover, insertions are performed directly in the data space rather than via mask tokens, as in our method. While this avoids the need for a separate insertion head, it further increases the complexity of the learning objective. In this sense, both our approach and Branching Flows benefit from a simpler and more structured treatment of insertions.

Edit Flows Havasi et al. (2025) propose another method for modeling variable length in discrete spaces. They combine substitutions, insertions and deletions to model the data distribution. The combination of insertions and deletions allows modeling the length in a flexible manner. Inserted elements follow the marginal distribution at a given time and can subsequently be refined through substitutions, similar in spirit to our denoising of inserted coordinates.

OneFlow Nguyen et al. (2025) extends the Edit Flows framework to Euclidean data by inserting an image token equipped with its own time variable; after insertion, a noise variable associated with the token is progressively denoised. While this extension enables certain multimodal settings, it imposes structural constraints: the modalities must be paired one-to-one within a single sequence, and the data is restricted to a sequential insertion-and-refinement composition.

Transfusion Zhou et al. (2025) interleaves discrete tokens and continuous values into a single sequence and applies a unified transformer that handles each position with the appropriate per-token loss (cross-entropy for discrete, denoising for continuous). Like One Flow, it is tied to the specific pairing of one discrete and one continuous modality and does not extend naturally to general state spaces or to more than two modalities. Its variable-length behaviour is also inherited from the underlying autoregressive backbone rather than by defining a stochastic process as the other approaches considered here.

C Background and Proofs

C.1 Background on Generators

In this section we provide some definitions regarding generators of Markov processes.

Definition C.1. (Generator of a process) Given a time homogenous process X_t we can define the generator as:

$$\mathcal{L}f = \lim_{h \to 0+} \frac{1}{h} \mathbb{E}[f(X_h) - f(X_0)|X_0 = x] \tag{20}$$

for f in the domain of $\mathcal{L}$.

In practice it is common to consider processes that are not time homogenous, for this reason it is useful to consider an augmented process $\tilde{x}_t = (x_t, t)$, this process also has an augmented generator $\tilde{\mathcal{L}}$ that satisfies the relation:

$$\tilde{\mathcal{L}}_t f(\tilde{x}) = \mathcal{L}f(x,t) = \partial_t f(x,t) + \mathcal{L}_t f(x,t)$$

where $\mathcal{L}_t f(x,t) = \lim_{h \to 0+} \frac{1}{h} \mathbb{E}[f(X_{t+h}, t) - f(X_t, t)|X_t = x]$ is the generator at time t. A notable property of generators is that it allows us to define an equation for the time evolution of the density of our process, to do so we need the notion of the adjoint operator.

Definition C.2. (Adjoint Operator) We define the adjoint operator with respect to the inner product as the unique operator satisfying: We define the adjoint operator $\mathcal{L}^*$ with respect to the inner product $\langle \cdot, \cdot \rangle$ on $L^2(E, \mu)$.

$$\langle \mathcal{L}f, g \rangle = \langle f, \mathcal{L}^* g \rangle$$

Given the adjoint operator we can write down the Kolmogorov Forward Equation, otherwise known as Fokker-Planck equation.

Theorem C.3. *(Fokker-Planck Equation) The marginal density $p_t(x)$ of a process with generator $\mathcal{L}$ satisfies:*

$$\partial_t p_t = \mathcal{L}_t^* p_t$$

Alternatively, in its weak form the equation above becomes:

$$\partial_t \int f(x) p_t(x) = \int \mathcal{L}_t f(x) p_t(x) \tag{21}$$

for all test functions f. For a proof refer to Theorem A.23 of Ren et al. (2025b).

For **jump components** of the process, we specify the dynamics through a jump kernel $R_t(z, z' \mid x_1)$, which gives the transition rate from state z to state z'. The associated generator acts as

$$\mathcal{L}_t f(z) = \sum_{z' \neq z} R_t(z, z' \mid x_1)\big(f(z') - f(z)\big),$$

or analogously as an integral when the state space is continuous.

C.2 Auxiliary Generator Matching Proofs

We start by giving the proof of theorem 3.1.

Proof. We first show that $\mathcal{K}_t$ generates $p_t(x)$ by showing it satisfies the Fokker Planck equation:

$$\begin{aligned}
\partial_t \int_{\mathbb{X}} f(x) p_t(x) &= \partial_t \int_{\mathbb{X} \times \mathbb{S}} \varphi_f(x,s) p_t(x,s) \\
&= \int_{\mathbb{X} \times \mathbb{S}} \mathcal{L}_t \varphi_f(x,s) p_t(x,s) \\
&= \int_{\mathbb{X}} p_t(x) \int_{\mathbb{S}} \mathcal{L} \varphi_f(x,s) p_t(s_t | X_t = x) \\
&= \int_{\mathbb{X}} p_t(x) \mathcal{K}_t f(x)
\end{aligned}$$

Which proves the first part of the theorem. The factorization principle is a direct result from the representation above and equation 2:

$$\begin{aligned}\mathcal{K}_t f(x) &= \mathop{\mathbb{E}}_{s_t}[\mathcal{L}_t \varphi_v(x, s_t) | X_t = x] \\ &= \mathop{\mathbb{E}}_{s_t}\Big[\mathop{\mathbb{E}}_{x_1, s_1}[\mathcal{L}_{t, x_1, s_1} \varphi_f(x, s_t) | X_t = x]\Big]\end{aligned}$$

rearranging the integrals gives what we wanted. □

Remark C.4. The theorem above gives us a direct construction of how to compute a marginalized process, as well as sufficient conditions under which such process can be linearized, so that it can be learnable under the framework of generator matching.

Remark C.5. A similar theorem in Billera et al. (2025) was used to construct their process. We must emphasize that the theorems are different. Intuitively, their approach allows for conditioning on a *discrete* process, on the other hand, instead of conditioning on a process, we consider a joint process over an *arbitrary* state space that we marginalize over. Mathematically, Billera et al. (2025) assume that (in our notation) for each $s \in \mathbb{S}$ there exists a generator $\mathcal{L}_t^s$ such that:

$$\partial_t \langle f(x), p_t(x, s) \rangle = \langle \mathcal{L}_t^s f(x), p_t(x, s) \rangle$$

That is, it assumes that there exists a generator for the process under a fixed s. On the other hand, we assume that we have a generator over a joint space $\mathbb{X} \times \mathbb{S}$ and directly construct a process that is learnable through generator matching, which simplifies the design of the auxiliary process.

Remark C.6. We would like to comment on the assumption of $K\varphi_f(x, s) = K_x f(x)$. This is not restrictive as it corresponds to operators such that K affects coordinates locally, while the coupling between variables is captured by the function F. This class admits coupled dynamics—including flows, diffusions, and joint jumps whose coefficients depend on latent variables—but excludes operators that mix coordinates. For example generators of the form $\mathcal{L}_t f(x, s) = \partial_{x,s} f(x, s)$ do not satisfy this assumption. Under this assumption, marginalization of the s variable preserves the operator structure.

We now present the proof that Bregman divergences are a suitable training objective as in Theorem 3.2.

Proof. By the factorization property (6) and the linear representation (7),

$$\begin{aligned}\langle F_t(x), K_x f(x) \rangle &= \mathcal{K}_t f(x) \\ &= \mathbb{E}_{x_1, s_1, s_t}\big[\langle F_{t, x_1, s_1}(x, s_t), K_x f(x) \rangle \,\big|\, X_t = x\big] \\ &= \Big\langle \mathbb{E}_{x_1, s_1, s_t}\big[F_{t, x_1, s_1}(x, s_t) \,\big|\, X_t = x\big], K_x f(x) \Big\rangle.\end{aligned}$$

Since this holds for all test functions f, we obtain

$$F_t(x) = \mathbb{E}_{x_1, s_1, s_t}\big[F_{t, x_1, s_1}(x, s_t) \,\big|\, X_t = x\big].$$

Let D be the Bregman divergence generated by a convex function ϕ,

$$D(a, b) = \phi(a) - \phi(b) - \langle a - b, \nabla\phi(b) \rangle.$$

Then

$$\begin{aligned}L_{\text{gm}} &= \mathbb{E}_t \mathbb{E}_{x_t}\big[D(F_t(x_t), F_t^\theta(x_t))\big] \\ &= \mathbb{E}_t \mathbb{E}_{x_t}\Big[\phi(F_t(x_t)) - \phi(F_t^\theta(x_t)) - \langle F_t(x_t) - F_t^\theta(x_t), \nabla\phi(F_t^\theta(x_t)) \rangle\Big].\end{aligned}$$

Substituting the conditional expectation representation of $F_t(x_t)$ and applying the tower property yields

$$\begin{aligned}L_{\text{gm}} = \mathbb{E}_t \mathbb{E}_{x_1, s_1} \mathbb{E}_{x_t, s_t | x_1, s_1}\Big[&\phi(F_{t, x_1, s_1}(x_t, s_t)) - \phi(F_t^\theta(x_t)) \\ &- \langle F_{t, x_1, s_1}(x_t, s_t) - F_t^\theta(x_t), \nabla\phi(F_t^\theta(x_t)) \rangle\Big] + C,\end{aligned}$$

where

$$C = \mathbb{E}_t \mathbb{E}_{x_t}\big[\phi(F_t(x_t)) - \mathbb{E}_{x_1,s_1,s_t|X_t=x_t}\phi(F_{t,x_1,s_1}(x_t,s_t))\big]$$

is independent of θ.

Therefore,

$$L_{\text{gm}} = \mathbb{E}_t \mathbb{E}_{x_1,s_1} \mathbb{E}_{x_t,s_t|x_1,s_1}\big[D(F_{t,x_1,s_1}(x_t,s_t), F_t^\theta(x_t))\big] + C,$$

with C independent of θ and the loss functions have the same gradients. □

C.3 Proofs for our Process

We now move on to prove what are the conditional probability paths for our process. We start by stating a notion of independence from a variable for generators.

Definition C.7 (Variable independence of a generator)**.** Let (X_t, S_t) be a time-inhomogeneous Markov process with generator $\mathcal{L}_t$ acting on bounded test functions $f : \mathcal{X} \times \mathcal{S} \to \mathbb{R}$ via

$$(\mathcal{L}_t f)(x,s) := \lim_{h\downarrow 0} \frac{1}{h}\Big(\mathbb{E}[f(X_{t+h}, S_{t+h}) \mid X_t = x, S_t = s] - f(x,s)\Big).$$

We say that $\mathcal{L}_t$ is *independent of* s if for every f there exists an operator $\mathcal{A}_t$ acting on functions $\varphi : \mathcal{X} \to \mathbb{R}$ such that

$$(\mathcal{L}_t f)(x,s) = (\mathcal{A}_t \varphi_s)(x), \qquad \varphi_s(\cdot) := f(\cdot, s),$$

for all (x, s). Equivalently, $(\mathcal{L}_t f)(x, s)$ depends on (x, s) only through x. Similarly, $\mathcal{L}_t$ is *independent of* x if $(\mathcal{L}_t f)(x, s)$ depends on (x, s) only through s.

Lemma C.8 (Factoring out an s-only term)**.** *Assume $\mathcal{L}_t$ is independent of s as in definition C.7, with associated operator $\mathcal{A}_t$. Then for any product test function $f(x, s) = h(s)g(x)$ we have*

$$(\mathcal{L}_t f)(x,s) = h(s)\,(\mathcal{A}_t g)(x).$$

Proof. The proof follows from the definition of the generator:

$$\begin{aligned}
\mathcal{L}_t f(x,s) &= \mathcal{A}_t f(\cdot, s)(x) \\
&= \lim_{h\to 0} \frac{1}{h}\,\mathbb{E}\left[f(x_{t+h}, s) - f(x_t, s)|X_t = x\right] \\
&= \lim_{h\to 0} \frac{1}{h}\,\mathbb{E}\left[h(s)g(x_{t+h}) - h(s)g(x_t)|X_t = x\right] \\
&= h(s) \lim_{h\to 0} \frac{1}{h}\,\mathbb{E}\left[g(x_{t+h}) - g(x_t)|X_t = x\right] \\
&= h(s)\mathcal{A}_t g(x)
\end{aligned}$$

as we wanted. □

We now prove the main Theorem 3.3:

Proof. Fix (x_1, s_1) and write $p_t(x, s) := p_t(x, s \mid x_1, s_1)$. The forward (Kolmogorov) equation is

$$\partial_t p_t \;=\; \mathcal{L}^*_{t,x_1}\, p_t. \tag{22}$$

By our construction,

$$\mathcal{L}_{t,x_1} \;=\; \mathcal{L}^X_{t,x_1} \;+\; \mathcal{L}^S_t,$$

where $\mathcal{L}^X_{t,x_1}$ acts only on the x-coordinate (and depends on s only through the restricted endpoint $x_1^{(s)}$), and $\mathcal{L}^S_t$ acts only on the s-coordinate (and is independent of x). Consequently the adjoint also splits:

$$\mathcal{L}^*_{t,x_1} \;=\; (\mathcal{L}^X_{t,x_1})^* \;+\; (\mathcal{L}^S_t)^*.$$

We claim that the conditional law factorizes as

$$p_t(x, s \mid x_1, s_1) \;=\; p_t(s \mid s_1)\, p_t(x \mid x_1^{(s)}). \tag{23}$$

Differentiating (23) in time gives

$$\partial_t p_t(x, s \mid x_1, s_1) = \big(\partial_t p_t(s \mid s_1)\big)\, p_t(x \mid x_1^{(s)}) + p_t(s \mid s_1)\, \partial_t p_t(x \mid x_1^{(s)}).$$

On the other hand, applying the split adjoint to (23) yields

$$\begin{aligned}\mathcal{L}^*_{t,x_1}\, p_t(x, s \mid x_1, s_1) &= (\mathcal{L}^X_{t,x_1})^*\Big[p_t(s \mid s_1)\, p_t(x \mid x_1^{(s)})\Big] + (\mathcal{L}^S_t)^*\Big[p_t(s \mid s_1)\, p_t(x \mid x_1^{(s)})\Big] \\ &= p_t(s \mid s_1)\, (\mathcal{L}^X_{t,x_1})^* p_t(x \mid x_1^{(s)}) + p_t(x \mid x_1^{(s)})\, (\mathcal{L}^S_t)^* p_t(s \mid s_1),\end{aligned}$$

since $(\mathcal{L}^X_{t,x_1})^*$ acts only on x and $(\mathcal{L}^S_t)^*$ acts only on s and using Lemma C.8.

Therefore (22) holds for the product form (23) provided that

$$\partial_t p_t(s \mid s_1) = (\mathcal{L}^S_t)^* p_t(s \mid s_1), \qquad \partial_t p_t(x \mid x_1^{(s)}) = (\mathcal{L}^X_{t,x_1})^* p_t(x \mid x_1^{(s)}),$$

with the natural initial conditions at $t = 1$. By uniqueness of solutions to the forward equation on the joint space, (23) must coincide with the true conditional law. Hence

$$p_t(X_t, S_t \mid x_1, s_1) = p_t(S_t \mid s_1)\, p_t(X_t \mid x_1^{(S_t)}),$$

as claimed. □

Finally, we prove Lemma 3.4:

Proof. Recall that the conditional generator is $\mathcal{L}_{t,x_1,s_1} = \mathcal{L}^{\text{denoise}}_{t,x_1} + \mathcal{L}^{\text{insert}}_{t,x_1,s_1}$, and that each component admits a linear parametrization in the sense of (7). Applying Theorem 3.2 yields the objective

$$\mathbb{E}_{t,(x_1,s_1),(x_t,s_t)}\Big[D\big(F_{t,x_1,s_1}(x_t, s_t), F^\theta_t(x_t)\big)\Big],$$

and since the generator decomposes additively, the feature representation, and hence the loss, separates into a denoising and an insertion contribution. The denoising term is $D\big(F^{\text{denoise}}_{t,x_1,s_1}(x_t, s_t), F^{\theta,\text{denoise}}_t(x_t)\big)$.

The insertion component is a jump process. Write $\lambda_t := \dot\gamma_t/(1-\gamma_t)$ and $m_t[j] := s_t[j] - s_t[j-1] - 1$ for the number of missing coordinates in gap j. On $\mathbb{X}$, the state reached by inserting a missing coordinate i of gap j with value v is $\text{ins}_j(x_t, v, t)$, which does not depend on i. Summing the kernel (13) over these coordinates, the conditional jump measure out of gap j is

$$\mu_{t,j}(dv) = \lambda_t \sum_{s_t[j-1]<i<s_t[j]} q\big(dv \mid x_1^{(i)}\big),$$

with total mass $\lambda_t\, m_t[j]$. We parametrize it as $\mu^\theta_{t,j}(dv) = \lambda_t\, F^{\theta,\text{insert}}_t(x_t, j)\, q^\theta_t(dv \mid x_t, j)$ with q^θ_t normalized, and take as Bregman divergence the generalized KL between jump measures, $D(\mu, \mu^\theta) = \int \big(d\mu^\theta - \log \frac{d\mu^\theta}{d\nu}\, d\mu\big)$ up to terms independent of θ, for a reference measure ν. Substituting,

$$D\big(\mu_{t,j}, \mu^\theta_{t,j}\big) = \lambda_t\, \phi\Big(m_t[j], F^{\theta,\text{insert}}_t(x_t, j)\Big) + \lambda_t \sum_{s_t[j-1]<i<s_t[j]} \mathbb{E}_{v\sim q(\cdot\mid x_1^{(i)})}\big[-\log q^\theta_t(v \mid x_t, j)\big] + C,$$

where C does not depend on θ. The first term is the insertion loss and the second the initialization loss; when q is a prior, $q^\theta_t = \pi$ is fixed and the second term is constant. The factor $\lambda_t > 0$ depends only on t and the minimizer is attained pointwise in (t, x_t), so dropping it reweights the objective over time without changing its minimizer. Summing over gaps and collecting the terms inside the common expectation yields the stated objective. □

D Design choices for our method

D.1 Insertion Methods

As discussed in Section 3.2, the insertion kernel only adds positions to S_t (never removes them) and can be specified independently of the denoising dynamics. All variants below share the same global

schedule γ_t, which controls the *rate* at which insertions happen; they differ in *which* positions receive the next insertion. The list below is illustrative, not exhaustive: any rate matrix that monotonically grows S to the target size yields a valid variable-length generator, and the choice can be tailored to the structure of the data.

Random insertion. The default kernel from Equation (12) of the main body, repeated here for reference:

$$R_t\big((x,S),(\text{ins}(x,i),S\cup\{i\})\mid x_1\big)=\frac{\dot\gamma_t}{1-\gamma_t}\,\mathbb{1}\{s^{(i-1)}<i<s^{(i)}\},$$

with boundary conventions $s^{(0)}=0$ and $s^{(|S|+1)}=\text{len}(x_1)+1$. The main appeal is order-agnosticism: the model can fill in the structure in whatever order best matches the data.

Auto-regressive insertion. Alternatively, one can restrict insertions to the right end of the current sequence, recovering an auto-regressive growth pattern:

$$R_t\big((x,S),(\text{ins}(x,\text{len}(x)+1),S\cup\{\text{len}(x)+1\})\mid x_1\big)=\frac{\dot\gamma_t}{1-\gamma_t}\,(\text{len}(x_1)-\text{len}(x)).$$

The $(\text{len}(x_1)-\text{len}(x))$ factor scales the rate by the remaining length, so the expected number of insertions tracks the global schedule γ_t regardless of how far the sequence has grown. This kernel is natural when the underlying data has an inherent left-to-right ordering.

Block insertion. Both kernels above can be extended to insert several mask tokens at once, which is useful when the data evolves in fixed-size groups — for example, 3D coordinates that come in blocks of three, or block-wise schemes like Block Diffusion Arriola et al. (2025).

D.2 Deletions via Junk Augmentation

The recipe of Section 3.2 extends to processes that can also *delete* coordinates, through the standard procedure that we outline here. The idea is to augment each data point with junk coordinates that are deleted before $t=1$, so that the model learns which coordinates do not belong. We spell out the construction and show that Theorem 3.3 and Lemma 3.4 carry over unchanged.

Augmentation. Given $x_1\in\tilde{\mathbb{V}}^L$, sample $(\tilde x_1,\ell)\sim q_{\text{aug}}(\cdot\mid x_1)$, where $\tilde x_1\in\tilde{\mathbb{V}}^{L+J}$ is obtained by inserting $J\geq 0$ junk values into x_1 and $\ell\in\{\text{real},\text{junk}\}^{L+J}$ labels each slot; write $\mathcal{J}:=\{i:\ell_i=\text{junk}\}$. Removing the slots in $\mathcal{J}$ from $\tilde x_1$ recovers x_1. The augmentation distribution is a design choice; a natural default places a Poisson number of junk slots uniformly among the gaps of x_1 and draws their values from the marginal distribution of coordinate values, so that junk cannot be recognized from its value alone. The case $J=0$ recovers the insertion-only process of the main text.

Lifetimes. Fix increasing schedules γ_t and β_t with $\gamma_0=0$ and $\gamma_1=\beta_1=1$. Each slot $i\in[L+J]$ receives an insertion time τ_i with $\mathbb{P}(\tau_i>t)=1-\gamma_t$, as in the main text, and each junk slot additionally receives a deletion time $\delta_i>\tau_i$ with $\mathbb{P}(\delta_i>t\mid\tau_i)=(1-\beta_t)/(1-\beta_{\tau_i})$; real slots have $\delta_i=\infty$. Slot i is alive during $[\tau_i,\delta_i)$. The auxiliary variable records the alive slots and the slots already deleted,

$$s_t=(S_t,\Delta_t),\qquad S_t=\{i:\tau_i\leq t<\delta_i\}\text{ (ordered)},\qquad \Delta_t=\{i\in\mathcal{J}:\delta_i\leq t\},$$

and slots in neither set have not been inserted yet. Since $\gamma_1=\beta_1=1$, at $t=1$ every real slot is alive and every junk slot has been deleted, so the process ends at x_1. While alive, every slot follows the base interpolant on its local clock exactly as in the main text, $x_t^{(i)}\sim p_{u(t;\tau_i)}(\cdot\mid\tilde x_1^{(i)})$; junk slots are denoised toward their junk value, so in isolation they are statistically indistinguishable from real ones and can only be identified through their context.

Generator. The conditional generator gains a deletion term,

$$\mathcal{L}_{t,\tilde x_1,\ell}=\mathcal{L}^{\text{insert}}_{t,\ell}+\mathcal{L}^{\text{denoise}}_{t,\tilde x_1}+\mathcal{L}^{\text{delete}}_{t,\ell}.\tag{24}$$

Write $z=(x,S,\Delta)$, let $z^+_{k,i}(v):=(\text{ins}_k(x,v,t),\,S\cup\{i\},\,\Delta)$ denote the state after inserting slot i at position k with value v and clock t, and let $z^-_k:=(\text{delete}_k(x),\,S\setminus\{s^{(k)}\},\,\Delta\cup\{s^{(k)}\})$ denote

the state after removing the k-th coordinate together with its clock. Insertion uses the kernel (13) restricted to slots that have not been deleted, and deletion fires only on junk:

$$R_t^{\text{ins}}\big(z \to z_{k,i}^{+}(v)\big) = \frac{\dot{\gamma}_t}{1-\gamma_t}\,\mathbb{1}\{s^{(k-1)} < i < s^{(k)},\ i \notin \Delta\}\,q\big(v \mid \tilde{x}_1^{(i)}\big), \tag{25}$$

$$R_t^{\text{delete}}\big(z \to z_k^{-}\big) = \frac{\dot{\beta}_t}{1-\beta_t}\,\mathbb{1}\{s^{(k)} \in \mathcal{J}\}. \tag{26}$$

A junk slot inserted at time τ survives until time t with probability $(1-\beta_t)/(1-\beta_\tau)$, which vanishes at $t=1$, and a deleted slot is never re-inserted. The denoising generator is (14) with targets $\tilde{x}_1$. Real and junk slots are inserted through the same kernel, so the insertion mechanism cannot distinguish them.

D.3 Sampler Pseudocode

We provide pseudocode for the two higher-order sampler variants from Section 3.3.

```
def sample(x, N):  # x: initial condition, N: nu
    h = 1 / N
    t1 = 0
    t2 = 0
    x, t2 = flow_A(x, t1, h / 2), h/2
    for n in range(N):
        x = flow_B(x, t2, h)
        t1 += h
        dt = h if n != N-1 else h/2
        x = flow_A(x, t1, dt)
        t2 += dt
    return x
```

Listing 1: Staggered operator-splitting sampler (NFEs = 2).

```
def sample(x, N):  # x: initial condition, N: num steps
    h = 1 / N
    t1 = 0
    t2 = 0
    for n in range(N):
        x = flow_A(x, t1, h / 2)
        t2 += h / 2
        x = flow_B(x, t2, h)
        t1 += h
        x = flow_A(x, t1, h / 2)
        t2 += h / 2
    return x
```

Listing 2: Direct operator-splitting sampler (NFEs = 3).

E Experimental Details

In this section we present experimental details for the experiments. Every model was trained and evaluated on a single NVIDIA A100 (80 GB) GPU. Hyperparameters for RICO and PubLayNet are in Table 4.

Table 4: Hyperparameters for layout generation on RICO and PubLayNet. The backbone matches the size of the Transformer used by Guerreiro et al. (2024).

Hyperparameter	**RICO**	**PubLayNet**
Architecture	Transformer over the element set, no positional encoding	
Layers	4	4
Width	512	512
Heads	8	8
Parameters	15.7M	15.7M
Insertion	from the prior, own clock	masked, learned initialization
Learned initialization head	–	16-component Gaussian mixture
Reveal horizon $t_{\max}$	0.5	0.5
Loss weights (category / insertion / ℓ_1)	0.25 / 0.1 / 0.2	0.25 / 0.1 / 0.2
Training	2,000 epochs (124K steps)	1,000 epochs (609K steps)
Batch size	512	512
Optimizer	AdamW	AdamW
Learning rate	5×10^{-4}, reduced $10\times$ on a validation-FID plateau	
EMA	none	none
Integration steps (eval)	100 (Euler)	100 (Euler)

E.1 Layout Generation: Additional Details

Data representation. Each element is a pair of a category in $[C]$ and a bounding box in $[0, 1]^4$, parametrized by center and size and normalized by the canvas dimensions. Layouts with more than 20 elements are discarded, following Zhang et al. (2023); Jiang et al. (2023). There are $C = 25$ categories on RICO and 5 on PubLayNet. The layout is treated as a set: the backbone has no positional encoding and elements that do not exist yet are hidden by a padding mask, so the insertion kernel adds elements without an ordering. The RICO model inserts an element with a rate per existing element and draws its box from the Gaussian prior on its own clock; the PubLayNet model inserts masked elements at one rate per layout and reveals each box at the current noise level from the learned mixture head.

Evaluation. We follow the same evaluation protocol as in the official respository of Zhang et al. (2023). Including the task types, architecture and data splits.

Table 5: **Layout generation on RICO and PubLayNet, all tasks and methods.** Conditional tasks are reported for VL-DE on RICO only. Baseline numbers are from Guerreiro et al. (2024);

		RICO				PubLayNet			
Task	Model	FID↓	Ali→	Ove→	mIoU↑	FID↓	Ali→	Ove→	mIoU↑
Un-Gen	LayoutTransformer	24.32	0.037	0.542	0.587	30.05	0.067	0.005	0.359
	LayoutFormer++	20.20	0.051	0.546	0.634	47.08	0.228	0.001	0.401
	LayoutDM*‡	4.43	0.143	0.584	0.582	36.85	0.180	0.132	0.382
	DLT*†	13.02	0.271	0.571	0.566	12.70	0.117	0.036	0.431
	LayoutDiffusion‡	2.49	0.069	0.502	0.620	8.63	0.065	0.003	0.417
	LayoutFlow†	2.37	0.150	0.498	0.570	8.87	0.057	0.009	0.424
	VL-DE (ours)	2.32	0.253	0.547	0.578	9.23	0.132	0.014	0.411
Gen-Type	NDN-none	13.76	0.560	0.550	0.350	35.67	0.350	0.170	0.310
	LayoutFormer++	2.48	0.124	0.537	0.377	10.15	0.025	0.009	0.333
	LayoutDM*	2.39	0.222	0.598	0.341	39.12	0.267	0.139	0.348
	DLT*	6.64	0.303	0.616	0.326	7.09	0.097	0.040	0.349
	LayoutDiffusion	1.56	0.124	0.491	0.345	3.73	0.029	0.005	0.343
	LayoutFlow	1.48	0.176	0.517	0.322	3.66	0.037	0.011	0.350
	VL-DE (ours)	1.36	0.177	0.523	0.339	–	–	–	–
Gen-TypeSize	LayoutDM*	1.76	0.175	0.606	0.424	29.91	0.246	0.160	0.436
	DLT*	6.27	0.332	0.609	0.424	5.35	0.130	0.053	0.426
	LayoutFlow	1.03	0.283	0.523	0.470	1.26	0.041	0.031	0.454
	VL-DE (ours)	1.11	0.193	0.525	0.453	–	–	–	–
	Validation data	2.10	0.093	0.466	0.658	8.10	0.022	0.003	0.434

Table 6: **Completion and refinement on RICO and PubLayNet.** VL-DE is reported on RICO only. Baseline numbers are from Guerreiro et al. (2024); models marked with $*$ were retrained there on the common split. In completion, LayoutFlow and LayoutDM are given the number of missing elements, whereas VL-DE generates it.

		RICO				PubLayNet			
Task	Model	FID↓	Ali→	Ove→	mIoU↑	FID↓	Ali→	Ove→	mIoU↑
Completion	LayoutDM*	6.80	0.054	0.630	0.678	25.02	0.169	0.107	0.678
	LayoutFlow	1.51	0.150	0.474	0.741	1.10	0.054	0.127	0.746
	VL-DE (ours)	2.92	0.154	0.539	0.532	–	–	–	–
Refinement	RUITE	7.93	0.177	0.492	0.658	7.89	0.073	0.038	0.637
	LayoutFormer++	3.67	0.141	0.503	0.656	2.94	0.042	0.013	0.642
	LayoutDM*	2.91	0.143	0.575	0.437	48.61	0.286	0.173	0.372
	LayoutDiffusion	0.55	0.102	0.469	0.719	2.05	0.035	0.008	0.660
	LayoutFlow	0.77	0.152	0.455	0.700	0.18	0.020	0.005	0.723
	VL-DE (ours)	0.84	0.136	0.454	0.688	–	–	–	–
	Validation data	2.10	0.093	0.466	0.658	8.10	0.022	0.003	0.434

E.2 Insertion Mechanism and Schedule

All variants share the setup of Table 4, draw the category of a new element from a classifier, and insert following $\kappa(t) = \min(t/t_{\max}, 1)$. They differ in the inserted box:

- **Fully noised:** $x \sim \mathcal{N}(0, I)$, denoised on its own clock $u = (t - \tau)/(1 - \tau)$.
- **Noisy data at the global time:** $x = t\,\hat{x}_1 + (1-t)\,\varepsilon$, with $\hat{x}_1$ sampled from a learned head, denoised on the shared clock t.
- **Masked, then revealed:** a mask token, revealed later as in the previous bullet.

The first two predict one insertion rate per existing element, the third a single rate per layout. Since the layout is a set, the new element could be any missing one, so the target of the head is multimodal; we therefore compare a point estimate, a single Gaussian and a 16-component Gaussian mixture.

Table 7 shows that per-element rates matter more than the form of the box, that $t_{\max} = 0.5$ is best while $t_{\max} = 1$ fails, and that the mixture head is best and converges in about half the epochs. The main results use the fully noised mechanism with $t_{\max} = 0.5$.

Table 7: **Insertion mechanism and schedule on RICO**, unconditional generation. Mechanism rows: $t_{\max} = 0.5$, 5 sampling seeds with the sampler of the main results. Schedule rows: masked-then-revealed mechanism, single runs. Head rows: the same model at fixed length, where all elements are present from the start and revealed from the head, single runs; "epochs" is the number of training epochs to reach a validation FID of 2.5.

	Setting	FID↓	Ali→	Ove→	epochs
Mechanism	Fully noised	2.34	0.217	0.557	–
	Noisy data at the global time	2.30	0.221	0.543	–
	Masked, then revealed	2.59	0.215	0.513	–
Schedule	$t_{\max} = 0.25$	2.55	0.240	0.497	–
	$t_{\max} = 0.5$	2.43	0.233	0.509	–
	$t_{\max} = 1$	11.94	0.269	0.645	–
Head	Gaussian mixture (16 components)	2.30	0.205	0.512	749
	Single Gaussian	2.39	0.223	0.532	1524
	Point estimate	2.39	0.235	0.509	1299

E.3 Other Ablations on RICO

Clocks. Training the category and box clocks independently, as Diffuse Everything does for any-to-any generation, costs almost two FID points; we train them jointly and obtain the conditional tasks from a task mixture instead (2.58 unconditional). Table 8 gives the numbers.

Table 8: **Other ablations on RICO**, unconditional generation. 5 sampling seeds.

	Setting	FID↓	Ali→	Ove→
Clocks	Independent category and box clocks	4.17	0.284	0.674
	Shared clock, task mixture	2.58	0.169	0.497

Variable-length baselines. All baselines share our backbone size, data and evaluation, and are reported with the best-by-validation checkpoint over 5 sampling seeds. Edit Flows (Havasi et al., 2025) and FlexMDM (Kim et al., 2025) run on the 32-bin tokenization of Chai et al. (2023), with 1,000 steps; for FlexMDM we use the exact random unmasking order, and insert the 5 tokens required per-box at once. OneFlow (Nguyen et al., 2025) runs with 50 steps. Branching Flows (Billera et al., 2025) runs with 1,000 steps and predicts the velocity toward its anchor, which was needed for the boxes to sharpen.

Qualitative samples. Figure 4 shows samples of every method. LayoutFlow receives the number of elements; the other methods generate it.

E.4 Content-Aware Layout Generation on CGL

We use RALF's release of CGL (Horita et al., 2024) (splits of $48{,}544$ / $6{,}002$ / $6{,}002$ posters with at most 10 elements, inpainted canvases and saliency maps) and run its evaluation script unmodified on our samples. The model is the RICO backbone of Table 4 with the learned-initialization insertion, which transfers better than inserting from the prior on this dataset. Each canvas is encoded once by a frozen DINOv2-small (Oquab et al., 2024) into an 8×8 grid of patch tokens with the mean saliency of each cell appended; elements read it through a gated cross-attention layer, and a pairwise geometric attention bias as in LayoutGD (Zhou et al., 2026) is added between elements. To avoid memorizing the training canvases, we hide half of the canvas tokens and, for 10% of the samples, the whole canvas, and flip canvas and layout horizontally with probability 0.5. Training runs for $1{,}000$ epochs with the optimizer of Table 4, and the checkpoint is selected by validation FID. At the last sampling step, each underlay is enlarged to strictly contain the element that is already mostly inside it; this containment projection raises strict underlay effectiveness from 0.78 to 0.95 without changing FID. Table 9 reports all of RALF's metrics and Figure 5 unselected samples.

Table 9: **Content-aware layout generation on CGL** (test split, RALF's evaluator), grouped by how the number of elements is modeled. RALF rows are computed from their released outputs; † marks numbers taken from the papers, which report no FID. "Proj." applies a single containment projection at the last sampling step (Appendix E.4). Table 2 in the main text reports the four-column summary.

Model	FID↓	Occ↓	Rea↓	Und_l ↑	Und_s ↑	Ove↓
Histogram of N (canvas ignored)						
VL-DE, canvas-blind	1.63	0.318	0.041	0.960	0.877	0.006
Fixed number of slots, padded with an empty class						
VL-DE backbone, padding	4.02	0.144	0.022	–	0.774	0.013
Autoregressive with an end token						
RALF without retrieval	2.89	0.125	0.019	0.967	0.917	0.011
Insertion						
VL-DE (ours)	1.53	0.148	0.023	0.933	0.784	0.0097
VL-DE + proj.	1.54	0.149	0.022	0.959	0.950	0.0097
Reference: retrieval-augmented and published diffusion models						
RALF (Horita et al., 2024) (16 retrieved layouts)	1.32	0.126	0.018	0.992	0.978	0.0042
LayoutDiT† (**?**)	–	0.124	0.016	0.995	0.988	0.0016
LayoutGD† (Zhou et al., 2026)	–	0.120	0.014	0.999	0.994	0.0004
Test data	0.80	0.125	0.017	0.995	0.988	0.0003

E.5 Demonstration: Independent Lengths on Crello

The per-element state may itself be variable-length. This section demonstrates that the construction accommodates two independently varying lengths in one sampling run; it is not a benchmark result. On Crello (Yamaguchi, 2021), a design is a variable number of elements, and every text element carries a string of its own length. We keep the layout model of Section 4.2, which generates the number of elements by insertion, and give the strings their own generator, a fine-tuned autoregressive language model; which we model as a Markov process on the text coordinate, so it fits the construction unchanged, but it is not trained with our insertion objective. In partial completion, half of a design's elements are given with their strings, and the model must decide how many elements to add and write the strings of the new text elements. In the joint sampler the strings are written once during insertion, at $t = 0.35$, and condition the remaining steps, so the element count and the string lengths are generated in one run. Compared with generating the layout first and the strings afterwards, this reduces the error in the number of added text elements from 1.26 to 1.06 (paired bootstrap interval $[0.12, 0.26]$ over $1{,}691$ test designs) and the layout FID from 2.48 to 2.34 (Table 10, samples in Figure 6). We also compare to a single autoregressive sequence over all elements and strings . We find that this approach models the lengths better, but at a large cost in layout quality. Extending this experiment to jointly generate the layout and the text that goes into them at a larger scale are left for future work, and hope this serves as a proof of concept.

Table 10: **Partial completion on Crello** (1,691 test designs; real designs add 4.4 elements). Errors are mean absolute differences in the number of added elements and added text elements. †: significantly better than the pipeline (paired bootstrap over designs).

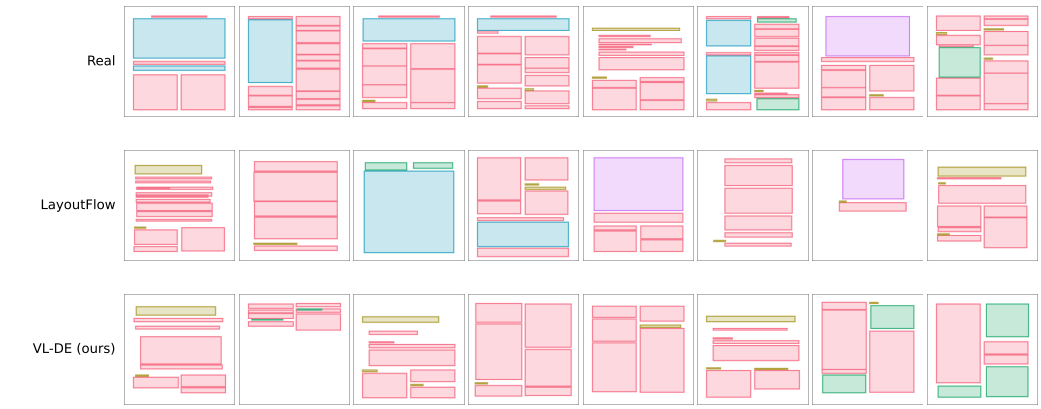
Model	added-elem. err.↓	added-text err.↓	FID↓
Layout, then strings (pipeline)	2.09	1.26	2.48
Joint (VL-DE)	1.96†	1.06†	**2.34**
Flat autoregressive sequence	**1.81**	**0.89**	8.54

Protocol. We use the Crello dataset (Yamaguchi, 2021) (18,414 / 1,754 / 1,869 templates with at most 20 elements, five element types). The first element of a template is the full-canvas background, which conditions the layout; the other elements are generated. The layout model is the CGL model of Appendix E.4 with the font size of text elements as a fifth box coordinate and a per-element feature of the string (a CLIP text embedding and its character, word and line counts), trained with conditioning dropout and with batches in which a random subset of elements is given. The strings are written by Qwen2.5-7B-Instruct with a LoRA adapter, prompted with the template's title and keywords and the layout serialized as text, which writes all strings of a design in one pass and ends each at its own length. Layout FID uses a layout network trained on Crello with RALF's recipe.

For every test design with at least two text elements (1,691 designs), a random half of the elements is given with type, box, font size and string, with at least one text element given and one held out. The pipeline completes the layout without seeing the given strings and then writes the new strings. The joint sampler conditions on the given strings from the start, calls the language model once at $t = 0.35$ on the current clean-layout estimate, and a last time for elements inserted afterwards. Rewriting the strings repeatedly during sampling is unstable, since each rewrite lengthens them and the layout grows to fit. The flat baseline is the same language model and adapter size trained to emit the whole design as one sequence, without the background image. Confidence intervals are paired bootstraps over designs with 5,000 resamples.



(a) RICO (25 categories).



(b) PubLayNet (5 categories: text, title, list, table, figure).

Figure 4: **Unconditional layout generation.** Rows are methods, columns eight layouts of a seeded batch; colors are categories. LayoutFlow is given the number of elements, the other methods generate it.



Figure 5: **Content-aware layouts on CGL, unselected test canvases.** Boxes are drawn over the inpainted canvas; underlays are drawn beneath the other elements.

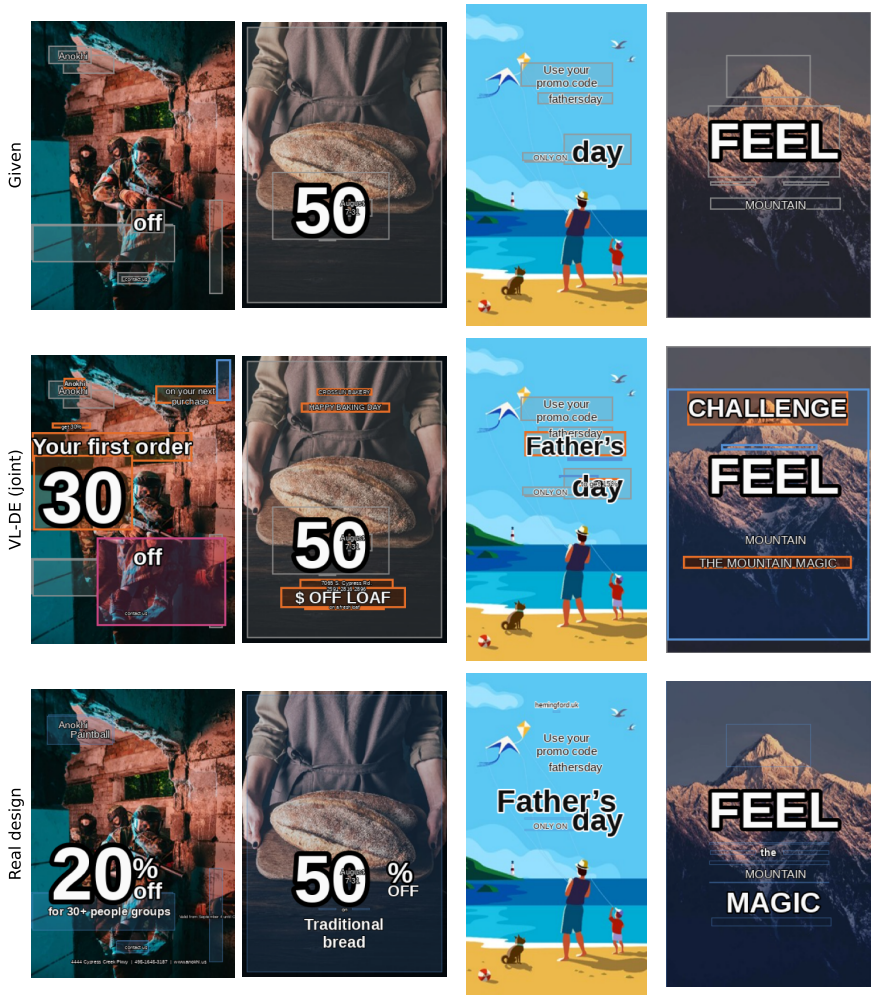


Figure 6: **Partial completion on Crello**, four test designs. **Top:** the given half of the design, in gray with its strings. **Middle:** the joint sampler of VL-DE adds elements (colored boxes) and writes their strings in the same run. **Bottom:** the real design. Designs were selected from test designs without a near-duplicate in the training set.