Branching Flows: Discrete, Continuous, and Manifold Flow Matching with Splits and Deletions

Lukas Billera[1*†], **Hedwig Nora Nordlinder**[1†], **Jack Collier Ryder**[1], **Anton Oresten**[1], **Aron Stålmarck**[1], **Theodor Mosetti Björk**[1], **Ben Murrell**[1,2‡]
[1] Department of Microbiology, Tumor and Cell Biology, Karolinska Institutet
[2] Science for Life Laboratory, Stockholm, Sweden

Abstract

Diffusion and flow matching approaches to generative modeling have shown promise in domains where the number of elements in a state is fixed in advance (e.g. images), but require *ad hoc* solutions when, for example, the length of a response from a large language model, the number of atoms in a molecule, or the number of amino acids in a protein chain is not known *a priori*. Here we propose Branching Flows, a generative modeling framework that, like diffusion and flow matching approaches, transports a simple distribution to the data distribution. But in Branching Flows, the elements in the state evolve over a forest of binary trees, branching and dying stochastically with rates that are learned by the model. This allows the model to control, during generation, the number of elements in the sequence. We show that Branching Flows can compose with any flow matching base process on discrete sets, continuous Euclidean spaces, Riemannian manifolds, and 'multimodal' product spaces that mix these components, and we demonstrate distribution matching on small molecules and antibody sequences, and that this scales to complicated domains such as protein structures. Code available at `https://github.com/MurrellGroup/BranchingFlows.jl`.

1 Introduction

For many applications that involve generating a sequence of elements, it isn't possible to know the required number of elements in advance. For sequences of discrete tokens this is well addressed by autoregressive (AR) models which control length by sampling stop tokens. Continuous AR models are possible [Billera et al., 2024] but underexplored, and current examples are limited in the range of problems they can address. Diffusion and flow matching methods avoid left-to-right AR structure and excel for structured grids and fixed-length sequences, but sacrifice the ability to naturally handle length variation.

One challenge not naturally addressed by fixed-length diffusion/flow models nor AR models (especially in the continuous state context) is what we call the 'unknown-length infix sampling' problem, where a segment of unknown length must be inserted into a sequence, conditioned on *a priori* known flanking regions. As a concrete example from protein design: conditioned on framework regions and some binding pose, if you wish to design the complementarity-determining regions of an antibody you must first specify how long they must be. These length distributions, especially when conditioned on some structural context, are usually not known. Further, there is a 'soft' version of this that arises for fixed-length diffusion/flow models, where flanking regions happen to resolve early during an inference trajectory, and the model cannot control the number of elements spanning them.

*Present affiliation: Chai Discovery
†Equal contribution
‡Correspondence: benjamin.murrell@ki.se

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

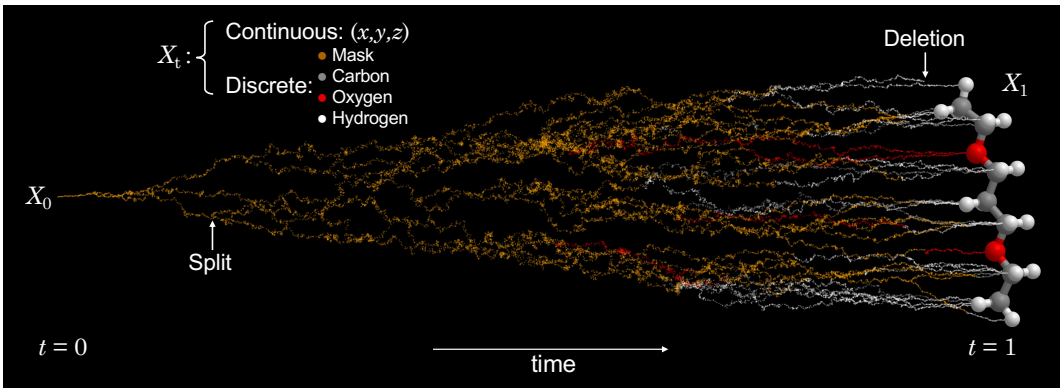


Figure 1: **Generative modeling with splits and deletions.** An illustration of an inference trajectory from a Branching Flows model where a variable-length state is generated by a stochastic process with splits and deletions, starting from a single element.

Here we present Branching Flows, a framework for generative modeling over variable-length sequences where the elements can be continuous, manifold-valued, discrete, or combinations of these. In Branching Flows, we construct a conditional process that transports X_0 samples from a simple distribution to X_1 samples from the data distribution via element splitting and deletion, such that the X_0 and X_1 elements do not have to be the same length and we show how to learn a marginal generator of samples from the data distribution without knowing their length *a priori*.

Contributions

- We develop a branching conditional path construction that couples samples of different length, and that composes with a variety of flow matching processes on continuous, manifold-valued, and discrete states, as well as multimodal combinations of these.
- We generalize both Generator Matching [Holderrieth et al., 2024] and Edit Flows [Havasi et al., 2025] to allow the flexibility of Generator Matching while marginalizing over time-dependent discrete latent processes. Using this, we prove that training under the conditional Branching Flows loss learns a marginal generator that samples from the data distribution.
- We show that Branching Flows outperforms prior discrete-only and continuous-only variable length frameworks, and that it scales to complicated multimodal domains like protein structure and sequence sampling.
- We show, on small molecule generation, that while default choices are well-behaved, the large theoretically well-founded Branching Flows design space can be used to encode favourable inductive biases that can improve task-specific performance.

2 Related Work

Edit Flows. Havasi et al. [2025] describe Edit Flows, a method for varying-dimensionality generation of discrete sequences. Edit Flows works by parameterizing the rates of *edit operations* (insert, delete and substitute). During training, an *auxiliary alignment process* is introduced. This entails adjoining to the alphabet a null token ε such that the sequence length of X_0 (a sample from a simple distribution) is the same as the sequence length of X_1 (a sample from the data distribution). Then, transitions in this augmented space are interpreted as either insertions (swapping ε to a token in the alphabet), deletions (swapping a token in the alphabet for ε) or a substitutions (swapping two tokens in the alphabet). Edit Flow's proves, for the discrete case, that you can marginalize over this time-dependent latent process, learning to sample from the data distribution.

Transdimensional jumps. Transdimensional jump diffusion [Campbell et al., 2023] models continuous-valued variable-cardinality states. Between jumps, a base SDE evolves the continuous coordinates, while at jump times, the dimension changes by one. In the forward data-noising direction (diffusion models adopt the opposite convention to flow matching), components are deleted, one at a time, at a specified rate, chosen such that the forward process will terminate at a single surviving component. The time reversal of the forward process is again a jump diffusion process, but

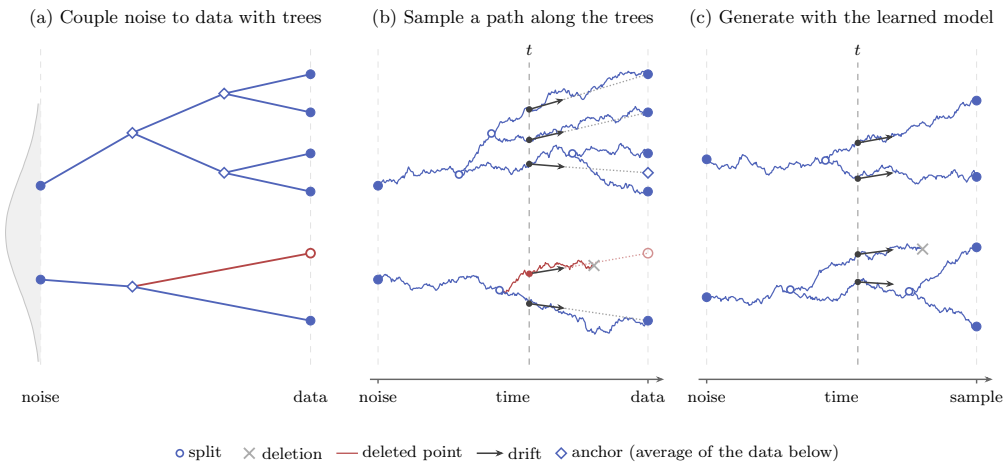


Figure 2: In Branching Flows, during training a forest of trees (a) couples a sample from noise X_0 to data X_1 (augmented with additional to-be-deleted elements), and the conditional process (b) evolves the state over this coupling via stochastic split and deletion events, with the element-wise drift on internal branches governed by 'anchors' (which can be chosen arbitrarily - e.g. as the average value of descendant elements). A model learns, given the current state, the expected drift and split & deletion rates, which are then used during sampling (c) with no explicit tree structure.

where elements are inserted into the state, and a neural network parametrizes the insertion state and post-insertion distribution of the inserted element. One challenge is that, for theoretically-justified distribution learning, the model is required to output an arbitrarily flexible continuous distribution over where the next element might be inserted, which is challenging in practice.

DrugFlow. In Schneuing et al. [2025] a trans-dimensional model for small molecule generation is presented. It works by introducing a virtual atom type whose coordinates are set to the center of mass of the ligand and which have no bonds. During training the training data is augmented by adding $n_{\text{virt}} \sim U(\{0, 1, \ldots, N_{\text{max}}\})$ virtual nodes, to allow the model to learn to set some atom categories to the virtual category. The model is thus deletion-only and requires specifying a priori N_{max} which leads to a fixed upper bound on generated molecule size. This contrasts to Branching Flows, which can generate arbitrary-length objects and allows both insertions and deletions. Furthermore, elements in the virtual category are retained in the flow and evolve under it (although their bond types are set to 'None'), which forces model computation to always be done on the maximum number of elements for the specific molecule being generated.

3 Methods

3.1 Generator Matching

Generator Matching (GM) [Holderrieth et al., 2024, Lipman et al., 2024] provides a common framework for generative processes based on flows, diffusions, and jumps. For a suitable test function f, the infinitesimal generator of a time-inhomogeneous Markov process is

$$\mathcal{L}_t f(x) := \left.\frac{d}{dh}\right|_{h=0^+} \mathbb{E}[f(X_{t+h}) \mid X_t = x].$$

It describes the instantaneous evolution of the process. For example, a Euclidean jump diffusion with drift u_t, diffusion covariance a_t, and jump kernel Q_t has generator

$$\mathcal{L}_t f(x) = \nabla f(x)^\top u_t(x) + \tfrac{1}{2}\operatorname{tr}(a_t(x)\nabla^2 f(x)) + \int (f(y) - f(x)) Q_t(dy; x).$$

A generator admits a *linear parametrization* when $\mathcal{L}_t f(x) = \langle \mathcal{K}_{t,x} f, F_t(x)\rangle_{t,x}$, where $\mathcal{K}_{t,x}$ is a map from the space of test functions to some vector space $V_{t,x}$ and represents the structural part of the generator, while the function $F_t(x)$ specifies the process dynamics. For a flow, F_t contains the velocity; for a diffusion, it may contain drift and diffusion parameters; for a process with finitely many possible jumps, it contains their rates.

During training one introduces a latent variable Z, chosen so that the conditional processes begins at the simple distribution and ends at the data distribution after marginalisation. Conditioned on a realisation z, the *conditional generator* $\mathcal{L}_t^z$ describes a process transporting one specific x_0 to one specific x_1, and its parametrisation $F_t^z(x)$ can be computed directly from z and the sampled state. In ordinary flow matching Z is essentially the pair of endpoints; in Branching Flows it additionally carries the branching structure relating them.

Under standard regularity conditions, posterior averaging over Z gives a generator for this marginal path: $\mathcal{L}_t f(x) = \mathbb{E}[\mathcal{L}_t^Z f(x) \mid X_t = x]$. A neural network is trained, with a Bregman divergence loss against $F_t^Z(x)$, learning $F_t(x) := \mathbb{E}[F_t^Z(x) \mid X_t = x]$, which allows learning the marginal generator by training against samples from the conditional process.

Branching Flows employs and extends Generator Matching in the following two ways: i) We explicitly describe how one can augment a base stochastic process such as a stochastic interpolant path with *splits* and *deletions* which respectively increase or decrease the number of elements in the state and ii) we prove that when augmenting generation with a latent discrete stochastic process these auxiliary dynamics may be marginalised out and ignored during inference, extending the static latent variable theorems from [Holderrieth et al., 2024] as well as the discrete only theorems of [Havasi et al., 2025].

The augmentation of base processes with splits and deletions works in the following way: A forest of trees is used to couple a sample from a simple distribution (with elements at the roots) with the data distribution (elements at the leaves). Like the processes used in phylogenetics [Felsenstein, 1981], the elements evolve independently along each branch but duplicate and decouple at bifurcations in the tree. This construction, composed with optional deletions, allows us to train a generative model over sequences of varying length. We refer to figure 2 and appendix figure 6 for illustrations of the construction.

3.2 Branching Flows state space and coupling

In Branching Flows, the state at time t is an ordered variable-length list $X_t = (X_t^{(1)}, \ldots, X_t^{(L_t)}), X_t^{(i)} \in \mathcal{E}$, where L_t is the length at time t, and $\mathcal{E}$ may be Euclidean, discrete, a manifold, or a product of these. The state space is $\mathcal{X} := \bigsqcup_{n \geq 1} \mathcal{E}^n$. To construct a conditional path, draw $X_1 \sim p_{\text{data}}$ and then sample the static latent variable $Z = (X_1^{+\emptyset}, X_0, \mathcal{T}, \mathcal{A})$.

1. $X_1^{+\emptyset}$ is obtained by inserting elements into $X_1 \sim p_{\text{data}}$ and marking them for deletion.
2. X_0, the initial state, is drawn through a coupling whose marginal is p_{simple}, with $L_0 \leq L_1^{+\emptyset}$ (but note that L_0 can be greater than L_1 as long as the number of deletions is chosen appropriately).
3. The forest $\mathcal{T}$ contains one rooted binary tree for each element of X_0. Its roots are associated with the elements of X_0 and leaves associated one-to-one with the elements of $X_1^{+\emptyset}$. For an ordered sequence, the trees are planar and the leaf order matches the sequence order. For permutation equivariant models, there is no planar restriction on the trees. Leaves associated with inserted elements are marked for deletion; the remaining leaves survive to $t = 1$.
4. Anchors $\mathcal{A}$ (with $\mathcal{A}^{(i)} \in \mathcal{E}$) serve as intermediate generator targets along internal branches.

For concrete schemes for sampling Z we refer to appendix C

3.3 Conditional process and generator

Conditional on Z elements evolve from the roots toward the leaves of $\mathcal{T}$. Identify each branch by the node v ahead of it. Let $w(v)$ be the number of leaves descended from v, let $a(v)$ be the anchor at v, let $\delta(v) \in \{0, 1\}$ indicate that v is a deleted leaf, and write $v^\uparrow, v^\downarrow$ for the children of an internal node.

Anchors steer along internal branches. Between split and deletion events, each element evolves independently of the others along its branch, governed by a "base" generator, which can be any element-wise conditional generator from GM. In fixed-length flow matching this generator is conditioned on the endpoint $x_1^{(i)}$ of the element, and we write $\mathcal{L}_t^{\text{base}}(\cdot \mid e)$ for the generator conditioned to terminate at $e \in \mathcal{E}$. For Branching Flows, an element of X_t cannot always be conditioned on an element of x_1, because, prior to its final split, it is ancestral to several of them. Anchors supply the target instead. The anchor $a(v)$ of a node v is an element of $\mathcal{E}$ that serves as the target of the base generator along the branch leading to v. At a leaf, the anchor is the element of $X_1^{+\emptyset}$ associated

with that leaf; at internal nodes it can be chosen freely. An element on branch v evolves under $\mathcal{L}_t^{\text{base}}(\cdot \mid a(v))$: it would terminate at $a(v)$ were it to reach $t = 1$ without splitting or being deleted. An element on an internal branch instead splits before $t = 1$, after which its two copies are steered toward the anchors of the two child branches. The base generator may be any GM process for which the required conditional states can be sampled.

Parameterizing split and deletion rates. Choose absolutely continuous split- and deletion-time distributions H_{split} and H_{del} on $[0, s_b]$ with $s_b \leq 1$ the upper bound on the event time, $b \in \{\text{split}, \text{del}\}$. With $\xi_b(t)$ denoting the density, and with cumulative distribution functions satisfying $\Xi_b(t) < 1$ for $t < s_b$ and $\Xi_b(s_b) = 1$, we define the event hazards $h_b(t) = \xi_b(t)/(1 - \Xi_b(t))$ for $t < s_b$ and $h_b(t) = 0$ otherwise. Consequently, each required event occurs before $t = s_b$ almost surely.

Let G_t record the current branch of every element, and let v_i be the branch occupied by element i when $(Z, G_t) = (z, g)$. Define $R_{t,i}^{z,g}(x) := w(v_i) - 1$, $\rho_{t,i}^{z,g}(x) := \delta(v_i)$, where $w(v_i)$ is the number of leaves on the subtree below node v_i (so $w(v_i) - 1$ is the number of splits remaining in the subtree ahead of element i) and $\delta(v_i)$ indicates whether the current branch terminates in deletion. The corresponding instantaneous event rates are $h_{\text{split}}(t)R_{t,i}^{z,g}(x)$ and $h_{\text{del}}(t)\rho_{t,i}^{z,g}(x)$. When element i splits, it is replaced in place by two copies, each assigned to a descendant branch of v_i. When it is deleted, it is removed. All required splits and deletions occur by $t = 1$, and every surviving leaf has its associated target element as anchor. Hence the conditional process starts at X_0 and terminates at X_1 almost surely. Closed-form sampling of the next event time, together with a tractable base-process bridge, allows $X_t, G_t \mid Z$ to be sampled recursively in time linear in the number of leaves. We refer to Appendix E.3 and Algorithm E.1 for further details.

The Conditional Branching Flows Generator Let $\mathcal{L}_t^{z,g,\text{branch}}$ denote the conditional infinitesimal generator, $\text{split}_i(x)$ duplicate the i-th element in place and let $\text{del}_i(x)$ remove the i-th element. Let $\mathcal{L}_t^{z,g,\text{base}}$ be the sum of the per-element base generators. For $x \in \mathcal{E}^n$, the conditional generator acting on functions of X_t is

$$\begin{aligned}\mathcal{L}_t^{z,g,\text{branch}} f(x) = \mathcal{L}_t^{z,g,\text{base}} f(x) + h_{\text{split}}(t) \sum_{i=1}^{n} R_{t,i}^{z,g}(x)[f(\text{split}_i(x)) - f(x)] \\ + h_{\text{del}}(t) \sum_{i=1}^{n} \rho_{t,i}^{z,g}(x)[f(\text{del}_i(x)) - f(x)].\end{aligned} \tag{1}$$

The three terms respectively govern element values, splits, and deletions. Suppose the base generator is linearly parametrized by $F_t^{z,g,\text{base}}(x)$. Then the conditional generator above is linear in $F_t^{z,g,\text{branch}}(x) := (R_t^{z,g}(x), \rho_t^{z,g}(x), F_t^{z,g,\text{base}}(x))$.

The fixed structural part of this parametrization consists of the base operator, the two jump operators, and the hazard functions. The learnable targets are therefore the base-process targets, remaining split counts, and deletion indicators. We refer to appendix G for the fully general formulation of Branching Flows in Generator Matching, which allows insertions to be sampled from an arbitrary distribution conditioned on current time and state, instead of merely duplicating an existing node.

3.4 The marginal process

Let p_t denote the marginal law of X_t obtained by drawing $Z \sim p_Z$ and $(X_t, G_t) \sim p_{t|Z}$. The following result extends the static-latent Generator Matching argument to the evolving auxiliary variable G_t.

Theorem 3.1. *Let $\mathcal{L}_t^{z,g,\text{branch}}$ be linearly parametrized by $F_t^{z,g,\text{branch}}$ (cf. 1) . Then, the marginal generator*

$$\mathcal{L}_t^{\text{branch}} f(x) := \mathbb{E}_{(G_t,Z)\sim p(\cdot|x)}[\mathcal{L}_t^{Z,G_t,\text{branch}}(f)(x)]$$

generates a stochastic process X_t with marginals p_t and is linearly parametrized by

$$F_t(x) = \mathbb{E}[F_t^{Z,G_t,\text{branch}}(x) \mid X_t = x].$$

Proof. See Appendix G.4.1. □

Because every conditional path terminates at its sampled X_1, the marginal process satisfies $p_1 = p_{\text{data}}$. Trees, anchors, and branch indicators are therefore required to construct conditional training paths but not to sample from the learned marginal process. Since the branch indicators G_t evolve over time, the

theorem extends the static-latent result of Holderrieth et al. [2024] and the discrete auxiliary-process result of Havasi et al. [2025].

3.5 Loss and training

We now introduce the model parametrisation and corresponding conditional loss, and state that the conditional and marginal losses agree up to a θ-independent constant. Proofs are in appendix G.

Definition 3.2. Let the model output $F_t^{\theta,\text{base}}(x)$, $R_t^\theta(x) \in (0,\infty)^n$, $\rho_t^\theta(x) \in (0,1)^n$ for $x \in \mathcal{E}^n$. Let $D_{t,x}^{\text{base}}$, $D_{t,x}^{\text{split}}$, and $D_{t,x}^{\text{del}}$ be Bregman divergences for the base parameters, remaining split counts, and deletion indicators. These divergences may depend on time and state [Lipman et al., 2024, Billera et al., 2026]. The conditional Branching Flows loss is

$$L_{\text{CBF}}(\theta) := \mathbb{E}_{t,Z,X_t,G_t}\big[D_{t,X_t}^{\text{base}}(F_t^{Z,G_t,\text{base}}(X_t), F_t^{\theta,\text{base}}(X_t)) + D_{t,X_t}^{\text{split}}(R_t^{Z,G_t}(X_t), R_t^\theta(X_t)) + D_{t,X_t}^{\text{del}}(\rho_t^{Z,G_t}(X_t), \rho_t^\theta(X_t))\big].$$

The expectation draws t, then Z, and then $(X_t, G_t) \sim p_{t|Z}$. Only X_t is supplied to the model; Z and G_t are used to construct the targets.

In our experiments, the split loss is the elementwise Poisson negative log-likelihood, up to target-only constants,

$$D_{t,X_t}^{\text{split}} = \sum_{i=1}^{L_t}\big[R_{t,i}^\theta(X_t) - R_{t,i}^{Z,G_t}(X_t)\log R_{t,i}^\theta(X_t)\big],$$

and the deletion loss is elementwise binary cross-entropy,

$$D_{t,X_t}^{\text{del}} = -\sum_{i=1}^{L_t}\big[\rho_{t,i}^{Z,G_t}(X_t)\log\rho_{t,i}^\theta(X_t) + (1-\rho_{t,i}^{Z,G_t}(X_t))\log(1-\rho_{t,i}^\theta(X_t))\big].$$

Definition 3.3. Define the posterior-averaged targets $F_t^{\text{base}}(x) = \mathbb{E}[F_t^{Z,G_t,\text{base}}(x) \mid X_t = x]$, $R_t(x) = \mathbb{E}[R_t^{Z,G_t}(x) \mid X_t = x]$, $\rho_t(x) = \mathbb{E}[\rho_t^{Z,G_t}(x) \mid X_t = x]$. Let L_{BF} be the corresponding marginal loss: the same sum of divergences as above, with these posterior-averaged targets and an expectation only over (t, X_t).

Theorem 3.4. *Let L_{BF}, L_{CBF} be as in Definition 3.2 and 3.3. Then, we have*

$$\nabla_\theta L_{CBF}(\theta) = \nabla_\theta L_{BF}(\theta).$$

Proof. See Appendix G.4.2. □

For training, sample t, X_1, Z, draw $(X_t, G_t)|Z$ with Algorithm E.1, recover the three targets from the occupied branches, and take a gradient step. By theorem 3.4 this matches the gradient of the marginal objective, encouraging the model to learn the posterior-averaged generator from 3.1.

3.6 Sampling

Starting from $X_0 \sim p_{\text{simple}}$, we discretize the learned marginal process. At each step, the model predicts the base-process parameters, expected remaining split count, and deletion probability for every element. Each element is advanced under the learned base generator, and split and deletion events are sampled with intensities $h_{\text{split}}(t)R_{t,i}^\theta(X_t)$ and $h_{\text{del}}(t)\rho_{t,i}^\theta(X_t)$. A split duplicates the element in place; a deletion removes it (Alg. E.2). No forest, anchor, or branch indicator is present during inference. The final length is determined by the realized split and deletion events rather than specified in advance. We further note that, in general, the wall clock computational cost of sampling is primarily determined by number of tokens processed, and there is little overhead stemming from splits and deletions (see appendix H.6)

4 Empirical Results

In order to evaluate the performance of Branching Flows on situations where length variation is crucial we consider the following four tasks: i) Atom position and type generation for small molecules (continuous and discrete), ii) Antibody heavy chain generation (discrete sequence only), iii) Protein backbones, comprising amino acid backbone 'frames' [Jumper et al., 2021] with Euclidean positions, $SO(3)$ orientations and discrete amino acid labels (continuous, manifold and discrete) iv) Partially-latent all-atom protein structure and sequence co-design (continuous).

QM9. QM9 [Ramakrishnan et al., 2014] contains small organic molecules selected from the GDB (Generated Database) chemical space enumeration. We train Branching Flows to jointly generate atom coordinates and discrete labels; full data preprocessing, Branching Flows specification, and model/training details are in Appendix H.1. Figure 3 compares generated and data distributions (10k samples each) for molecule fingerprints, which encode molecule properties into a fixed-length bit vector, embedded into two-dimensions by UMAP [McInnes et al., 2020], showing no obvious differences. Also shown are histograms of atom counts and molecule properties, with some subtle departures, most visible in LogP and molecular weight. Using a Kolmogorov-Smirnov based statistic ($1 - KS_D$; see App. H.1) to quantify property agreement between samples and the data distribution (Figure 4), we also compare 10k samples from a fixed length oracle (matched in architecture, process, and training) which is provided with the correct sequence length at training time, and a sample from the data length at generation time; a fixed-length baseline ADiT Joshi et al. [2025]; and from a QM9-trained 'Transdimensional Jump' diffusion model [Campbell et al., 2023].

Further, we use QM9 to explore the behavior of Branching Flows across the design space, modifying the way the trees and anchors are sampled, modifying the X_0 size distribution, ablating splits, ablating (and oversampling) deletions, modifying the atom ordering, and considering fully permutation invariant model variants. As shown for a subset of models in Figure 3 and a larger set in SI Fig. 11, performance is robust across this large theoretically-valid design space, except where choices are made to be intentionally challenging for the model (e.g. randomized atom ordering). Further, the large design space allows domain-respecting choices that provide minor performance advantages.

We also consider a 'molecule completion' task, where some atoms are fixed (see SI section H.1.3). In this instance, the Oracle length model has information that simply isn't available at sampling time, because, conditioned on a subset of the molecule, the number of remaining atoms for each designable portion of the molecule isn't known. Here, Branching Flows again performs close to this oracle, and outperforms (especially in distribution matching) a fixed-length model that attempts to use a simple heuristic length sampling strategy.

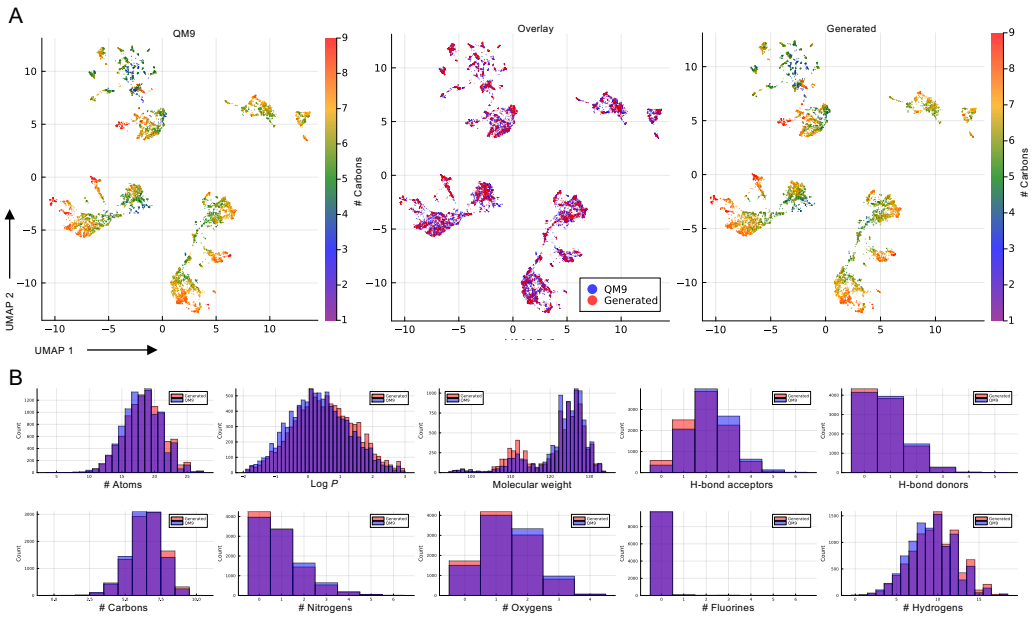


Figure 3: **QM9 data vs Branching Flows samples.** Panel A computes molecular fingerprints (using RDKit) and jointly embeds QM9 data and generated samples using UMAP. Left shows QM9-only embeddings, colored by the number of carbon atoms in each molecule; right shows generated samples, similarly colored, and center shows the overlap of QM9 (blue) vs Branching Flows generated samples (red), showing similar distributions. Panel B depicts property distributions from QM9 and Branching Flows generated samples, including atom counts, as well as a number of standard molecular descriptors computed with RDKit.

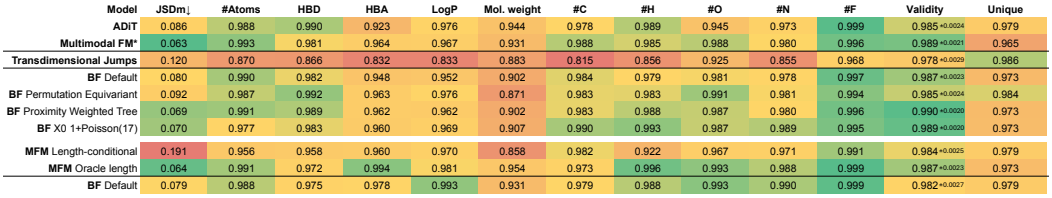


Figure 4: **QM9: model behavior.** Here we compare samples generated by Branching Flows (including a 'Default' variant, and a number of variants described in App H.2). The top 7 rows show unconditional molecule generation. We compare to fixed-length baselines ADiT [Joshi et al., 2025] and an architecture, process, and training matched fixed-length Multimodal Flow Matching (MFM) variant, which are both provided with the true length, and then with a variable length Transdimensional Jump Diffusion [Campbell et al., 2023]. The bottom three rows show results from a 'molecule completion' task where part of the molecule is fixed. Here we compare to a matched Oracle Length MFM model (note: this has length information that would be unavailable for 'completion' tasks) and an MFM model that attempts to draw the completion length so that the marginal length distribution is matched. Distribution-matching comparisons include a binned multivariate Jensen-Shannon divergence (JSDm), and 1-KS statistics (used for columns 2 to 10).

Antibody Sequences. We investigate a discrete sequence learning task: antibody amino acid sequence generation. We compare a Branching Flows model to an 'oracle-length' discrete flow matching model, which is trained knowing the true length, and sampled using a 'true' length from the data distribution. We also compare to a re-implementation of Edit Flows [Havasi et al., 2025]. Dataset details, Branching Flows specification, and model/training details are in Appendix H.3. Figure 15 (Appendix) shows the distributions of statistics (App. H.3) from sequences generated by both Branching Flows and Edit Flows models, perplexity-over-time training dynamics, and seqUMAP [Hanke et al., 2022] representations of the 10 000 generated sequences. Overall, Branching Flows is close to the oracle length model, can learn sequence-position-specific amino acid frequencies (Figure 14, Appendix), and outperforms Edit Flows on property matching (Table 1 and Figure 15, Appendix).

Similar to QM9, we also consider a conditional 'completion' task for antibody sequences, where the model is required to fill in missing segments when their length is unknown. Here we similarly outperform Edit Flows, with performance closer to the Oracle Length model (which, again, uses information not usually available at inference time).

Protein Structure and Sequence. Starting with ChainStorm [Oresten et al., 2025], a multimodal flow matching model that jointly generates protein backbones and amino acid sequence labels, as a base model, we finetuned two variants: 'BranchChain', where the designable residues are complete chains, and BranchSegment which designs random-length contiguous segments (see App. H.4). Figure 5 shows that BranchChain samples backbones with protein-like geometry, consistently recovered by the Boltz2 folding model when provided with a pseudo multiple sequence alignment [Oresten et al., 2025], and Figure 17 (Appendix) shows how BranchSegment addresses the 'unknown length infix sampling' problem, by sampling antibody CDR3s of variable length (see Appendix H.4).

Table 1: Distribution agreement ($1 - \mathrm{KS}_D$, higher is better) between properties of samples from each model vs. validation sequences. From left to right: Sequence Length, Diversity (within-samples min Cosine distance), Novelty (minimum edit distance to true sequence), CDR3 length, and AbNatiV 'humanness' score. Histograms of these properties are shown in Figure 13 (Appendix). Shown are results for an unconditional task where the entire sequence is generated, and a conditional task that generates multiple missing segments of the sequence. For the conditional task the 'Oracle length' model knows the length of each missing segment.

	Method	**Length**	**Diversity**	**Novelty**	**CDR3 length**	**AbNatiV**
Unconditional	Oracle length	0.995	0.899	0.903	0.984	0.943
	Branching Flows	0.986	0.894	0.888	0.989	0.931
	Edit Flows	0.989	0.713	0.770	0.983	0.769
Conditional	Oracle length	0.986	0.972	0.956	0.987	0.988
	Branching Flows	0.976	0.942	0.917	0.978	0.981
	Edit Flows	0.931	0.885	0.869	0.968	0.886

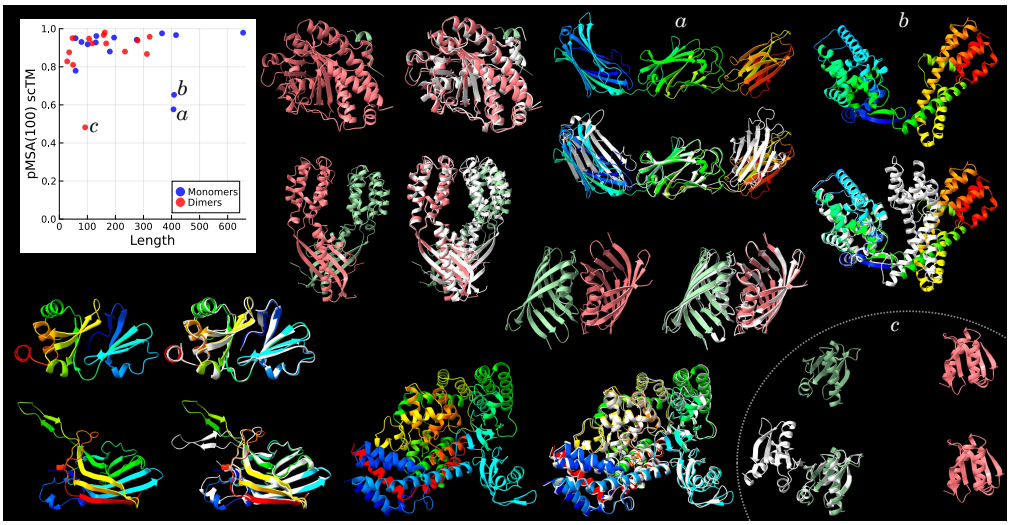


Figure 5: **Branching Flows protein samples.** Samples from BranchChain finetune model. Top left inset shows pseudoMSA scTM refolding scores, and selected samples are shown (rainbow colored for single chains, and per-chain for dimers). To the right or below each is shown an overlay of the sample and the refolded structure (grey). Specifically, the three worst scTMs are depicted (labelled a, b, and c on the inset and the main panel) showing that the low scTMs are driven by unpredictability due to flexible linkers, or incorrect dimerization prediction.

Partially-latent Protein Structure La Proteina [Geffner et al., 2025] uses "partially-latent" Flow Matching where each residue in a protein chain is represented by its C-alpha coordinates, and an 8-dimensional latent encoding the residue label and the placement of other atoms, obtained via a Variational Autoencoder. We finetune Branching Flows into this by introducing split and deletion heads into the LD3 La Proteina model. Table 2 shows that the Branching Flows finetune achieves excellent designability (via an MPNN-generated sequence), co-designability (sequence via the sampled latents), and diversity metrics.

Computational overhead and timing In all cases presented herein the computational overhead from the branching mechanism during both training and sampling is negligible, and wall clock time is straightforwardly predicted by the number of tokens processed by the models. For empirical results pertaining to timing we refer to Appendix H.6

5 Discussion

We have shown that Branching Flows is a capable 'distribution-learner' of multimodal sequences that vary in length, addressing a previously-unsolved fundamental problem in generative modeling. For example it is, to our knowledge, the first method that can sample an infix of unknown length from a multimodal sequence. We demonstrated that Branching Flows works in a variety of domains, outperforming other domain-specific variable length flow and diffusion models, and we have explored the behavior and performance across the Branching Flows design space, showing that it is robust to design choices. For unconditional generation, we show that we can approximately match fixed length models without needing to provide the length. For conditional completion, the missing-length distribution is generally unavailable, and on these tasks Branching Flows outperforms Edit Flows and approaches oracle-length models (which are given information that is not available for such tasks).

Table 2: **Branching La Proteina metrics.** Designability, co-designability, and diversity (structure only, and structure & sequence) for $N = 500$ samples. Intervals are symmetric two-sigma Wald intervals for binomial proportions.

Sample set	MPNN1 designability	Co-designability	Div. str	Div. str+seq
Branching Flows LD3	0.990 ± 0.009	0.966 ± 0.016	128	215
La Proteina LD3	0.972 ± 0.015	0.952 ± 0.019	110	173

For the QM9 small-molecule dataset, we compare samples from our Branching Flows-trained model with those from the Transdimensional Jump Diffusion model of Campbell et al. [2023], showing favorable distribution matching for Branching Flows. We suspect this difference is attributable to two things. Firstly, in transdimensional jump diffusion, while the conditional distribution of the element spawn point is Gaussian conditioned on the corresponding X_1 element, the marginal distribution during inference is a (possibly infinite) mixture of Gaussians, which is challenging to approximate. Branching Flows avoids this by spawning new elements via splitting. Secondly, transdimensional jump diffusion handles discrete atom types by converting them to continuous vectors. In other contexts, this does not compare favorably to discrete character diffusion models [Sahoo et al., 2024].

For the antibody sequence modeling problem, we compare Branching Flows to Edit Flows [Havasi et al., 2025]. Here the model architectures are matched between Branching Flows and Edit Flows, where the only difference is that the Edit Flows insertion head is as large as the alphabet size, whereas Branching Flows only needs one splitting rate per element. While we see favorable performance of Branching Flows, one caveat is that this is our own Edit Flows implementation for this task, rather than some pre-existing baseline.

For generative models of protein structure and sequence, it is practically notable that Branching Flows split and deletion heads can be finetuned into existing pretrained fixed-length models at very low cost (one or two days on a single consumer GPU), suggesting that the representations required for learning variable-length design are already, to some degree, present in fixed-length models.

The design space of fixed-length element-wise Generator Matching approaches is already large (encompassing almost all diffusion and flow matching methods), and Branching Flows layers element splitting and deletion atop this, necessarily expanding the design space further.

Some design choices are structurally dictated by the task. For example, if the samples are ordered sequences, then the sampled trees must be planar (ie. coalescing only adjacent elements), but if the samples are sets of elements being modelled with a permutation-equivariant model, then no such constraint is required. Further, the trees can enforce useful structural constraints: for example, if you are modeling protein sequences with multiple chains and you wish to directly control the number of chains, then during training you prevent the coalescence of any elements from different chains.

Then there are choices that are not fully dictated by the task, but where there is often a predictable inductive benefit, such as averaging for anchor choices in most continuous tasks, or preferring to merge elements that are spatially close during tree construction, both of which reduce average drift.

Just as in standard diffusion and flow matching, however, there are also choices that are not dictated by the problem structure at all, such as the hazard distributions (where mostly splits very close to $t = 1$ should be avoided), and choices that involve computational tradeoffs, such as increasing the X_0 length, increasing the volume of deletions, or using a pure-deletion model (which all increase the X_t length and thus, quadratically, the training compute). For further discussion, refer to appendix C.5.

Overall, we hope that the availability of a robust framework for constructing theoretically-justified variable-length models will spur development across domains wherever the requirement of *a priori* known fixed-lengths are a hinderance.

6 Limitations

While we have empirically characterized the behavior of Branching Flows across a substantial part of the design space on QM9, showing that it is generally well-behaved, we don't have a formal theory to guide these choices, and it is beyond the scope of this work to perform similar studies across the other domains (discrete antibody sequences and protein sequence and structure). While we have clear metrics when investigating the behavior of 'completion' tasks, where Branching Flows is expected to have an advantage, for protein models we demonstrate this task (with ChainStorm) but lack metrics to evaluate it. Further, on tasks where sample annealing is typically required such as protein structure sampling, we have not systematically explored annealing strategies, and must leave this for future work. Next, for fixed-length flow matching models, approximate likelihoods can often be computed, which can be useful for some downstream tasks. We have not articulated how to calculate such likelihoods for Branching Flows.

Acknowledgments

This project received support from the Swedish Research Council (2024-00390 and 2023-02516) and the Knut and Alice Wallenberg Foundation (2024.0039) to B.M. Development of key dependencies used in this work was enabled by the Berzelius resource provided by the Knut and Alice Wallenberg Foundation at the National Supercomputer Centre.

References

Giuseppina Albano and Virginia Giorno. Inference on the effect of non homogeneous inputs in Ornstein-Uhlenbeck neuronal modeling. *Mathematical Biosciences and Engineering*, 17(1): 328–348, 2020. doi: 10.3934/mbe.2020018.

Lukas Billera, Anton Oresten, Aron Stålmarck, Kenta Sato, Mateusz Kaduk, and Ben Murrell. The continuous language of protein structure. *bioRxiv*, 2024. doi: 10.1101/2024.05.11.593685.

Lukas Billera, Hedwig Nora Nordlinder, and Ben Murrell. Time dependent loss reweighting for flow matching and diffusion models is theoretically justified. In *ICLR 2026 2nd Workshop on Deep Generative Model in Machine Learning: Theory, Principle and Efficacy*, 2026. URL https://openreview.net/forum?id=COJIuG0eRN.

Martin Buttenschoen, Garrett M. Morris, and Charlotte M. Deane. Posebusters: Ai-based docking methods fail to generate physically valid poses or generalise to novel sequences. *Chemical Science*, 15(9):3130–3139, 2024. ISSN 2041-6539. doi: 10.1039/d3sc04185a. URL http://dx.doi.org/10.1039/D3SC04185A.

Andrew Campbell, William Harvey, Christian Weilbach, Valentin De Bortoli, Tom Rainforth, and Arnaud Doucet. Trans-dimensional generative modeling via jump diffusion models. *arXiv*, 2023. doi: 10.48550/arXiv.2305.16261.

James Dunbar and Charlotte M. Deane. ANARCI: antigen receptor numbering and receptor classification. *Bioinformatics*, 32(2):298–300, 2016. doi: 10.1093/bioinformatics/btv552.

Joseph Felsenstein. Evolutionary trees from DNA sequences: a maximum likelihood approach. *Journal of molecular evolution*, 17(6):368–376, 1981. doi: 10.1007/BF01734359.

Itai Gat, Tal Remez, Neta Shaul, Felix Kreuk, Ricky T. Q. Chen, Gabriel Synnaeve, Yossi Adi, and Y. Lipman. Discrete flow matching. *ArXiv*, abs/2407.15595, 2024. doi: 10.48550/arXiv.2407.15595.

Tomas Geffner, Kieran Didi, Zhonglin Cao, Danny Reidenbach, Zuobai Zhang, Christian Dallago, Emine Kucukbenli, Karsten Kreis, and Arash Vahdat. La-Proteina: Atomistic protein generation via partially latent flow matching, 2025. URL https://arxiv.org/abs/2507.09466.

Leo Hanke, Daniel J Sheward, Alec Pankow, Laura Perez Vidakovics, Vivien Karl, Changil Kim, Egon Urgard, Natalie L Smith, Juan Astorga-Wells, Simon Ekström, Jonathan M. Coquet, Gerald M. McInerney, and Ben Murrell. Multivariate mining of an alpaca immune repertoire identifies potent cross-neutralizing SARS-CoV-2 nanobodies. *Science advances*, 8(12):eabm0220, 2022. doi: 10.1126/sciadv.abm0220.

Marton Havasi, Brian Karrer, Itai Gat, and Ricky T. Q. Chen. Edit flows: Flow matching with edit operations. *ArXiv*, abs/2506.09018, 2025. doi: 10.48550/arXiv.2506.09018.

Peter Holderrieth, Marton Havasi, Jason Yim, Neta Shaul, Itai Gat, T. Jaakkola, Brian Karrer, Ricky T. Q. Chen, and Y. Lipman. Generator matching: Generative modeling with arbitrary Markov processes. *ArXiv*, abs/2410.20587, 2024. doi: 10.48550/arXiv.2410.20587.

Keller Jordan, Yuchen Jin, Vlado Boza, Jiacheng You, Franz Cesista, Laker Newhouse, and Jeremy Bernstein. Muon: An optimizer for hidden layers in neural networks, 2024. URL https://kellerjordan.github.io/posts/muon/.

Chaitanya K. Joshi, Xiang Fu, Yi-Lun Liao, Vahe Gharakhanyan, Benjamin Kurt Miller, Anuroop Sriram, and Zachary W. Ulissi. All-atom diffusion transformers: Unified generative modelling of molecules and materials, 2025. URL https://arxiv.org/abs/2503.03965.

John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596(7873):583–589, 2021. doi: 10.1038/s41586-021-03819-2.

Aleksandr Kovaltsuk, Jinwoo Leem, Sebastian Kelm, James Snowden, Charlotte M Deane, and Konrad Krawczyk. Observed antibody space: A resource for data mining next-generation sequencing of antibody repertoires. *The Journal of Immunology*, 201(8):2502–2509, 10 2018. doi: 10.4049/jimmunol.1800708.

Yeqing Lin and Mohammed AlQuraishi. Generating novel, designable, and diverse protein structures by equivariantly diffusing oriented residue clouds, 2023. URL `https://arxiv.org/abs/2301.12485`.

Yaron Lipman, Marton Havasi, Peter Holderrieth, Neta Shaul, Matt Le, Brian Karrer, Ricky T. Q. Chen, David Lopez-Paz, Heli Ben-Hamu, and Itai Gat. Flow matching guide and code. *ArXiv*, abs/2412.06264, 2024. doi: 10.48550/arXiv.2412.06264.

Leland McInnes, John Healy, and James Melville. UMAP: Uniform manifold approximation and projection for dimension reduction. *ArXiv*, 2020. doi: 10.48550/arXiv.1802.03426.

Anton Oresten, Kenta Sato, Aron Stålmarck, Lukas Billera, Hedwig Nora Nordlinder, Jack Collier Ryder, Mateusz Kaduk, and Ben Murrell. Spontaneous emergence of symmetry in a generative model of protein structure. *bioRxiv*, 2025. doi: 10.1101/2025.11.03.686219.

William Peebles and Saining Xie. Scalable diffusion models with transformers. *ArXiv*, 2023. doi: 10.48550/arXiv.2212.09748.

Raghunathan Ramakrishnan. QM9PACK: A python package for data-mining the qm9 dataset, 2024. URL `https://github.com/raghurama123/qm9pack`.

Raghunathan Ramakrishnan, Pavlo O. Dral, Matthias Rupp, and O. Anatole von Lilienfeld. Quantum chemistry structures and properties of 134 kilo molecules. *Scientific Data*, 1, 2014. doi: 10.1038/sdata.2014.22. Article Number: 140022.

Aubin Ramon, Montader Ali, Misha Atkinson, Alessio Saturnino, Kieran Didi, Cristina Visentin, Stefano Ricagno, Xing Xu, Matthew Greenig, and Pietro Sormanni. Assessing antibody and nanobody nativeness for hit selection and humanization with AbNatiV. *Nature Machine Intelligence*, 6:74–91, 2024. doi: 10.1038/s42256-023-00778-3.

Subham Sekhar Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin T Chiu, Alexander Rush, and Volodymyr Kuleshov. Simple and effective masked diffusion language models. *ArXiv*, 2024. doi: 10.48550/arXiv.2406.07524.

Arne Schneuing, Ilia Igashov, Adrian W. Dobbelstein, Thomas Castiglione, Michael Bronstein, and Bruno Correia. Multi-domain distribution learning for de novo drug design. *ArXiv*, abs/2508.17815, 2025. doi: 10.48550/arXiv.2508.17815.

Dmitry V Shcheblyakov, Irina A Favorskaya, Inna V Dolzhikova, Anastasia I Korobkova, Irina A Alekseeva, Ilias B Esmagambetov, Olga L Voronina, Amir I Tukhvatulin, Olga V Zubkova, Artem A Derkaev, Ekaterina I. Ryabova, Anna A. Iliukhina, Ilia D. Zorkov, Daria M. Grousova, Dmitriy A. Reshetnikov, Natalia N. Ryzhova, Ekaterina I. Ermolova, Marina S. Kunda, Ilya O. Matyuta, Konstantin M. Boyko, Vladimir O. Popov, Denis Y. Logunov, Nikolai N. Sluchanko, and Alexander L. Gintsburg. Ultra-potent RBM-specific single-domain antibody broadly neutralizes multiple SARS-CoV-2 variants with picomolar activity. *International Journal of Biological Macromolecules*, 319(Part 3):145386, 2025. doi: 10.1016/j.ijbiomac.2025.145386.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. RoFormer: Enhanced transformer with rotary position embedding. *ArXiv*, 2023. doi: 10.48550/arXiv.2104.09864.

Oliver M Turnbull, Dino Oglic, Rebecca Croasdale-Wood, and Charlotte M Deane. p-IgGen: a paired antibody generative language model. *Bioinformatics*, 40(11):btae659, 11 2024. doi: 10.1093/bioinformatics/btae659.

Michel van Kempen, Stephanie S. Kim, Charlotte Tumescheit, Milot Mirdita, Jeongjae Lee, Cameron L. M. Gilchrist, Johannes Söding, and Martin Steinegger. Fast and accurate protein structure search with Foldseek. *Nature Biotechnology*, 42(2):243–246, 2024. doi: 10.1038/s41587-023-01773-0. URL `https://doi.org/10.1038/s41587-023-01773-0`.

Appendix Contents

A Broader Impacts **14**

B LLM usage **15**

C Z sampling schemes **15**

C.1 Initial Distribution: $X_0 \sim \mathcal{D}_0|(X_1 = x_1)$ 15

C.2 Deletions: $X_1^{+\emptyset} \sim \mathcal{D}_\emptyset|(X_1 = x_1, X_0 = x_0)$ 15

C.3 Forest: $\mathcal{T} \sim \mathcal{D}_\mathcal{T}|(X_1^{+\emptyset} = x_1^{+\emptyset}, X_0 = x_0)$ 16

C.4 Internal Anchors: $\mathcal{A}^\mathcal{I} \sim \mathcal{D}|(X_1^{+\emptyset} = x_1^{+\emptyset}, X_0 = x_0, \mathcal{T})$ 16

C.5 Structural constraints, inductive biases, and practical notes 16

D Base processes **17**

D.1 The Ornstein-Uhlenbeck process with Time Dependent Diffusion Coefficient . . . 17

D.2 Conditional and Marginal Paths for a Noisy Discrete Process 17

E Waiting times, conditional-path sampling, and algorithms **18**

E.1 Waiting-time results 18

E.2 Application to split and deletion branches 20

E.3 Conditional-state sampling 20

E.4 Conditional sampling, training, and marginal sampling algorithms 20

F Branch tracking, splits, and deletions **22**

G Branching Flows in Generator Matching **23**

G.1 The Infinitesimal Generator of Jump Processes 23

G.2 The Marginal Branching Infinitesimal Generator 23

G.3 The Branching Flows Loss 25

G.4 Proofs 25

H Supplementary Empirical Details **29**

H.1 QM9 analysis 29

H.2 Branching Flows QM9 variations and ablations 32

H.3 Antibodies 34

H.4 Proteins 38

H.5 La Proteina Branching Flows fine-tuning and evaluation 40

H.6 Training and Inference Timing 42

A Broader Impacts

This paper presents work whose goal is to advance the field of Generative Modeling. We anticipate Branching Flows will be especially relevant for scientific domains such as those studied here, including generative modeling of small molecules and proteins, which have applications in medicine and biotechnology. Though we have not explored it, Branching Flows may also prove useful in other domains, such as video generation, which may have positive and negative social consequences. As this work primarily aimed to demonstrate the feasibility of this new approach, and there is no

reason to expect any models trained herein to pose any specific risks or benefits beyond those already developed by the field, but the Branching Flows framework might enable such models in the future.

B LLM usage

Large Language Models were used in the writing alongside humans to help identify issues in mathematical proofs and derivations. Large Language Models coding assistants were used for routine coding tasks and experimental setup. All mathematical derivations were double-checked by the authors.

C Z sampling schemes

Here we outline concrete schemes for sampling $Z := (X_1^{+\emptyset}, X_0, \mathcal{T}, \mathcal{A})$, starting from X_1, that we use in our empirical examples below. Some datasets have elements that are naturally grouped (e.g. proteins, grouped by chain), and one can construct Z to respect these groupings, where it makes sense to do so. In the case where there is only one group, all elements have the same group index. Further, if any elements are 'fixed' (e.g. in the protein example below, by a conditioning mask), then any elements with a fixed element between them in the sequence must belong to a different group.

- **Data:** $X_1 \sim \mathcal{D}_1$
  - $(x_1^{(1)}, x_1^{(2)}, x_1^{(3)}, x_1^{(4)})$
- **Initial:** $X_0 \sim \mathcal{D}_0 | (X_1 = x_1)$
  - $(x_0^{(1)}, x_0^{(2)})$
- **Data with deletions:** $X_1^{+\emptyset} \sim \mathcal{D}_\emptyset | (X_1 = x_1, X_0 = x_0)$
  - $(x_1^{+(1)} = x_1^{(1)}, x_1^{+(2)\emptyset}, x_1^{+(3)} = x_1^{(2)}, x_1^{+(4)} = x_1^{(3)}, x_1^{+(5)} = x_1^{(4)})$
- **Forest:** $\mathcal{T} \sim \mathcal{D}_\mathcal{T} | (X_1^{+\emptyset} = x_1^{+\emptyset}, X_0 = x_0)$
  - $((x_1^{+(1)}, x_1^{+(2)\emptyset}), (x_1^{+(3)}, (x_1^{+(4)}, x_1^{+(5)})))$
- **Internal anchors:** $\mathcal{A} \sim \mathcal{D}_\mathcal{A} | (X_1^{+\emptyset} = x_1^{+\emptyset}, X_0 = x_0, \mathcal{T})$
  - $(a_7 = m(x_1^{+(4)}, x_1^{+(5)}), \quad a_8 = m(x_1^{+(3)}, a_7), \quad a_6 = m(x_1^{+(1)}, x_1^{+(2)\emptyset}))$
- $Z := (X_1^{+\emptyset}, X_0, \mathcal{T}, \mathcal{A})$

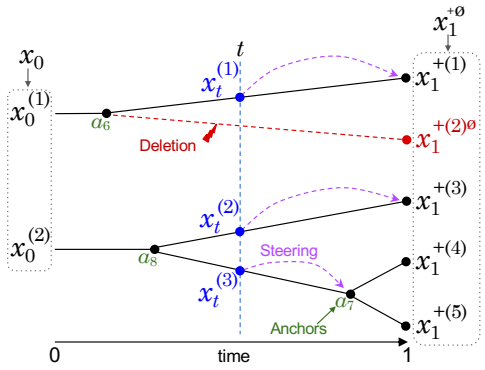


Figure 6: **Branching Flows construction.** Left outlines the sampling of Z when x_1 has 4 elements and x_0 has two elements, for a conditional bridge that will incur a single deletion (element $x_1^{+(2)\emptyset}$). $m(.,.)$ derives an anchor from two children, which could be e.g. a midpoint for continuous states and a mask token for discrete states. Right depicts the conditional sampling of $x_t | Z$, where elements can split and be deleted but only, in the conditional path, according to the pre-determined branching structure of the two trees $\mathcal{T}$ in Z.

C.1 Initial Distribution: $X_0 \sim \mathcal{D}_0 | (X_1 = x_1)$

To sample X_0 we need to sample the number of elements, and the state of each element. While in general the state of X_0 elements can be coupled to X_1, here we only concretely explore schemes where each X_0 element is independently sampled from a base distribution, which is a 'masked token' for all discrete state components, a standard isotropic Gaussian for all continuous components, and uniform samples on SO(3) for rotations.

We write ℓ_t^{group} for the realized number of elements in a group and $\ell_t = \sum_{\text{group}} \ell_t^{\text{group}}$ for the total length, a realization of L_t. To sample the X_0 length in each group, we use:

$$\ell_0^{\text{group}} \sim 1 + \text{Poisson}(\lambda_{\text{group}}),$$

where in many cases λ_{group} is 0, meaning X_0 starts with a single element (or one per group).

C.2 Deletions: $X_1^{+\emptyset} \sim \mathcal{D}_\emptyset | (X_1 = x_1, X_0 = x_0)$

Sampling $X_1^{+\emptyset} \sim \mathcal{D}_\emptyset | (X_1 = x_1, X_0 = x_0)$ includes how many 'to-be-deleted' elements are introduced, where they are introduced in the sequence, and what their state is, all which can be flexibly chosen. In what we think will provide a better learning target, we consider schemes where

elements of X_1 are duplicated, and their state copied. In some cases it also might make sense to modify the state, such as enforcing that 'to-be-deleted' discrete states are always masked tokens.

During training, after sampling X_1 and X_0, the X_1 state is augmented with "to be deleted" elements (denoted $X_1^{+\emptyset}$), such that it is at least as long as X_0.

To ensure this, one scheme we use is to specify a deletion padding multiplier $d_r \geq 1$ and, aiming for the expected total elements to be $\max(\ell_0^{\text{group}}, \ell_1^{\text{group}})d_r$, draw the number of to-be-deleted elements with

$$\text{del}_{\text{group}} \sim \max(\ell_0^{\text{group}}, \ell_1^{\text{group}}) - \ell_1^{\text{group}} + \text{Poisson}((d_r - 1)\max(\ell_0^{\text{group}}, \ell_1^{\text{group}}))$$

$\text{del}_{\text{group}}$ elements are then sampled, uniformly with replacement, from the elements from the matching group of X_1, and inserted, uniformly, either side of X_1 template element, inheriting its state.

C.3 Forest: $\mathcal{T} \sim \mathcal{D}_{\mathcal{T}}|(X_1^{+\emptyset} = x_1^{+\emptyset}, X_0 = x_0)$

We need ℓ_0^{group} trees in $\mathcal{T}$ (i.e. one tree per X_0 element). Since the trees must be planar for ordered sequences, we adopt a backward coalescence scheme that starts with the elements of $X_1^{+\emptyset}$ and uniformly selects adjacent pairs that belong to the same group, coalescing them and replacing the pair with a single node. This proceeds until there are only ℓ_0 remaining elements, which become the root of each tree that is associated, maintaining the order, with the elements of X_0. Note that other schemes are studied in the QM9 variations and ablations.

C.4 Internal Anchors: $\mathcal{A}^{\mathcal{I}} \sim \mathcal{D}|(X_1^{+\emptyset} = x_1^{+\emptyset}, X_0 = x_0, \mathcal{T})$

Denote by $\mathcal{A}^{\mathcal{I}}$ the collection of internal-node anchors in $\mathcal{A}$. Anchors control the conditional process along branches, guiding each element so that it would terminate, were t to reach 1, at the anchor. Along terminal branches, the anchors are equal to the elements of $X_1^{+\emptyset}$, but along internal branches we have freedom in how we construct them.

Consider a path along one tree from the root to a single leaf. An element drifts, along this path, towards one anchor after another, changing each time a split event occurs. If the anchors change too erratically, the process will be difficult to learn. One principle is to sample anchors that avoid, as much as possible, high-velocity changes.

Since anchors can be sampled conditioned on $X_1^{+\emptyset}$, one convenient approach is to specify an anchor merging distribution $p(a_{\text{parent}}|a_{\text{child}_1}, a_{\text{child}_2}, \mathcal{T})$ and recursively determine the anchors from leaves to root. This is depicted as $m(\cdot, \cdot)$ in figure 6. Since ordered sequences of continuous values (especially for spatial positions) often exhibit strong autocorrelation with similar values for neighboring elements but discrete values typically do not, the anchor merging schemes can differ depending on whether the state components are continuous or discrete. For our empirical investigations, we use:

- Continuous ($\mathbb{R}^n$): mean of a_{child_1} and a_{child_2}, weighted by the number of descendants of each child.
- Manifold: Geodesic interpolation between a_{child_1} and a_{child_2}, weighted by the number of descendants of each child.
- Discrete: All internal anchors are set as a mask/dummy token.

An alternative choice which we have explored in the QM9 variations study is copying a random child state (with probability weighted by some function of the number of descendants of each child).

C.5 Structural constraints, inductive biases, and practical notes

The tree construction provides substantial flexibility for adapting Branching Flows to respect structural constraints that may arise in generative modeling tasks.

If the elements form a sequence, then the trees must be planar (i.e. coalescence must be between neighbours in the sequence), so that the extra token that arises from a split event is in the right position. Our theoretical exposition assumes this is the case. However, if the elements are from an unordered set (as in the "equivariant" QM9 variation), then there is no planar restriction on merging when sampling the trees, and the theory extends straightforwardly to this case.

If there is some substructure in your sequences that you want to preserve (e.g. if you want to control the number of chains in a protein, as in the BranchChain example), then you can enforce this via

the trees: simply only coalesce elements within chains, but never between chains, using the 'groups' structure above.

If you wish to have some part of the sequence fixed length (e.g. in conditional design tasks), then disallow coalescence between any pair of nodes when one of them is in the fixed-length set.

Within these constraints, there are still many ways to sample tree couplings between X_0 and X_1. As we show in the QM9 variations, when elements are points in some space, sampling quality improves when tree neighbours tend to also be spatial neighbours (and we can explicitly bias the tree sampling to encourage this if we wish - see the Proximity Weighted variation). The worst case for a tree-like generative process over continuous elements is when elements form a sequence, but the spatial position of elements is completely unrelated to their position in the sequence. We show this for the deliberately-adversarial "random ordering" QM9 variation. Note that the theory is still valid here, but the marginal generator is very hard to learn.

Overall, a reasonable heuristic for the continuous case is "choose a tree and anchor sampling scheme that, if you place the x_1 elements on the leaf nodes and anchors on the internal nodes, will reduce the total sum of branch lengths across the tree" (where the branch length is just the distance between nodes).

When splitting elements, the operator we describe and implement in our examples involves duplicating the elements in-place in the sequence. The theory additionally covers duplicating and adding noise, or simply drawing new elements from some base distribution. Generally, for the conditional path construction, stochastically sampling new elements conditioned on X_t is non-problematic, but conditioning on X_1 can cause the marginal paths to require arbitrary mixture insertion distributions, which is to be avoided.

All examples with continuous states implemented here use stochastic conditional paths. Deterministic ODE continuous paths do not interact well with splitting elements by duplicating in-place, as this can require the model to learn a deterministic map from a low-dimensional space to a high dimensional one. If using ODE continuous paths, we suggest adding substantial noise upon each split event, but we have not studied this in depth, empirically.

D Base processes

A branch in the conditional path may begin at a random split time $s > 0$. We therefore need to sample the single-element base transition $Y_t \mid (Y_s = y_s, Y_1 = a)$ for $s < t < 1$, where a is the branch's anchor. At a split, both children inherit the same sampled value at time s and then evolve toward their respective anchors, with endpoint conditioning still at the absolute time 1. These transitions are used by the recursive conditional-path sampler in Appendix E.3.

D.1 The Ornstein-Uhlenbeck process with Time Dependent Diffusion Coefficient

Consider the stochastic differential equation

$$dY_t = \theta(\mu - Y_t)dt + \sqrt{v_t}\, dW_t, \tag{2}$$

where $\theta > 0$ and v_t is the time-dependent infinitesimal variance (so the diffusion coefficient is $\sqrt{v_t}$). We will, for some a, consider the time-inhomogeneous OU-bridge $Y_t \mid Y_1 = a$ where the unconditional process Y_t solves

$$dY_t = \theta(a - Y_t)dt + \sqrt{v_t}dW_t. \tag{3}$$

A slightly more general problem is studied in Albano and Giorno [2020], specifically of the form (See their equation (2.4))

$$dY_t = \left(-\frac{Y_t}{\nu} + \mu + m_t\right) dt + \sqrt{v_t}dW_t, \tag{4}$$

where explicit formulas for the transition densities (which remain Gaussian) are given.

D.2 Conditional and Marginal Paths for a Noisy Discrete Process

In what follows, we use an extension of Discrete Flow Matching (DFM Gat et al. [2024]) to allow time intervals $[t_0, 1]$, with $0 \le t_0 < 1$. Let $\mathcal{E}_{\text{disc}}$ be the finite single-element state space, and let p_u denote the probability mass function of the uniform-noise component on it. Let Ξ_1 and Ξ_2 be CDFs of some continuous distributions supported on $[0, 1]$, and let $\omega_u \in [0, 1)$. Then we define the

schedulers

$$\kappa_t^1 = \Xi_1(t), \qquad \kappa_t^2 = \omega_u(1-\Xi_1(t))\Xi_2(t), \qquad \kappa_t^3 = 1-\kappa_t^1-\kappa_t^2.$$

The ratio expressions below are understood where the displayed denominators are positive and the indicated derivatives exist. The path assumes $\kappa_{t_0}^3 > 0$, and the rate expressions are used at $t < 1$ with $\kappa_t^3 > 0$.

It can be shown that our choice of schedulers enforces $\ell_\kappa := \operatorname{argmin}_{j\in\{1,2,3\}} \frac{\dot\kappa_t^j}{\kappa_t^j} = 3$. With these schedulers, define the interpolant path

$$p_t(x|x_{t_0},x_1) := (\kappa_t^1 - \kappa_{t_0}^1\frac{\kappa_t^3}{\kappa_{t_0}^3})\delta_{x_1}(x) + (\kappa_t^2 - \kappa_{t_0}^2\frac{\kappa_t^3}{\kappa_{t_0}^3})p_u(x) + \frac{\kappa_t^3}{\kappa_{t_0}^3}\delta_{x_{t_0}}(x)$$

between x_{t_0} and x_1 in the time-interval $[t_0, 1]$, with terminal law $p_1(\cdot \mid x_{t_0}, x_1) = \delta_{x_1}$. This path is generated by the rate matrix

$$q_t(x; y \mid x_{t_0}, x_1) = (\dot\kappa_t^1 - \kappa_t^1\frac{\dot\kappa_t^3}{\kappa_t^3})\delta_{x_1}(x) + (\dot\kappa_t^2 - \kappa_t^2\frac{\dot\kappa_t^3}{\kappa_t^3})p_u(x) + \frac{\dot\kappa_t^3}{\kappa_t^3}\delta_y(x).$$

Here $x, y \in \mathcal{E}_{\text{disc}}$ are the destination and current states, respectively.

The rate expression follows from applying DFM Theorem 3 on the interval $[t_0, 1]$. To see this, define $\alpha_t^1 = \kappa_t^1 - \kappa_{t_0}^1\frac{\kappa_t^3}{\kappa_{t_0}^3}$, $\alpha_t^2 = \kappa_t^2 - \kappa_{t_0}^2\frac{\kappa_t^3}{\kappa_{t_0}^3}$ and $\alpha_t^3 = \frac{\kappa_t^3}{\kappa_{t_0}^3}$. It can be seen that $\ell_\alpha := \operatorname{argmin}_{j\in\{1,2,3\}} \frac{\dot\alpha_t^j}{\alpha_t^j} = 3$, along with $\alpha_t^1, \alpha_t^2, \alpha_t^3 \geq 0$ and $\sum_{i=1}^3 \alpha_t^i = 1$. The expression for the rates follows from

$$\begin{aligned}\dot\alpha_t^1 - \alpha_t^1\frac{\dot\alpha_t^3}{\alpha_t^3} &= (\dot\kappa_t^1 - \kappa_{t_0}^1\frac{\dot\kappa_t^3}{\kappa_{t_0}^3}) - (\kappa_t^1 - \kappa_{t_0}^1\frac{\kappa_t^3}{\kappa_{t_0}^3})\frac{\dot\kappa_t^3}{\kappa_t^3} \\ &= \dot\kappa_t^1 - \kappa_t^1\frac{\dot\kappa_t^3}{\kappa_t^3}\end{aligned}$$

and analogously

$$\dot\alpha_t^2 - \alpha_t^2\frac{\dot\alpha_t^3}{\alpha_t^3} = \dot\kappa_t^2 - \kappa_t^2\frac{\dot\kappa_t^3}{\kappa_t^3}.$$

E Waiting times, conditional-path sampling, and algorithms

E.1 Waiting-time results

Let H be a hazard distribution with no atom at 1 supported on $[0, 1]$, implying that $S_H(1) = 1 - \Xi_H(1) = 0$. Since $S_H(t) = 1 - \Xi_H(t)$ is the probability of the event not occurring by time t, this implies the event occurs almost surely by $t = 1$. The hazard rate associated to H is given by

$$h(t) = \frac{\xi_H(t)}{1 - \Xi_H(t)}.$$

We also note that since

$$\int_0^t h(s)\, ds = -\log(1 - \Xi_H(t)),$$

a hazard $h(t) \geq 0$ gives rise to a hazard distribution supported on $[0, 1]$ with no atom at 1 as long as

$$\int_0^1 h(t)\, dt = \infty.$$

As in the main construction, the event-time laws in the results below are absolutely continuous. Beyond the support cutoff, the hazard is set to zero as in Section 3.3.

E.1.1 Rates from independent event times

Lemma E.1.1. Let $T_1, T_2, \ldots, T_n \sim H$ be independent samples from the hazard distribution H, with cumulative distribution function $\Xi_H(t)$ and hazard $h(t)$. Define

$$X_t := \sum_{i=1}^n I_{\{T_i \leq t\}}.$$

Conditioned on $X_t = x_t$, there are exactly $n - x_t$ remaining events. Denote $R := n - x_t$. It holds that

$$\mathbb{P}[X_{t+\delta} = x_t + 1 | X_t = x_t] = Rh(t)\delta + o(\delta).$$

Proof. Let $T_t^+ := \min\{T_i : T_i > t\}$ denote the arrival time of the next jump at time t. Condition on the specific at risk index set $A = \{i : 1 \le i \le n,\ T_i > t\}$, corresponding to $|A| = R$ remaining events. Then for any $i \in A$ we have

$$\begin{aligned} p_\delta := \mathbb{P}(t < T_i \le t + \delta \mid T_i > t) &= 1 - \frac{S_H(t+\delta)}{S_H(t)} \\ &= 1 - \frac{S_H(t) + \delta S_H'(t) + o(\delta)}{S_H(t)} = h(t)\delta + o(\delta), \end{aligned}$$

noting that $S_H'(t) = -h(t)S_H(t)$. By independence, the number of jumps in $(t, t+\delta]$ among the R at-risk units is $\mathrm{Binomial}(R, p_\delta)$, and thus

$$\mathbb{P}(X_{t+\delta} = x_t + 1 \mid X_t = x_t) = Rp_\delta(1 - p_\delta)^{R-1}.$$

Since $p_\delta = h(t)\delta + o(\delta) = O(\delta)$

$$(1 - p_\delta)^{R-1} = 1 - (R-1)p_\delta + O(p_\delta^2) = 1 + o(1),$$

it follows

$$\mathbb{P}(X_{t+\delta} = x_t + 1 \mid X_t = x_t) = R(h(t)\delta + o(\delta))(1 + o(1)) = Rh(t)\delta + o(\delta).$$

□

Corollary E.1.2. Fix $z_c \in \mathbb{N}_{\ge 0}$ and define $X_t = \sum_{i=1}^{z_c} I_{\{T_i \le t\}}$. Then it holds that X_t evolves as a CTMC with rate matrix

$$Q_t^{z_c}(x_t + 1 | x_t) = (z_c - x_t)h(t)I_{\{x_t < z_c\}},$$

and $Q_t^{z_c}(x_t | x_t) = -Q_t^{z_c}(x_t + 1 | x_t)$, with $Q_t^{z_c}(\cdot | x_t) = 0$ otherwise. In other words, if the event times are distributed according to H, then counting the events that have occurred up until the current time out of z_c total events corresponds to the counting flow process conditioned on z_c, starting at zero.

E.1.2 Interarrival Times

Theorem E.1.3. Fix $n \ge 1$, and let $T_1, \ldots, T_n$ be independent samples from an absolutely continuous event-time law H, with CDF Ξ_H and survival function $S_H = 1 - \Xi_H$. Set

$$X_t = \sum_{i=1}^{n} I_{\{T_i \le t\}}.$$

Fix $t \in [0, 1)$ with $S_H(t) > 0$, and condition on $X_t = x_t$, where $\mathbb{P}(X_t = x_t) > 0$ and $R := n - x_t \ge 1$. Let V_t have the conditional law of the next event time $\min\{T_i : T_i > t\}$, and let $W_t = V_t - t$. For $0 < u < 1$, the quantile function of V_t is

$$\Xi_{V_t}^{-1}(u) = \Xi_H^{-1}\left(1 - S_H(t)(1-u)^{1/R}\right),$$

where $\Xi_H^{-1}(u) := \inf\{y : \Xi_H(y) \ge u\}$. Consequently, for $U \sim \mathrm{Unif}(0, 1)$,

$$W_t \overset{d}{=} \Xi_H^{-1}\left(1 - S_H(t)(1-U)^{1/R}\right) - t.$$

If $R = 0$, there is no further event, and the next event time is taken to be $+\infty$.

Proof. Condition first on the set of indices whose event times exceed t. The R remaining times are independent with the truncated law H conditioned on $(t, 1]$, whose CDF, for $y \ge t$, is

$$\Xi_{H,t}(y) = \frac{\Xi_H(y) - \Xi_H(t)}{1 - \Xi_H(t)} = 1 - \frac{S_H(y)}{S_H(t)}.$$

This law does not depend on which indices remain, so the same next event time distribution holds conditional on $X_t = x_t$. By independence,

$$\mathbb{P}(V_t > y) = \left(\frac{S_H(y)}{S_H(t)}\right)^R, \qquad y \ge t.$$

For $0 < u < 1$, it follows that

$$\Xi_{V_t}(y) \geq u \quad \Longleftrightarrow \quad \Xi_H(y) \geq 1 - S_H(t)(1-u)^{1/R}.$$

Taking the infimum over y gives the quantile formula, including when the CDF has flat intervals. Inverse-transform sampling gives $V_t \overset{d}{=} \Xi_{V_t}^{-1}(U)$, and subtracting t gives the formula for W_t. □

E.2 Application to split and deletion branches

We construct Branching Flows with split and deletion events. The split hazard h_{split} is defined from H_{split} in the same way as h above. For an internal branch leading to node v, the number of remaining split events is $w(v) - 1$, where $w(v)$ counts the leaves in the subtree starting at v. Its split rate is therefore $(w(v)-1)h_{\text{split}}(t)$, and its next split time is obtained from Theorem E.1.3, with the current branch's starting time in place of t and $R = w(v) - 1$.

Let H_{del} be a hazard distribution supported on $[0, 1]$ with no atom at 1, with associated hazard

$$h_{\text{del}}(t) = \frac{\xi_{\text{del}}(t)}{1 - \Xi_{\text{del}}(t)}.$$

A branch marked for deletion has one remaining deletion event, with rate $h_{\text{del}}(t)$. Branches not marked for deletion have deletion rate zero.

E.3 Conditional-state sampling

Fix the conditioning variable $Z = z = (x_1^{+\emptyset}, x_0, \mathcal{T}, \mathcal{A})$ and an observation time $t \in [0, 1)$. We wish to sample the augmented conditional state $\tilde{X}_t = (X_t, G_t) \mid Z = z$. An element inherits its value from its ancestors at their split times, so its state at t is obtained by following the forest from its root to the branch occupied at t. We sample the split and deletion events using the results from E.1. Between splits and deletions, along each branch, the base process is used to update the element values.

The required base transition is

$$p\Big(x_{t+\Delta}^{(k)} \mid x_t^{(k)}, a_k\Big), \qquad 0 \leq t \leq t + \Delta < 1,$$

over a branch segment with no split or deletion in its interior. Here k identifies that branch on this interval, and conditioning on a_k is interpreted as conditioning on $x_1^{(k)} = a_k$ were it to exist without splitting and deletion until $t = 1$. The index k is local to an uninterrupted branch segment. We write $\text{bridge}(u; s, y, a)$ for a draw from this transition with starting time s, current value y, anchor a, and end time u. The continuous and discrete base transitions used in our experiments are discussed in Appendix D.

Let $r_1, \ldots, r_{L_0}$ be the roots in initial-sequence order, with root r_j initialized at $x_0^{(j)}$. For a node v, retain the main text's anchor $a(v)$, descendant-leaf count $w(v)$, and deletion indicator $\delta(v)$. Let $g(v)$ denote its branch label. We refer to Appendix F for the tree-and-child-address encoding.

E.4 Conditional sampling, training, and marginal sampling algorithms

E.4.1 Conditional sampling given the forest and anchors

For event type b, write $S_b = 1 - \Xi_b$ and define

$$\text{next}_b(s, R) := \Xi_b^{-1}\Big(1 - S_b(s)(1-U)^{1/R}\Big), \qquad U \sim \text{Unif}(0, 1),$$

with an independent uniform draw on each call, for $R > 0$ and $S_b(s) > 0$. By Theorem E.1.3, this is the earliest of R independent event times with law H_b conditioned on exceeding s. It uses one uniform draw regardless of R.

Algorithm E.1 describes how to sample from from the conditional process efficiently, given Z. We present this for clarity but note that, in our practical implementation, we interleave conditional sampling with tree and anchor construction.

Algorithm E.1 Conditional state and training targets given Z

Require: Fixed $Z = z = (x_1^{+\emptyset}, x_0, \mathcal{T}, \mathcal{A})$, time $t \in [0, 1)$, admissible event laws
Ensure: (X_t, G_t) and per-element conditional targets

Initialize $\mathcal{B} \leftarrow [(r_j, 0, x_0^{(j)})]_{j=1}^{L_0}$, in root order, and $\mathcal{O} \leftarrow [\,]$.
while $\mathcal{B}$ is nonempty **do**
  Remove the first (v, s, y) from $\mathcal{B}$; set $T_v \leftarrow +\infty$.
  if $w(v) > 1$ **then**
    $T_v \leftarrow \mathrm{next}_{\mathrm{split}}(s, w(v) - 1)$.
  else if $\delta(v) = 1$ **then**
    $T_v \leftarrow \mathrm{next}_{\mathrm{del}}(s, 1)$.
  end if
  if $T_v \le t$ **then**
    if $w(v) > 1$ **then**
      $y' \leftarrow \mathrm{bridge}(T_v; s, y, a(v))$.
      Prepend $[(v^{\uparrow}, T_v, y'), (v^{\downarrow}, T_v, y')]$, in this order, to $\mathcal{B}$.
    end if ▷ A deleted leaf emits no element.
  else
    Append $(\mathrm{bridge}(t; s, y, a(v)), v)$ to $\mathcal{O}$.
  end if
end while
Write $\mathcal{O}[i] = (x_t^{(i)}, v_i)$, and set $L_t \leftarrow |\mathcal{O}|$.
$X_t \leftarrow (x_t^{(i)})_{i=1}^{L_t}, \quad G_t \leftarrow (g(v_i))_{i=1}^{L_t}$.
return (X_t, G_t) and the targets $(a_i, F_i^\star, R_i^\star, \rho_i^\star)_{i=1}^{L_t}$ defined below.

This visits the roots and their descendants in sequence order, so the output is already ordered. Each split samples one parent value, which both children inherit before evolving with their own anchors. For each output record, set

$$a_i = a(v_i), \qquad R_i^\star = w(v_i) - 1, \qquad \rho_i^\star = \delta(v_i), \qquad F_i^\star = \mathrm{target}^{\mathrm{base}}(t, x_t^{(i)}, a_i).$$

Here $\mathrm{target}^{\mathrm{base}}$ returns the training target of the chosen base generator; for endpoint prediction, it returns the anchor in the representation used by the base loss. Thus $F^\star = F_t^{z,G_t,\mathrm{base}}(X_t)$, $R^\star = R_t^{z,G_t}(X_t)$, and $\rho^\star = \rho_t^{z,G_t}(X_t)$. Retaining the node identifiers makes G_t explicit. Each node is visited at most once, requiring $\mathcal{O}(L_1^{+\emptyset})$ event-time and base-transition draws.

E.4.2 Training from conditional samples

To form a training example, draw $X_1 \sim p_{\mathrm{data}}$ and $Z \sim p_{Z|X_1}(\cdot \mid X_1)$ using a coupling from Appendix C. Independently draw t from the training-time law on $[0, 1)$, and apply Algorithm E.1 to this fixed pair (Z, t). Only (t, X_t) is supplied to the model; the forest and G_t determine the targets. Take a stochastic gradient step on the conditional loss in Definition 3.2: the base-process loss against $F^\star$, the Poisson loss against $R^\star$, and binary cross-entropy against $\rho^\star$. The elementwise split and deletion losses are summed, as in the main-text objective, optionally balanced by weighting across loss components and across time [Billera et al., 2026].

E.4.3 Numerical marginal sampling

At inference, draw the complete initial sequence from p_{simple} and use the learned marginal parameters. Each step first advances the base process without changing the sequence length, then independently samples a Poisson split count and a deletion flag for each element. For each element, we sample whether it splits, is deleted, or remains unchanged. A split produces two copies, a deletion removes the element, and otherwise the element is retained. In the event that both a split and deletion are sampled, the deletion takes precedence.

Algorithm E.2 Numerical sampling from the learned marginal process

Require: Joint sequence prior p_{simple}, grid $0 = t_0 < \cdots < t_K = 1$, learned generator parameters
Ensure: A generated sequence, possibly empty

Draw $X_{t_0} \sim p_{\text{simple}}$.
for $k = 0, \ldots, K-1$ **do**
  $t \leftarrow t_k, \quad \Delta \leftarrow t_{k+1} - t_k, \quad L \leftarrow |X_t|$.
  if $L = 0$ **then**
    return $[\,]$.
  end if
  Evaluate $F_t^{\theta,\text{base}}(X_t)$, $R_t^\theta(X_t)$, and $\rho_t^\theta(X_t)$.
  $(y_1, \ldots, y_L) \leftarrow \text{step}^{\text{base}}\Big(t, \Delta; X_t, F_t^{\theta,\text{base}}(X_t)\Big)$.
  Initialize $X' \leftarrow [\,]$.
  for $i = 1, \ldots, L$, in order **do**
    Draw $n_i \sim \text{Poisson}\big(\Delta h_{\text{split}}(t) R_{t,i}^\theta(X_t)\big)$.
    Independently draw $d_i \sim \text{Bernoulli}\big(\rho_{t,i}^\theta(X_t)\,[1 - \exp\{-\Delta h_{\text{del}}(t)\}]\big)$.
    if $d_i = 0$ **then**
      Append $1 + n_i$ copies of y_i to X'.
    end if
  end for
  $X_{t_{k+1}} \leftarrow X'$.
end for
return X_{t_K}.

Model predictions and hazard multipliers are evaluated at the start of each grid interval, and new copies are first processed on the next step. This is a finite-step approximation. An empty sample is allowed and terminates sampling. The sampler can optionally suppress empty samples if they are disallowed in the conditional process, which is appropriate in many settings.

If each original element were restricted to at most one split per step, one could instead use competing stay/split/delete events. With $\lambda_{\text{split}} = h_{\text{split}}(t) R_{t,i}^\theta(X_t)$, $\lambda_{\text{del}} = h_{\text{del}}(t) \rho_{t,i}^\theta(X_t)$, and $\lambda = \lambda_{\text{split}} + \lambda_{\text{del}}$, this gives

$$\begin{aligned} \mathbb{P}(E = \text{stay}) &= e^{-\lambda\Delta}, \\ \mathbb{P}(E = b) &= \frac{\lambda_b}{\lambda}\left(1 - e^{-\lambda\Delta}\right), \qquad b \in \{\text{split}, \text{del}\}, \end{aligned} \tag{5}$$

when $\lambda > 0$; for $\lambda = 0$, take $E = \text{stay}$. This alternative first-order approximate sampler avoids the arbitrary precedence of deletions, but we prefer Poisson split counts because multiple splits per element within a step can occur in practice during very early sampling, particularly when starting from a single element.

F Branch tracking, splits, and deletions

This section gives a concrete construction of the auxiliary branch process G_t. Together with Z, it identifies each element's current branch and hence its anchor and remaining split and deletion targets. Augmenting X_t with G_t makes the conditional process Markov.

We identify a branch with the node 'ahead' of it (in the root-to-leaf, $t = 0$ to $t = 1$ direction), and $g_t^{(i)} = (\tau_t^{(i)}, b_t^{(i)})$ stores a tree index $\tau_t^{(i)}$, and a branch indicator vector $b_t^{(i)}$ comprising a possibly-empty ordered sequence of child indices $\uparrow$ (first child) or $\downarrow$ (second child) which, since the tree is binary, read from root towards leaves identifies a single branch on the tree by identifying which of the two children are descended at each bifurcation. An element of the augmented state reads $\tilde{x}_t^{(i)} = (x_t^{(i)}, g_t^{(i)}) = (x_t^{(i)}, (\tau_t^{(i)}, b_t^{(i)}))$. With $<>$ the empty branch indicator tuple (using non-standard tuple brackets to make the tuple nesting easier to read), $(\tau_t^{(i)} = k, b_t^{(i)} = <>)$ denotes the root of the k^{th} tree, and $(\tau_t^{(i)} = k, b_t^{(i)} = <\uparrow\uparrow\downarrow>)$ would denote the second child of the first child of the first child of the root of the k^{th} tree.

Let $\tilde{x}_t^{(i)} = (x_t^{(i)}, (\tau_t^{(i)}, b_t^{(i)}))$. We use $\triangleright$ to denote the append operation

$$\tilde{x}_t^{(i)} \triangleright \uparrow := (x_t^{(i)}, (\tau_t^{(i)}, b_t^{(i)} \triangleright \uparrow)),$$

and analogously define $\tilde{x}_t^{(i)} \triangleright \downarrow$.

For a state $\tilde{x}_t = (\tilde{x}_t^{(1)}, \ldots, \tilde{x}_t^{(n)})$, we define a 'split operator':

$$\mathrm{split}_i(\tilde{x}_t) := (\tilde{x}_t^{(1)}, \ldots, (\tilde{x}_t^{(i)} \triangleright \uparrow),\ (\tilde{x}_t^{(i)} \triangleright \downarrow), \ldots, \tilde{x}_t^{(n)}).$$

This duplicates, in place, the element $\tilde{x}_t^{(i)}$, sending a copy down each of the two child branches. We also define a 'deletion' operator:

$$\mathrm{del}_i(\tilde{x}_t) := (\tilde{x}_t^{(1)}, \ldots, \tilde{x}_t^{(i-1)}, \tilde{x}_t^{(i+1)}, \ldots, \tilde{x}_t^{(n)})$$

that deletes the element $\tilde{x}_t^{(i)}$ at index i.

G Branching Flows in Generator Matching

G.1 The Infinitesimal Generator of Jump Processes

In line with Generator Matching (GM) [Holderrieth et al., 2024] and Flow Matching Guide (FMG) [Lipman et al., 2024], we model jumps by a time-dependent kernel $Q_t(dy; x)$ that, for each state $x \in S$, assigns a positive measure on $S \setminus \{x\}$. Its total mass

$$\lambda_t(x) = \int Q_t(dy; x)$$

gives the jump intensity or jump rate, i.e. the instantaneous hazard of leaving x, and we require $\lambda_t(x) < \infty$. When $\lambda_t(x) > 0$, normalizing Q_t yields the jump destination distribution

$$J_t(dy; x) = \frac{Q_t(dy; x)}{\lambda_t(x)},$$

which is a probability measure on $S \setminus \{x\}$. Now we see how specifying a jump kernel $Q_t(dy; x)$ describes a jump process. With the hazard/rate of $\lambda_t(x)$, we will jump to a point $y \sim J_t(dy; x)$.

For $\lambda_t(x_t) = 0$ the process is constant, and otherwise the sampling procedure is as follows:

$$(X_{t+\Delta t} | X_t = x_t) = \begin{cases} X_t & \text{with probability } 1 - \Delta t \lambda_t(x_t) + o(\Delta t), \\ \sim J_t(dy; x_t) & \text{with probability } \Delta t \lambda_t(x_t) + o(\Delta t). \end{cases}.$$

The infinitesimal generator $\mathcal{L}_t$ associated to a process with jump measure Q_t becomes

$$\mathcal{L}_t f(x) = \int (f(y) - f(x)) Q_t(dy; x).$$

G.2 The Marginal Branching Infinitesimal Generator

For the conditional construction in Section 3.3, the split and deletion targets are $R_{t,i}^{z,g}(x) = w(v_i) - 1$ and $\rho_{t,i}^{z,g}(x) = \delta(v_i)$. Their instantaneous event rates are obtained by multiplying these targets by the corresponding hazards. Theorem 3.1 and Definition 3.3 give the marginal parameters as posterior averages of these conditional targets. The linear parametrization and losses below therefore apply directly to remaining split counts and deletion indicators.

Let $S := \bigsqcup_{n \geq 1} \mathcal{E}^n$. We say that $x \in \mathcal{E}^n$ has n elements. We let $h_{\mathrm{split}}(t)$ and $h_{\mathrm{del}}(t)$ be overall hazard multipliers as in Appendix E, corresponding to an overall rate multiplier for insertion increments and an overall rate multiplier for deletions. Suppose that insertion to the $i+1$-st element out of n elements occurs with rate $R_{t,i}(x) h_{\mathrm{split}}(t)$ and that deletion of the i'th element out of n elements occurs with rate $\rho_{t,i}(x) h_{\mathrm{del}}(t)$. Here, $R_{t,i}(x) \geq 0$ is intended to indicate the 'remaining number of insertions of the i'th element' and $\rho_{t,i}(x) \in [0, 1]$ is intended to indicate the 'probability of deleting the i'th element'. For $n = 1$, we enforce a deletion probability is zero as to not result in an empty sequence.

We may now derive a time- and state varying linear parametrization for the infinitesimal generator of our branching process. The notion of a linear parametrization of a generator was introduced in Holderrieth et al. [2024] and further discussed in Lipman et al. [2024], was clarified to be able to vary with both time and state in Billera et al. [2026].

In what follows, we consider $x \in \mathcal{E}^n$ having $n \geq 1$ elements. A time- and state varying linear parametrization of the branching generator is

$$\mathcal{L}_t^{\mathrm{branch}} f(x) = \langle \mathcal{K}_{t,n}^{\mathrm{split}} f(x), R_t(x) \rangle_{t,n}^{\mathrm{split}} + \langle \mathcal{K}_{t,n}^{\mathrm{del}} f(x), \rho_t(x) \rangle_{t,n}^{\mathrm{del}} + \langle \mathcal{K}_{t,n}^{\mathrm{base}} f(x), F_t^{\mathrm{base}}(x) \rangle_{t,x}^{\mathrm{base}}.$$

In the above, using a time-dependent jump kernel as in [Billera et al., 2026], splits and deletions correspond to the time-dependent jump kernel

$$Q_t(dy; x) = \sum_{i=1}^{n} h_{\text{split}}(t) R_{t,i}(x) \mu_{t,i}(dy|x) + \sum_{i=1}^{n} h_{\text{del}}(t) \rho_{t,i}(x) \delta_{\text{del}_i(x)}(dy).$$

We may now write down the generator for our process. First, we introduce $\text{split}_i(x, u)$ as a function taking the current state x as well as some auxiliary insertion randomness u as inputs and outputting a post-split state. Denote by $\nu_{t,i}(du|x)$ the possibly time and state dependent insertion kernel and let $\mu_{t,i}(dy|x) := \text{split}_i(x, \cdot)\#\nu_{t,i}(\cdot|x)$ denote the push-forward of the auxiliary randomness through the split operator. All empirical results in this manuscript use deterministic splitting, corresponding to

$$\mu_{t,i}(dy|x) = \delta_{(x^{(1)},\ldots,x^{(i)},x^{(i)},x^{(i+1)},\ldots,x^{(n)})}(dy).$$

The infinitesimal generator for jumps becomes

$$\begin{aligned}
\mathcal{L}_t^{\text{jump}} f(x) &= \int_S (f(y) - f(x)) Q_t(dy; x) \\
&= \int_S \left((f(y) - f(x)) h_{\text{split}}(t) \cdot \sum_{i=1}^{n} R_{t,i}(x) \mu_{t,i}(dy|x) \right) \\
&\qquad + \int_S \left((f(y) - f(x)) h_{\text{del}}(t) \cdot \sum_{i=1}^{n} \rho_{t,i}(x) \delta_{\text{del}_i(x)}(dy) \right) \\
&= h_{\text{split}}(t) \cdot \sum_{i=1}^{n} \Big(R_{t,i}(x) \int (f(\text{split}_i(x, u)) - f(x)) \nu_{t,i}(du|x) \Big) \\
&\qquad + h_{\text{del}}(t) \cdot \sum_{i=1}^{n} \Big((f(\text{del}_i(x)) - f(x))\, \rho_{t,i}(x) \Big).
\end{aligned}$$

To express this generator, we define the time-varying weighted bilinear forms

$$\langle v, w \rangle_{t,n}^{\text{split}} := h_{\text{split}}(t) \langle v, w \rangle_{\mathbb{R}^n} \qquad \text{and} \qquad \langle v, w \rangle_{t,n}^{\text{del}} := h_{\text{del}}(t) \langle v, w \rangle_{\mathbb{R}^n}.$$

Each form is an inner product when its corresponding hazard is positive, and is identically zero when that hazard vanishes. The jump generator has the representation

$$\mathcal{L}_t^{\text{jump}} f(x) = \langle \mathcal{K}_{t,n}^{\text{split}} f(x), R_t(x) \rangle_{t,n}^{\text{split}} + \langle \mathcal{K}_{t,n}^{\text{del}} f(x), \rho_t(x) \rangle_{t,n}^{\text{del}},$$

where in the above, we set:

$$\mathcal{K}_{t,n}^{\text{split}} f(x) = \left(\int (f(\text{split}_i(x, u)) - f(x)) \nu_{t,i}(du|x) \right)_{i=1}^{n}$$

and

$$\mathcal{K}_{t,n}^{\text{del}} f(x) = (f(\text{del}_i(x)) - f(x))_{i=1}^{n}.$$

For all t, an equivalent Euclidean linear parametrization places the hazards in the structural operators:

$$\begin{aligned}
\mathcal{L}_t^{\text{jump}} f(x) = &\langle h_{\text{split}}(t) \mathcal{K}_{t,n}^{\text{split}} f(x), R_t(x) \rangle_{\mathbb{R}^n} \\
&+ \langle h_{\text{del}}(t) \mathcal{K}_{t,n}^{\text{del}} f(x), \rho_t(x) \rangle_{\mathbb{R}^n}.
\end{aligned}$$

This leaves the targets $R_t(x)$ and $\rho_t(x)$ unchanged.

In between jumps, we assume that the process evolves according to a base generator on each branch level — this could, for example, correspond to an SDE. We assume that we have a linear parametrization

$$\mathcal{L}_t^{\text{base}} f(x) = \langle \mathcal{K}_{t,n}^{\text{base}} f(x), F_t^{\text{base}}(x) \rangle_{t,x}^{\text{base}}$$

for the base process along each branch. This results in

$$\begin{aligned}\mathcal{L}_t^{\text{branch}} f(x) &= \mathcal{L}_t^{\text{jump}} f(x) + \mathcal{L}_t^{\text{base}} f(x) \\ &= \langle \mathcal{K}_{t,n}^{\text{split}} f(x), R_t(x) \rangle_{t,n}^{\text{split}} + \langle \mathcal{K}_{t,n}^{\text{del}} f(x), \rho_t(x) \rangle_{t,n}^{\text{del}} + \langle \mathcal{K}_{t,n}^{\text{base}} f(x), F_t^{\text{base}}(x) \rangle_{t,x}^{\text{base}}.\end{aligned}$$

G.3 The Branching Flows Loss

The conditional branching flows loss is the auxiliary-process-conditioned generator matching loss associated to the linear parametrization from Appendix G.2, with conditional dynamics as described in Section 3.3. For $x \in \bigsqcup_{n\geq 1} \mathcal{E}^n$, these paths specify a conditional infinitesimal generator of the form

$$\begin{aligned}\mathcal{L}_t^{z,g,\text{branch}} f(x) = \langle \mathcal{K}_{t,n}^{\text{split}} f(x), R_t^{z,g}(x) \rangle_{t,n}^{\text{split}} + \langle \mathcal{K}_{t,n}^{\text{del}} f(x), \rho_t^{z,g}(x) \rangle_{t,n}^{\text{del}} \\ + \langle \mathcal{K}_{t,n}^{\text{base}} f(x), F_t^{z,g,\text{base}}(x) \rangle_{t,x}^{\text{base}}.\end{aligned}$$

The structural operators and hazards are fixed, while the model outputs marginal predictions R_t, ρ_t and F_t^{base}. The model parametrized infinitesimal generator is therefore

$$\mathcal{L}_t^{\theta,\text{branch}} f(x) = \langle \mathcal{K}_{t,n}^{\text{split}} f(x), R_t^{\theta}(x) \rangle_{t,n}^{\text{split}} + \langle \mathcal{K}_{t,n}^{\text{del}} f(x), \rho_t^{\theta}(x) \rangle_{t,n}^{\text{del}} + \langle \mathcal{K}_{t,n}^{\text{base}} f(x), F_t^{\theta,\text{base}}(x) \rangle_{t,x}^{\text{base}}.$$

As in [Billera et al., 2026], when a generator is expressed as a sum of linear parametrizations, a valid per-term conditional generator matching loss is then the sum of Bregman divergences

$$D_{t,x}^{\text{split}}(R_t^{z,g}(x), R_t^{\theta}(x)) + D_{t,x}^{\text{del}}(\rho_t^{z,g}(x), \rho_t^{\theta}(x)) + D_{t,x}^{\text{base}}(F_t^{z,g,\text{base}}(x), F_t^{\theta,\text{base}}(x)),$$

again noting that x stands for $x \in S$. Assuming that $R_t^{z,g}(x)$ can be interpreted as the remaining number of splits ahead of each element and that $\rho_t^{z,g}(x)$ can be interpreted as deletion probabilities, we choose a Poisson-like Bregman for splits and a cross-entropy loss for deletions.

Note that these loss functions require a notion of Bregman divergence associated with convex functions that are only differentiable on a subset of their domain, as considered in [Billera et al., 2026]

If the base Bregman divergence also decomposes across elements, the conditional branching flows loss of Definition 3.2 is obtained by summing the three per-element losses over the elements of X_t and taking the conditional-path expectation. The corresponding marginal loss is given in Definition 3.3.

G.4 Proofs

Assumption G.1. For both the joint conditional process $(X_t, G_t)|Z = z$ and joint marginal process (X_t, G_t), we assume the regularity conditions in Generator Matching [Holderrieth et al., 2024].

Assumption G.2. We also assume the regularity conditions in Generator Matching for the marginal generator $\mathcal{L}_t$ and its corresponding Markov process X_t, as in Theorem 3.1.

G.4.1 Proof of Theorem 3.1

Definition G.4.1. Fix z and consider the conditional law $\mathbb{P}^z := \mathbb{P}(\cdot \mid Z = z)$. Denote the expectation under this law by $\mathbb{E}^z$. Let $\mathcal{G}$ be a finite discrete space and let (X_t, G_t) under this law be an $\mathcal{X} \times \mathcal{G}$ valued Markov process with infinitesimal generator $W_t^z : \mathcal{T}(\mathcal{X} \times \mathcal{G}) \to \mathcal{C}(\mathcal{X} \times \mathcal{G}; \mathbb{R})$. Write $p_t^z(dx, dg)$ for its joint law, $p_t^z(dx)$ for the X_t marginal, and $p_{G_t|X_t}^z(dg \mid x)$ for the branch posterior all conditioned on our fixed z. For test functions $f \in \mathcal{T}(\mathcal{X})$, let

$$\tilde{f} : \mathcal{X} \times \mathcal{G} \to \mathbb{R}, \qquad \tilde{f}(x, g) = f(x)$$

denote the *lift* of f to the domain $\mathcal{X} \times \mathcal{G}$. Then, for fixed $g \in \mathcal{G}$, define the *conditional infinitesimal generator*

$$\begin{aligned}\mathcal{L}_t^{z,g} f(x) &:= W_t^z(\tilde{f})(x, g) \\ &= \frac{d}{dh}\Big|_{h=0^+} \mathbb{E}^z[f(X_{t+h}) \mid X_t = x,\, G_t = g].\end{aligned}$$

Lemma G.4.2. Under the fixed-z law of Definition G.4.1, define the measure $\mu_{X_t}^{z,g}(A) := p_t^z(A \times \{g\})$ for measurable sets A. Then it holds that

$$\partial_t \sum_{g \in \mathcal{G}} \int_{\mathcal{X}} f(x) \mu_{X_t}^{z,g}(dx) = \sum_{g \in \mathcal{G}} \int_{\mathcal{X}} \mathcal{L}_t^{z,g} f(x) \mu_{X_t}^{z,g}(dx).$$

Proof. First, note that since $\mathcal{G}$ is discrete, each $E \subset \mathcal{G}$ is measurable, and

$$p_t^z(dx, E) = \sum_{g \in E} p_t^z(dx, \{g\}) = \sum_{g \in E} \mu_{X_t}^{z,g}(dx).$$

Now, write

$$\partial_t \sum_{g \in \mathcal{G}} \int_{\mathcal{X}} f(x) \mu_{X_t}^{z,g}(dx) = \partial_t \int_{\mathcal{X}} f(x) \sum_{g \in \mathcal{G}} \mu_{X_t}^{z,g}(dx)$$

where in the above we are integrating with respect to the measure $\sum_{g \in \mathcal{G}} \mu_{X_t}^{z,g}(dx)$. This becomes

$$\begin{aligned}
&= \partial_t \int_{\mathcal{X}} f(x) p_t^z(dx, \mathcal{G}) \\
&= \partial_t \int_{\mathcal{X}} f(x) \int_{\mathcal{G}} p_t^z(dx, dg) \\
&= \partial_t \int_{\mathcal{X} \times \mathcal{G}} f(x) p_t^z(dx, dg) \\
&= \partial_t \int_{\mathcal{X} \times \mathcal{G}} \tilde{f}(x, g) p_t^z(dx, dg)
\end{aligned}$$

and since W_t^z generates $p_t^z(dx, dg)$, by the Kolmogorov Forward Equation, it holds:

$$\begin{aligned}
&= \int_{\mathcal{X} \times \mathcal{G}} W_t^z \tilde{f}(x, g) p_t^z(dx, dg) \\
&= \sum_{g \in \mathcal{G}} \int_{\mathcal{X}} \mathcal{L}_t^{z,g} f(x) \mu_{X_t}^{z,g}(dx).
\end{aligned}$$

□

Theorem G.4.3. Under the same fixed-z law, let

$$\mathcal{L}_t^z f(x) := \sum_{g \in \mathcal{G}} \mathcal{L}_t^{z,g} f(x) p_{G_t | X_t}^z(g \mid x).$$

Then, it holds that $\mathcal{L}_t^z$ generates $p_t^z(dx)$, that is $\mathcal{L}_t^z$ solves the Kolmogorov Forward Equation for p_t^z:

$$\partial_t \int_{\mathcal{X}} f(x) p_t^z(dx) = \int_{\mathcal{X}} \mathcal{L}_t^z f(x) p_t^z(dx).$$

Moreover, suppose that, for each t, x, the parameters $F_t^{z,g}(x)$ lie in a common domain $\Omega_{t,x} \subset V_{t,x}$ and give the linear parametrization

$$\mathcal{L}_t^{z,g} f(x) = \langle \mathcal{K}_{t,x} f, F_t^{z,g}(x) \rangle_{t,x}$$

for each $g \in \mathcal{G}$. Here $\mathcal{K}_{t,x}$ maps test functions to $V_{t,x}$, and the operator and inner product $\langle \cdot, \cdot \rangle_{t,x}$ are common to all g. Then

$$F_t^z(x) = \mathbb{E}_{G_t \sim p_{G_t | X_t}^z(dg|x)}[F_t^{z,G_t}(x)]$$

linearly parametrizes $\mathcal{L}_t^z$.

Proof. For each $t \in [0, 1]$, we have $p_t^z(dx) = \sum_{g \in \mathcal{G}} \mu_{X_t}^{z,g}(dx)$, and hence:

$$\begin{aligned}
\partial_t \int_{\mathcal{X}} f(x) p_t^z(dx) &= \partial_t \int_{\mathcal{X}} f(x) \sum_{g \in \mathcal{G}} \mu_{X_t}^{z,g}(dx) \\
&= \partial_t \sum_{g \in \mathcal{G}} \int_{\mathcal{X}} f(x) \mu_{X_t}^{z,g}(dx)
\end{aligned}$$

and by Lemma G.4.2, it holds:

$$
\begin{aligned}
&= \sum_{g \in \mathcal{G}} \int_{\mathcal{X}} \mathcal{L}_t^{z,g} f(x) \mu_{X_t}^{z,g}(dx) \\
&= \sum_{g \in \mathcal{G}} \int_{\mathcal{X}} \mathcal{L}_t^{z,g} f(x) p^z_{G_t|X_t}(g \mid x) p_t^z(dx) \\
&= \int_{\mathcal{X}} \underbrace{\left(\sum_{g \in \mathcal{G}} \mathcal{L}_t^{z,g} f(x) p^z_{G_t|X_t}(g \mid x) \right)}_{= \mathcal{L}_t^z f(x)} p_t^z(dx).
\end{aligned}
$$

For fixed t, x, linearity of expectation and of the map $w \mapsto \langle v, w \rangle_{t,x}$ for fixed $v \in V_{t,x}$ gives

$$
\begin{aligned}
\mathcal{L}_t^z f(x) &= \mathbb{E}_{G_t \sim p^z_{G_t|X_t}(dg|x)} [\mathcal{L}_t^{z,G_t} f(x)] \\
&= \mathbb{E}_{G_t \sim p^z_{G_t|X_t}(dg|x)} [\langle \mathcal{K}_{t,x} f, F_t^{z,G_t}(x) \rangle_{t,x}] \\
&= \langle \mathcal{K}_{t,x} f, \mathbb{E}_{G_t \sim p^z_{G_t|X_t}(dg|x)} [F_t^{z,G_t}(x)] \rangle_{t,x} \\
&= \langle \mathcal{K}_{t,x} f, F_t^z(x) \rangle_{t,x}.
\end{aligned}
$$

□

We now return to the full law, with $Z \sim p_Z$ and p_t denoting the marginal law of X_t after averaging over Z, to prove Theorem 3.1, which we re-state for clarity.

Theorem 3.1. Let $\mathcal{L}_t^{z,g,\text{branch}}$ be linearly parametrized by $F_t^{z,g,\text{branch}}$ (cf. equation 1, Appendix G.2) . Then, the marginal generator

$$
\mathcal{L}_t^{\text{branch}} f(x) := \mathbb{E}_{(G_t, Z) \sim p(\cdot|x)} [\mathcal{L}_t^{Z,G_t,\text{branch}}(f)(x)]
$$

generates a stochastic process X_t with marginals p_t and is linearly parametrized by

$$
F_t(x) = \mathbb{E}[F_t^{Z,G_t,\text{branch}}(x) \mid X_t = x].
$$

Proof. First, apply Theorem G.4.3 under each fixed-z law to the conditional generator $\mathcal{L}_t^{z,g,\text{branch}}$. Marginalising over G_t gives the z-conditioned generator $\mathcal{L}_t^{z,\text{branch}}$. Averaging these generators and their parameters over $Z \mid X_t = x$ gives the joint posterior expectation over $(Z, G_t) \mid X_t = x$ by the tower property. Proposition 1 of Generator Matching [Holderrieth et al., 2024], extended to the time- and state-varying linear parametrization setting of Billera et al. [2026], then gives the stated marginal generator. □

Remark G.3. *Here we are of course using that the endpoints in Z are chosen such that $\pi_{1\#} p_Z = p_{\text{data}}$ with π_1 denoting the "endpoint map", that is the map that recovers the endpoint from Z. This is identical to standard Flow matching, we make this remark since in Branching Flows, Z does not merely carry endpoint information but also additional latent information relating to the branching process.*

G.4.2 Proof of Theorem 3.4

Lemma G.4.4. Fix z and use the conditional laws and parameters from Theorem G.4.3. Let $D_{t,x}$ denote the Bregman divergence for this parametrization. The auxiliary-process-conditioned generator matching loss and the loss after marginalising G_t coincide up to a constant in θ:

$$
\begin{aligned}
&\nabla_\theta \mathbb{E}_{t \sim \mathcal{D},\, (X_t, G_t) \sim p_t^z(dx, dg)} D_{t,X_t}(F_t^{z,G_t}(X_t), F_t^\theta(X_t)) \\
&\quad = \nabla_\theta \mathbb{E}_{t \sim \mathcal{D},\, X_t \sim p_t^z(dx)} D_{t,X_t}(F_t^z(X_t), F_t^\theta(X_t)).
\end{aligned}
$$

Proof. Since expectations commute with the left slot of Bregman divergences under the gradient (cf. Lipman et al. [2024], Billera et al. [2026]), we obtain:

$$
\begin{aligned}
&\nabla_\theta \mathbb{E}_{t\sim\mathcal{D},\,(X_t,G_t)\sim p_t^z(dx,dg)} D_{t,X_t}(F_t^{z,G_t}(X_t), F_t^\theta(X_t)) \\
&= \nabla_\theta \mathbb{E}_{t\sim\mathcal{D},\,X_t\sim p_t^z(dx)} \mathbb{E}_{G_t\sim p^z_{G_t|X_t}(dg|X_t)} D_{t,X_t}(F_t^{z,G_t}(X_t), F_t^\theta(X_t)) \\
&= \mathbb{E}_{t\sim\mathcal{D},\,X_t\sim p_t^z(dx)} \nabla_\theta \mathbb{E}_{G_t\sim p^z_{G_t|X_t}(dg|X_t)} D_{t,X_t}(F_t^{z,G_t}(X_t), F_t^\theta(X_t)) \\
&= \mathbb{E}_{t\sim\mathcal{D},\,X_t\sim p_t^z(dx)} \nabla_\theta D_{t,X_t}\Big(\mathbb{E}_{G_t\sim p^z_{G_t|X_t}(dg|X_t)}[F_t^{z,G_t}(X_t)], F_t^\theta(X_t)\Big) \\
&= \mathbb{E}_{t\sim\mathcal{D},\,X_t\sim p_t^z(dx)} \nabla_\theta D_{t,X_t}(F_t^z(X_t), F_t^\theta(X_t)) \\
&= \nabla_\theta \mathbb{E}_{t\sim\mathcal{D},\,X_t\sim p_t^z(dx)} D_{t,X_t}(F_t^z(X_t), F_t^\theta(X_t)).
\end{aligned}
$$

□

Theorem 3.4 Let L_{BF}, L_{CBF} be as in Definition 3.2 and 3.3. Then, we have

$$\nabla_\theta L_{CBF}(\theta) = \nabla_\theta L_{BF}(\theta).$$

Proof. Apply Lemma G.4.4 for each fixed z and average the gradient identity over $Z \sim p_Z$. Then apply Generator Matching's [Holderrieth et al., 2024] Proposition 2, extended to the setting with time- and state-varying linear parametrizations in Billera et al. [2026], to average the z-conditioned parameters over $Z \mid X_t$. This gives the stated equality of gradients for the full conditional and marginal losses in Definitions 3.2 and 3.3. □

H Supplementary Empirical Details

H.1 QM9 analysis

H.1.1 Data

We use QM9 data (license: CC0) [Ramakrishnan, 2024], where the atom order in the coordinate files is arranged to match the canonical SMILES string ordering, where hydrogens are after the heavy atoms. We reasoned that the splitting mechanism of Branching Flows would not excel if there are spatially distant elements that are neighbors in the primary sequence, since a late split would then require very large flow velocities. We thus kept the primary heavy atom ordering (since neighboring SMILES atoms are often bonded, and thus spatially close), but redistributed the hydrogens such that each hydrogen is inserted into the sequence in front of its nearest heavy atom, ordered by distance when there are more than one hydrogen neighbour per heavy atom.

H.1.2 Branching Flows specification

For QM9, we constructed our 'default' Branching Flow with the following choices:

- Conditional base process:
  - Continuous: OU process flow (Appendix D.1), with a mean-reversion rate $\theta = 5$, and where the variance decays from 10 when $t = 0$ to 0.001 by $t = 1$.
  - Discrete: DFM convex interpolation (appendix D.2) of x_0, uniform noise, and x_1, with CDFs Ξ_1 and Ξ_2 of $\mathcal{D}_1 = \text{Beta}(2, 2)$ and $\mathcal{D}_2 = \text{Beta}(2, 2)$, respectively, and $\omega_u = 0.2$.
- Split-time distribution: $\text{Beta}(1, 3/2)$

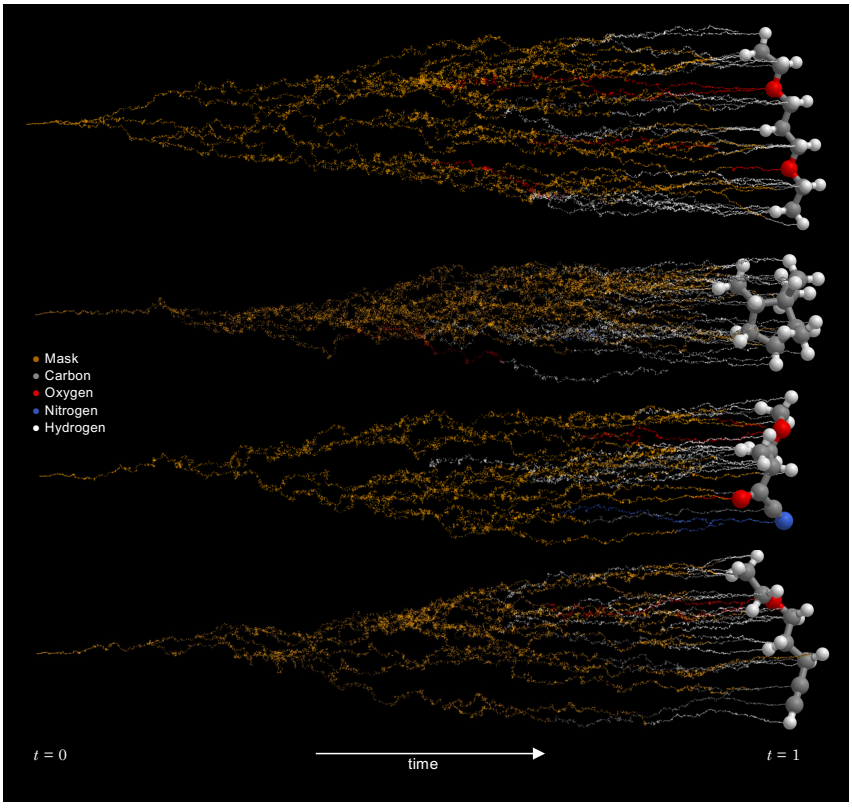


Figure 7: **QM9 Branching Flows sampling trajectories.** A visualization of inference trajectories from a QM9-trained Branching Flows model. The final sampled molecule, when $t = 1$, is depicted on the right. At every inference step from $t = 0$ to $t = 1$, we draw the current x_t state as colored points, with a small rightward displacement, which shows the branching and deletion history as x_t is transported from a single element when $t = 0$ to the final sampled molecule. The color of the trails shows the atom type which begins as 'Mask' (in orange) and switches to concrete atoms as $t \to 1$.

- Deletion-time distribution: $U(0, 1)$
- To-be-deleted policy: Poisson with mean 20% of the x_1 length elements are sampled (with replacement) and duplicated, and the duplicated elements are inserted either side of the original.
- x_0 state distribution: $N(0, 1)$ for each continuous coordinate, and mask/dummy token for the discrete atom label.
- x_0 length distribution: 1, always.
- Anchors: Weighted average for continuous states, and masked token for discrete states.

H.1.3 Model and Training

We use a 12 layer transformer model (12 heads, each with head dimension of 64), with an embedding dimension of 384. The atom positions are encoded by Random Fourier Features (RFFs), and pairwise spatial features are also provided to each layer by a learnable, layer-specific, head-specific attention bias. Rotary Positional Encoding (RoPE, Su et al. [2023]) is used to encode the primary sequence ordering. The final six layers additively update the atom positions (and the spatial pair features are recalculated after each position update). The model outputs the endpoint-predicted (i.e. by $t = 1$) locations, logits over the endpoint-predicted discrete state, the log of the per-element expected number of future bifurcations by $t = 1$, and a deletion binary logit. The model was trained, using Muon [Jordan et al., 2024] with post-warmup learning rate of 0.005, for 500k batches (where each batch was 128 randomly-sampled QM9 molecules) and a linear learning-rate cooldown for the last 50k batches. For sampling, 10k samples from the Branching Flows QM9 model were generated with a time step schedule $t = 1 - (\cos(\pi s) + 1)/2$ for $s \in (0, 1/1000, 2/1000, \ldots, 1)$.

We compare generated samples to the data distributions of atom counts (total, carbon, nitrogen, oxygen, fluorine, and hydrogen), and molecular properties. For molecule properties, since the training samples and model-generated samples are atom point clouds (ie. no bond structure is explicitly represented), we use OpenBabel to convert .xyz coordinates to .sdf, which also infers bond structure and SMILES strings. The .sdf is then read into RDKit for downstream analysis, and computing molecule properties, and fingerprints. OpenBabel was run using the OpenBabel.jl Julia wrapper, and RDKit was run using the MoleculeFlow.jl Julia wrapper.

Molecular fingerprints encode the structural features of a molecule into a fixed-length bit vector which can be used for similarity search, clustering, etc. Here, we compute the 'RDKit fingerprints' (using MoleculeFlow.jl) and embed these in two dimensions, using Uniform Manifold Approximation (run via UMAP.jl, with default settings). This is used to visualize the distribution of true QM9 and generated samples.

We compute a multivariate Jensen–Shannon divergence (JSDm) similar to [Schneuing et al., 2025] between the generated and reference QM9 distributions using the filtered subset of molecules only. The calculation uses the joint distribution of 10 properties: molecular weight, logP, hydrogen-bond donors, hydrogen-bond acceptors, the numbers of C/H/O/N/F atoms, and the total atom count. For each property, bin boundaries are defined by reference-data quantiles (4 bins per property), and both datasets are then mapped into the resulting 10-dimensional joint histogram. JSDm is the Jensen–Shannon divergence between these two joint histograms, reported in bits; lower values indicate better agreement with the QM9 reference distribution.

Following ADiT [Joshi et al., 2025] we also include Posebusters [Buttenschoen et al., 2024] metrics, and Validity and Uniqueness. For each sample that can be converted to SDF, we run PoseBusters and record the individual pass/fail outcomes of the standard molecular geometry and chemistry checks. Validity is the fraction of all generated samples whose reconstructed molecule yields a valid SMILES string under the analysis pipeline. This is also normalized by the full sample count. Uniqueness is the fraction of distinct SMILES strings among the SMILES-valid samples only.

To quantify distribution overlap, we use a Kolmogorov-Smirnov-like statistic between the empirical cumulative distribution functions of samples Y from each model against QM9 data. We use $1 - KS_D$, where

$$KS_D = \max_x |\hat{F}_Y(x) - \hat{F}_{QM9}(x)|$$

where $\hat{F}$ denotes the empirical CDF. This statistic is 1 when the distributions agree perfectly, and 0 when they do not overlap at all.

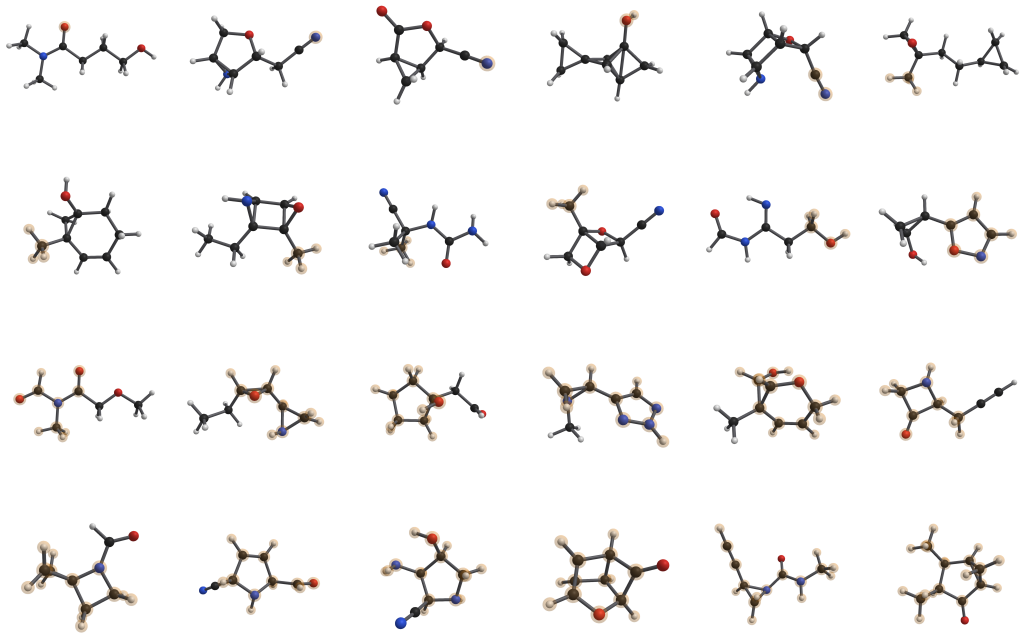


Figure 8: **Examples of 'Completion' task samples.** Spheres indicate designable atoms.

For Bernoulli metrics the tables report symmetric two-sigma Wald intervals:

$$\hat{p} \pm 2\sqrt{\frac{\hat{p}(1-\hat{p})}{N}}.$$

We compare to ADiT [Joshi et al., 2025], which is a strong, recent, oracle-length molecule sampler, using 10,000 samples obtainable directly from their code page. As a variable-length baseline, we additionally use the transdimensional-jump diffusion model from Campbell et al. [2023] to generate 10,000 samples, and analyze them as above.

QM9 Completion task: The QM9 completion experiments use the same basic generative setup as the unconditional baseline, but with an additional binary conditioning mask indicating which atoms are fixed and which atoms are designable. During training, fixed atoms are provided as context, while the model is trained to generate only the designable portion of the molecule. The conditioning masks are derived from a precomputed library of chemically sensible partial structures for each QM9 molecule. To build this library, each QM9 molecule is first reordered so that hydrogens are placed adjacent to their nearest heavy atoms, yielding contiguous heavy-atom blocks. Candidate masked regions are then defined as contiguous runs of these blocks whose heavy atoms form a connected subgraph and attach to the rest of the molecule through exactly one bond. This construction biases the candidates toward substituent-like pendant fragments rather than arbitrary atom subsets.

At training time, one first samples a molecule that has at least one valid candidate region. Then one candidate region is selected uniformly from that molecule's candidate list. Finally, one of two conditioning polarities is chosen with equal probability: either the candidate interval is fixed and the rest of the molecule is designable, or the complement is fixed and the candidate interval is designable. As a result, the model is trained on both fragment-completion and fragment- replacement style conditioning patterns. Each sampled training example therefore contains both fixed and designable atoms, and the mask is incorporated directly into the model input and loss masking.

As for the unconditional task, the metrics for the completion task evaluate the full molecule distribution (i.e. not just the completed part), which should be learned if the completion task is correctly learned.

H.2 Branching Flows QM9 variations and ablations

For the results in figures 4 the Branching Flows and architecture matched comparison models were trained for 800k iterations for the unconditional task and 400k iterations for the conditional tasks. For the full set of ablations and variations (described below) the models were trained for 400k iterations.

Here we describe all model variants. Unless a change is stated explicitly in a variation subsection below, the setting is identical to the Default model and process described above:

`Default.` As described above.

`Multimodal FM.` This is the fixed-length comparison. No splits or deletions, but the process, training, and architecture otherwise matches the 'Default' BF model. This model always knows the 'correct' molecule length.

`Brownian Motion.` The Ornstein–Uhlenbeck process is replaced by Brownian motion (variance = 1).

`Deletions: 2x.` The deletion padding is increased from 20% to 100% (ie. on average the X_1 length is doubled), so the bridge is trained with substantially more to-be-deleted leaves.

`Deletions: None.` Deletion is disabled by setting `deletion_pad`$= 0$ and removing the deletion head from both training and sampling. This learns length variation via splits only.

`Deletions only.` This ablation removes splits entirely, and begins with an X_0 length distribution that is Uniform between 30 and 50 (i.e. guaranteed to always be longer than the X_1 length, which is required for distribution learning with only deletions). We note that, since this model is computing over sequences that are typically longer the standard Branching Flows models, the training time (quadratic in length for transformers) is slower.

`X0 length: 1+Poisson(17).` The initial length prior is changed to $1 + \mathrm{Poisson}(17)$, no longer starting from a single element (and approximately matching the target distribution length mean).

`Trees: Proximity-Weighted.` The Default variant randomly merges adjacent elements when constructing trees. The Proximity Weighted variant still uses adjacent pairs, but instead of sampling them uniformly it computes a distance between the extracted node features and chooses the adjacent pair with minimum distance, with random tie-breaking inside a small tolerance band. Merges are biased toward geometrically nearby adjacent atoms. When merging internal nodes it uses the anchors, which are weighted averages of child nodes.

`Trees: Rich-get-Richer.` This still considers only eligible adjacent pairs, but it samples pair $(i, i+1)$ with probability proportional to

$$(w_i + w_{i+1})^\alpha,$$

where w_i and w_{i+1} are the descendant counts of the two candidate subtrees. With $\alpha = 1$, larger adjacent clusters are therefore more likely to merge first.

`Anchors: Random Child.` Instead of a weighted average for continuous states, and a mask token for discrete states, for continuous states this chooses one of the two child anchors with probability proportional to the child descendant counts and then copies that child exactly for continuous components instead of averaging the two children. Discrete anchors remain masked.

`Anchors: Discrete random Child.` Continuous anchors use the canonical weighted average, but discrete anchors copy one child atom-type state with probability proportional to child weight instead of being replaced by the masked token.

`Branching Hazard: Uniform.` The branch-time distribution is changed from $\mathrm{Beta}(1, 3/2)$ to $\mathrm{Uniform}(0, 1)$.

`Ordering: Hierarchical.` To test whether the canonical SMILES ordering is critical, the default order is removed. For each sampled molecule, the atoms are first randomly permuted, then clustered by Euclidean average-linkage hierarchical clustering, and finally linearized by recursively swapping the left and right children of each internal node at random. This yields a sequence order that respects the hierarchical geometric clustering while still changing stochastically each time the molecule is used. The Proximity Weighted tree policy is used.

`Ordering: random.` The default order is removed. Instead, every time a molecule is sampled for training, the atom order is replaced by a fresh random permutation. The Proximity Weighted tree policy is used.

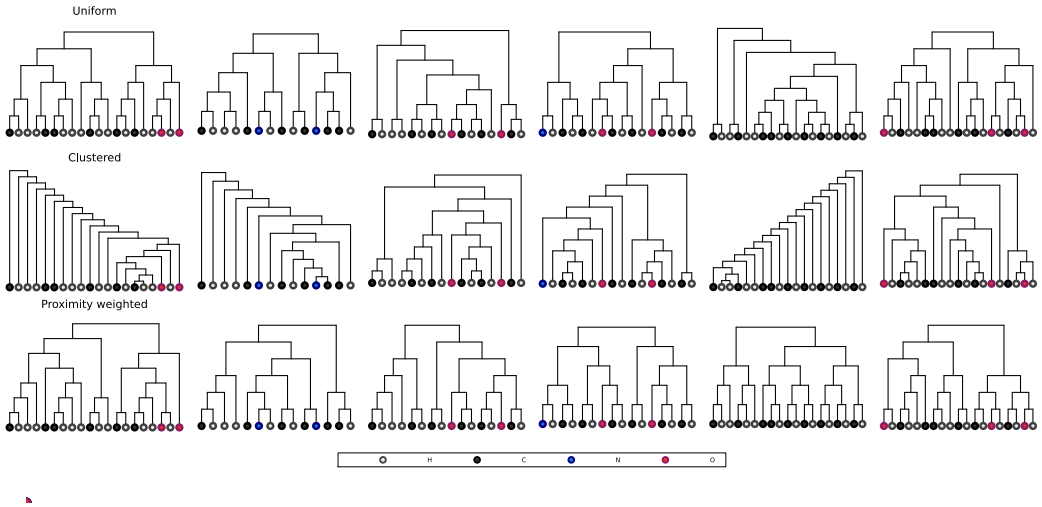


Figure 9: **Trees policites.** Shown are the default 'uniform adjacent' tree sampling, vs the Rich-get-richer (here labelled as 'Clustered') and Proximity Weighted strategies, described in SI H.2.

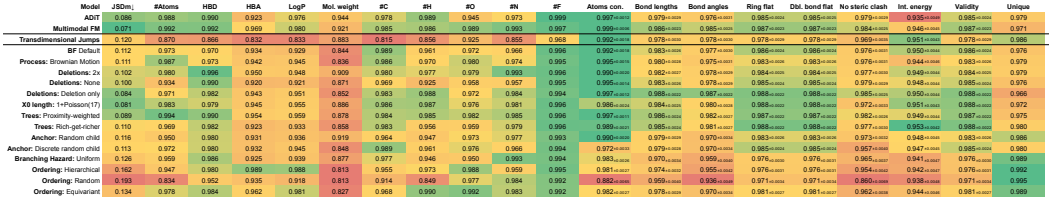
Model	Atoms con.	Bond lengths	Bond angles	Ring flat	Dbl. bond flat	No steric clash	Int. energy
ADiT	0.997±0.0012	0.979±0.0029	0.976±0.0031	0.985±0.0024	0.985±0.0025	0.979±0.0029	0.935±0.0049
Multimodal FM*	0.999±0.0006	0.988±0.0022	0.987±0.0022	0.989±0.0021	0.989±0.0021	0.987±0.0022	0.944±0.0046
Transdimensional Jumps	0.992±0.0018	0.978±0.0030	0.978±0.0030	0.978±0.0029	0.978±0.0029	0.969±0.0035	0.951±0.0043
BF Default	0.997±0.0011	0.986±0.0024	0.983±0.0026	0.987±0.0023	0.987±0.0023	0.983±0.0026	0.946±0.0045
BF Permutation Equivariant	0.995±0.0014	0.984±0.0025	0.980±0.0028	0.985±0.0024	0.985±0.0024	0.980±0.0028	0.945±0.0045
BF Proximity Weighted Tree	0.997±0.0010	0.989±0.0021	0.988±0.0022	0.990±0.0020	0.990±0.0020	0.986±0.0023	0.947±0.0045
BF X0 1+Poisson(17)	0.994±0.0015	0.989±0.0021	0.988±0.0022	0.989±0.0020	0.989±0.0021	0.983±0.0026	0.946±0.0045
MFM Length-conditional	0.983±0.0026	0.982±0.0027	0.978±0.0029	0.984±0.0025	0.983±0.0026	0.946±0.0045	0.965±0.0037
MFM Oracle length	0.999±0.0005	0.986±0.0023	0.982±0.0026	0.987±0.0023	0.987±0.0023	0.945±0.0046	0.985±0.0024
BF Default	0.994±0.0015	0.980±0.0028	0.971±0.0034	0.982±0.0027	0.981±0.0027	0.944±0.0046	0.975±0.0031

Figure 10: **Posebusters metrics** from the models in Figure 4. Uncertainty is shown with symmetric two-sigma Wald intervals.

`Ordering: Permutation Equivariant.` This variation removes explicit sequence-order dependence from both the neural network and the tree policy. RoPE is removed from the transformer. The tree construction policy is examines all eligible intragroup pairs (i, j) rather than only adjacent pairs and selects the pair with minimum Euclidean distance between the extracted node coordinates, again with random tie-breaking among near-ties.

H.2.1 QM9 compute resources

Each model training run of 400k iterations takes approximately one day on an RTX 6000 Ada, and two days for the 800k training runs.



Model	JSD$_{m_1}$	#Atoms	HBD	HBA	LogP	Mol. weight	#C	#H	#O	#N	#F	Atoms con.	Bond lengths	Bond angles	Ring flat	Dbl. bond flat	No steric clash	Int. energy	Validity	Unique
ADIT	0.086	0.988	0.996	0.923	0.976	0.944	0.978	0.989	0.945	0.973	0.999	0.997±0.0002	0.979±0.0029	0.976±0.0031	0.985±0.0024	0.985±0.0026	0.979±0.0029	0.935±0.0065	0.985±0.0024	0.979
Multimodal FM	0.071	0.992	0.992	0.969	0.980	0.921	0.985	0.986	0.989	0.993	0.997	0.999±0.0006	0.986±0.0025	0.985±0.0025	0.987±0.0023	0.987±0.0025	0.984±0.0025	0.946±0.0049	0.987±0.0023	0.971
Transdimensional Jumps	0.120	0.870	0.866	0.832	0.833	0.883	0.815	0.856	0.925	0.855	0.968	0.992±0.0014	0.978±0.0030	0.978±0.0030	0.978±0.0028	0.978±0.0029	0.969±0.0035	0.951±0.0042	0.978±0.0028	0.986
BF Default	0.112	0.973	0.970	0.934	0.929	0.844	0.989	0.961	0.972	0.966	0.996	0.992±0.0018	0.983±0.0026	0.977±0.0030	0.986±0.0024	0.986±0.0024	0.976±0.0031	0.950±0.0044	0.986±0.0024	0.976
Process: Brownian Motion	0.111	0.987	0.973	0.942	0.945	0.836	0.986	0.970	0.980	0.974	0.995	0.995±0.0015	0.980±0.0028	0.975±0.0031	0.983±0.0026	0.983±0.0026	0.976±0.0031	0.944±0.0046	0.983±0.0026	0.979
Deletions: 2x	0.102	0.980	0.996	0.950	0.948	0.909	0.980	0.977	0.979	0.993	0.996	0.990±0.0020	0.982±0.0027	0.978±0.0029	0.984±0.0025	0.984±0.0026	0.977±0.0029	0.940±0.0046	0.984±0.0026	0.979
Deletions: None	0.100	0.934	0.990	0.920	0.921	0.871	0.969	0.925	0.958	0.957	0.995	0.995±0.0014	0.983±0.0026	0.978±0.0029	0.985±0.0024	0.985±0.0024	0.979±0.0029	0.948±0.0044	0.985±0.0024	0.976
Deletions: Deletion only	0.084	0.971	0.982	0.943	0.951	0.852	0.983	0.988	0.972	0.984	0.994	0.997±0.0012	0.988±0.0022	0.987±0.0022	0.988±0.0022	0.988±0.0022	0.985±0.0026	0.950±0.0044	0.988±0.0022	0.966
X0 length: 1+Poisson(17)	0.081	0.983	0.979	0.945	0.955	0.886	0.986	0.987	0.976	0.981	0.996	0.996±0.0014	0.984±0.0025	0.980±0.0028	0.988±0.0022	0.988±0.0022	0.972±0.0033	0.951±0.0043	0.988±0.0022	0.972
Trees: Proximity-weighted	0.089	0.994	0.990	0.954	0.959	0.878	0.984	0.985	0.982	0.985	0.996	0.997±0.0011	0.986±0.0024	0.982±0.0027	0.987±0.0022	0.987±0.0022	0.982±0.0030	0.949±0.0044	0.987±0.0022	0.975
Trees: Rich-get-richer	0.110	0.969	0.982	0.923	0.933	0.858	0.983	0.956	0.959	0.979	0.996	0.989±0.0017	0.985±0.0024	0.981±0.0027	0.988±0.0022	0.988±0.0022	0.977±0.0028	0.953±0.0042	0.988±0.0022	0.980
Anchor: Random child	0.116	0.950	0.980	0.931	0.936	0.919	0.964	0.947	0.973	0.977	0.993	0.990±0.0018	0.979±0.0029	0.970±0.0034	0.983±0.0026	0.983±0.0026	0.973±0.0032	0.948±0.0045	0.983±0.0026	0.986
Anchor: Discrete random child	0.113	0.972	0.980	0.932	0.945	0.848	0.989	0.961	0.976	0.966	0.994	0.972±0.0033	0.979±0.0029	0.970±0.0034	0.985±0.0024	0.985±0.0024	0.957±0.0040	0.947±0.0045	0.985±0.0024	0.980
Branching Hazard: Uniform	0.126	0.959	0.986	0.925	0.939	0.877	0.977	0.946	0.950	0.993	0.994	0.983±0.0026	0.970±0.0034	0.959±0.0040	0.976±0.0030	0.976±0.0031	0.965±0.0037	0.941±0.0047	0.976±0.0030	0.989
Ordering: Hierarchical	0.162	0.947	0.980	0.989	0.988	0.813	0.955	0.973	0.988	0.959	0.995	0.981±0.0027	0.974±0.0032	0.955±0.0042	0.976±0.0031	0.976±0.0031	0.954±0.0042	0.942±0.0047	0.976±0.0031	0.992
Ordering: Random	0.193	0.834	0.952	0.935	0.918	0.813	0.914	0.849	0.977	0.984	0.992	0.882±0.0065	0.959±0.0039	0.936±0.0048	0.971±0.0034	0.971±0.0034	0.860±0.0069	0.938±0.0048	0.971±0.0034	0.995
Ordering: Equivariant	0.134	0.978	0.984	0.962	0.981	0.827	0.968	0.990	0.992	0.983	0.992	0.982±0.0027	0.978±0.0029	0.970±0.0034	0.981±0.0027	0.981±0.0027	0.962±0.0036	0.944±0.0046	0.981±0.0027	0.989

Figure 11: **QM9: Variations and Ablations.** Description of model variants is in H.2 and metrics are the same as from Figures 4 and 10.

H.3 Antibodies

H.3.1 Data

Antibody sequence models were trained on a filtered subset Turnbull et al. [2024] of the antibody heavy chains from the Observed Antibody Space database (OAS) [Kovaltsuk et al., 2018] (license: CC-BY-4.0). The filtering retained only human sequences and removed sequences with unknown residues, missing conserved cysteines and those which had framework region 1 or 4 shorter than allowed by IMGT definitions. Sequences were further filtered to remove redundancy by clustering at 95% identity. This resulted in a total of 118 million sequences, of which a random subset of 15 million were sampled and used for training. For the evaluation of the model, a similarly-filtered subset of the OAS data was used.

H.3.2 Branching Flows Specification

- Conditional base process:
  - Discrete: DFM convex interpolation (appendix D.2) of x_0, uniform noise, and x_1, with $\omega_u = 0.2$, and CDFs Ξ_1 and Ξ_2 of $\mathcal{D}_1 = \text{Beta}(1.5, 1.5)$, and $\mathcal{D}_2 = \text{Beta}(2, 2)$, respectively.
- Split-time distribution: $\text{Beta}(1, 2)$
- Deletion-time distribution: $U(0, 1)$
- To-be-deleted policy: deletion padding multiplier $d_r = 1.2$
- x_0 state distribution: masked tokens.
- x_0 length distribution: uniform between 110 and 140.

H.3.3 Model and Training

For the antibody sequence modeling task, we used a transformer-based architecture similar to that of a diffusion transformer [Peebles and Xie, 2023], but with additional split and deletion heads. Time was embedded with RFFs, and adaptive layer normalization (adaLN-Zero) was used for time conditioning. The model had an embedding dimension of 768, 12 attention heads, and 8 layers.

The models (Branching Flows, a matched Discrete Flow Matching [Gat et al., 2024] oracle-length baseline, and an appropriately-matched discrete-state variable length baseline using Edit Flows[Havasi et al., 2025]) were trained using a batch size of 64 resulting in a total of 234 375 training steps for the 15 million sequence dataset. Muon was used with a 2000 step linear warm-up followed by a cosine decay starting at 10^{-3} and decaying to 10^{-7}.

H.3.4 Antibody Sequence Generation: Metrics and Comparisons

To evaluate Branching Flows, 10 000 sequences were generated with 1 000 uniformly-spaced steps from Branching Flows, Oracle Length, and Edit Flows models and then compared to natural ones. Sequence diversity was assessed by computing the minimum pairwise cosine distance (over vectors of 3-mer counts) within each set. Sequence 'novelty' was evaluated by computing the minimum Levenshtein distance to any sequence in a hold-out validation data set (needed because otherwise the novelty of all natural sequences would be zero).

Sequence 'nativeness' was computed using AbNatiV Ramon et al. [2024]. Finally, the distribution of amino acids, distribution of sequence lengths and the distribution of CDR3 lengths were also computed. The latter was computed using ANARCI Dunbar and Deane [2016].

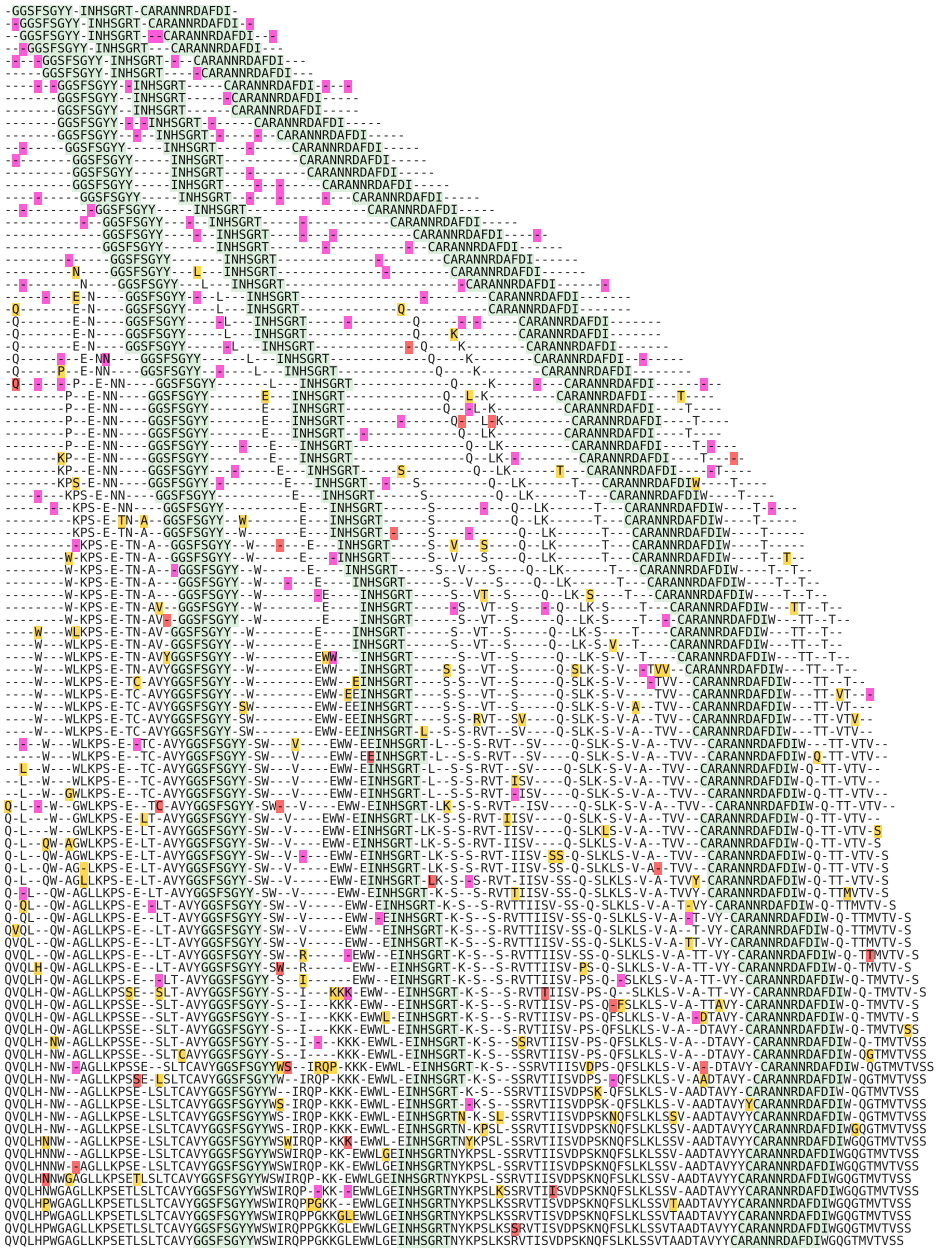


Figure 12: **Antibody segment sampling.** A sample from the Branching Flows model trained for the conditional antibody task, where multiple random-length segments were fixed, and the model generates the non-fixed segments. For this example, we fixed the complementarity determining regions and the model successfully samples four framework regions, including two outer flanking regions and two infix regions. Green shows fixed regions. Magenta shows insertions, yellow shows substitutions, and red shows elements that are deleted in the next step.

For investigating the training dynamics of Branching Flows compared to Oracle Length and Edit Flows models the losses cannot be directly compared, so after every 500 steps during training, 5 sequences were generated (with 200 uniformly-spaced steps) and saved. The perplexity of the generated sequences was computed as evaluated by a 130 million parameter autoregressive model trained on the data from Turnbull et al. [2024].

For the antibody conditional 'completion' task, for each heavy-chain sequence, a set of contiguous editable regions was sampled, while the remainder of the sequence was treated as fixed context,

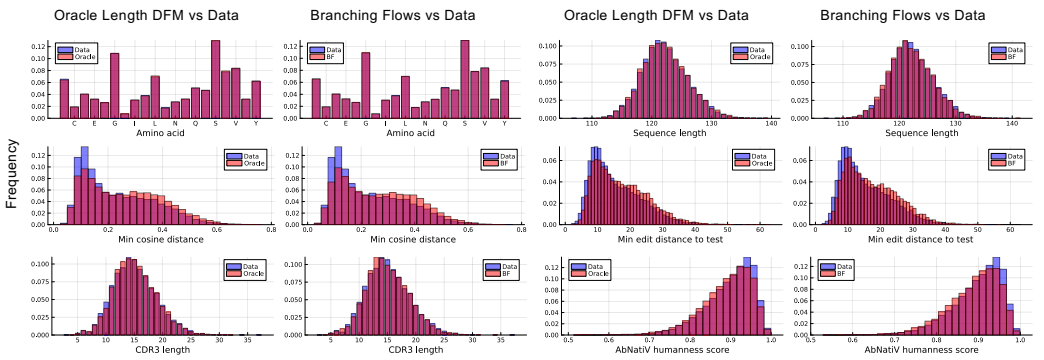


Figure 13: **Branching Flows vs Oracle Length.** Shown are property histograms for antibody sequence generation, for properties with KS agreement shown in table 1.

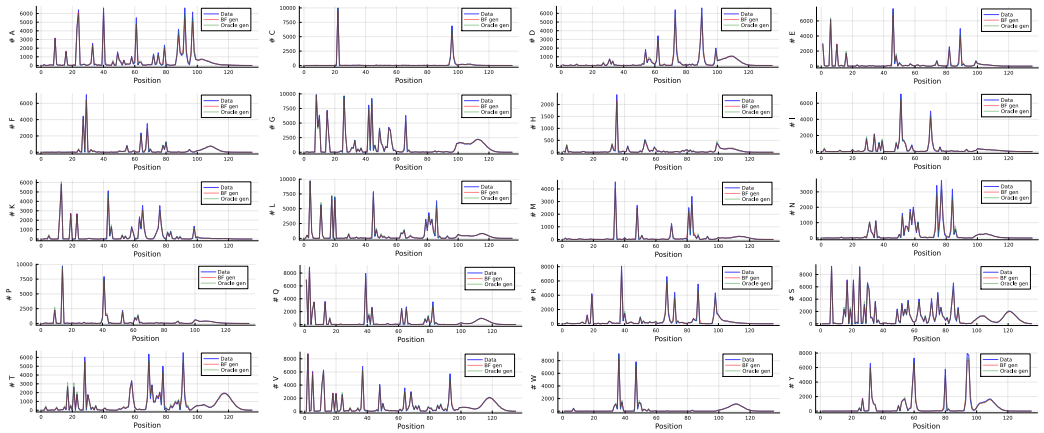


Figure 14: **Amino acid position distributions.** The count of the number of each amino acid at every position in natural data, and in samples from Branching Flows ('BF gen') and Oracle Length DFM ('Oracle gen').

meaning that insertions, deletions, and substitutions were disallowed outside the editable regions. The number of editable regions was sampled as $1 + \text{Poisson}(6 \cdot U(0,1))$ and the length of each proposed region was sampled as $\text{Poisson}(30 \cdot U(0,1))$. For each proposal, a starting position and a direction were sampled uniformly, and the final editable region was the union of all spans. Overlapping proposals therefore merged into contiguous editable regions.

For each contiguous editable region, in contrast to the unconditional Branching Flows setup, the completion task does not use a global x_0 length prior, but instead initialized each editable region from exactly one token (for both Branching Flows and Edit Flows). Editable regions were augmented with mean 30% 'to-be-deleted' states for Branching Flows.

The same 15 million antibody sequences were used as in the unconditional experiments. The model scale was preserved (hidden dimension 768, 12 attention heads, 8 layers), and the only difference in the models relative to the prior task was that the conditional models were conditioned on binary masks representing which positions were editable. Training hyperparameters were also matched to the unconditional setting: a batch size of 64 was used with a 2000-step linear warm-up to 10^{-3} and then a cosine decay to 10^{-7}. Finally, the models were evaluated by generating 10 000 sequences using the same conditional specification used during training. The fixed contexts were sampled from a held-out test set. Moreover, to quantify how well the sequences aligned with true sequences, the same distribution matching metrics as used for the unconditional task were computed using the entire sequences (not just the editable regions).

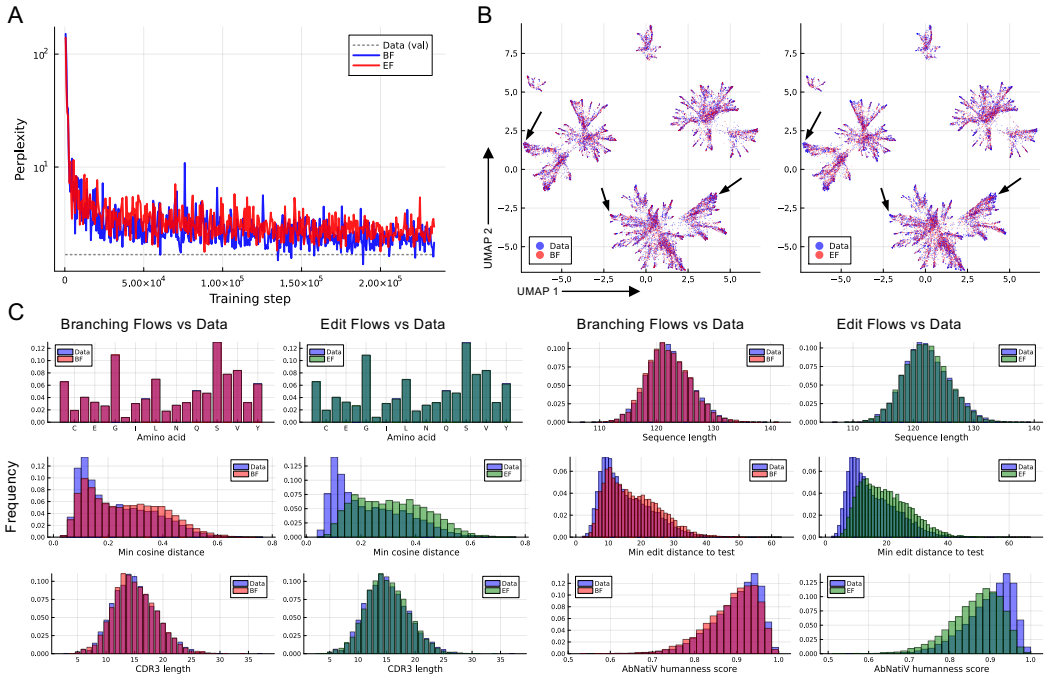


Figure 15: **Branching Flows vs Edit Flows. A** The perplexity, evaluated under an autoregressive LLM, of samples generated by Branching Flows (BF) vs Edit Flows (EF), which is a discrete-only variable length flow matching strategy. This is shown over training iterations. **B** two seqUMAP [Hanke et al., 2022] plots comparing Branching Flows and Edit Flows (respectively) each against real sequences. Arrows point to regions, at the extremities of the UMAP-embedded shapes, where Edit Flows density appears to be reduced. **C** Comparison of several distributions of the generated sequences with sequences from the validation data set.

H.3.5 Antibody compute resources

Each antibody model training run takes approximately 3.5 days on a single Nvidia A6000 GPU.

H.4 Proteins

To investigate whether Branching Flows would scale to a more complex learning problem where the state space is multimodal, we modified and finetuned a pre-trained ChainStorm model [Oresten et al., 2025] to allow the model to determine how long chains, or designable segments, should be.

Briefly, ChainStorm is a multimodal flow matching model that jointly generates protein backbones and amino acid sequence labels (but not side chain atom positions). Amino acid backbones are represented as sequences of 'frames' [Jumper et al., 2021] with Euclidean positions and SO(3) rotations, and ChainStorm uses an Invariant Point Attention [Jumper et al., 2021] transformer architecture to estimate the $t = 1$ frame positions and orientations, whose conditional/marginal paths are governed by Euclidean Brownian Motion and SO(3) rotational diffusion. Discrete Flow Matching [Gat et al., 2024] is used for amino acid sequence labels.

ChainStorm's Euclidean and SO(3) conditional processes were constructed with relatively low noise, but we reasoned that Branching Flows, especially with late-time branching events requiring higher flow velocities, might benefit from noisier processes which would have higher flow velocities generally, so we first finetuned the default ChainStorm model (weight license: MIT) with modified element-wise conditional processes, using

- for Euclidean positions, a time-inhomogeneous OU process (appendix D.1) with mean reversion rate of 100, and a variance of 150 at $t = 0$ that decays to 0 as $t \to 1$.
- for SO(3) rotations, a manifold-respecting version of the above OU process with identical parameters.
- for discrete states, a DFM convex interpolation (appendix D.2) with CDFs Ξ_1 and Ξ_2 of $\mathcal{D}_1 = \text{Beta}(3.0, 1.5)$ and $\mathcal{D}_2 = \text{Beta}(2, 2)$, respectively, and $\omega_u = 0.2$.

All Branching Flows protein models were then finetuned atop this model, which exhibited much higher variance sampling paths than the original ChainStorm.

We then modified the ChainStorm architecture to construct 'BF-ChainStorm' to allow for i) element branching and deletion, and ii) conditioning on fixed protein components (in general any set of residues, but here chains or contiguous segments of chains) that are then prevented from changing during training or inference. Conditioning is done by including a 'conditioning mask' embedding layer whose outputs get summed into the initial activations, and by preventing the per-layer frame updates for fixed residues. For element branching and deletions, we reasoned that, since we are finetuning into an existing model, we didn't want to only extract contributions from the final output layer (since this is used for rotations, positions, and discrete states), so we concatenate the output activations from the last three layers. This is summed into an embedding of t and that sum is passed into a SwiGLU-like layer that collapses to a single dimension. This is done separately for branching and deletion. The scalar outputs are transformed into the expected remaining split counts R_t^θ and deletion probabilities ρ_t^θ used in the losses. During inference, the corresponding event rates are $h_{\text{split}}(t)R_t^\theta$ and $h_{\text{del}}(t)\rho_t^\theta$. Finally, since the state dimension now changes during inference, we replaced ChainStorm's self-conditioning with recycling, which requires no architectural changes but instead requires additional model passes within each inference step.

One subtlety related to Branching Flows is that, originally, ChainStorm conditioned on residue indices which provide, during training, the model with information about when parts of backbone are missing (i.e. unresolved in the crystal or cryo-EM structures) since a gap in the residue numbering will be present. To generate backbones without 'chain breaks' (usually the desired behavior), at inference time the generation is conditioned on residue numbers with no gaps. However, since BF-ChainStorm's conditional/marginal paths change by insertions and deletions with t, including residue numbers no longer makes sense, as the number of residues for any intervening segment (which, in BF-ChainStorm, must be stochastically generated during inference) would be pre-determined by the residue numbering. So instead of residue indices, we use fixed-increment positional index. However, since the training data has many chain breaks, the model would then stochastically generate structures with chain breaks. So we additionally introduced a 'break mask' encoder (also summed into the starting activations) that encodes, on a per-chain basis, whether the 'designable' portion of the chain includes any chain breaks. When used during inference, this reduces the number of chain breaks but doesn't remove them entirely, possibly because the model is getting conflicting signals between the conditioner and the emerging structure (e.g. large unstructured loops are often not resolved in protein structures, so if the model is generating one that might overwhelm the chain-break conditioner).

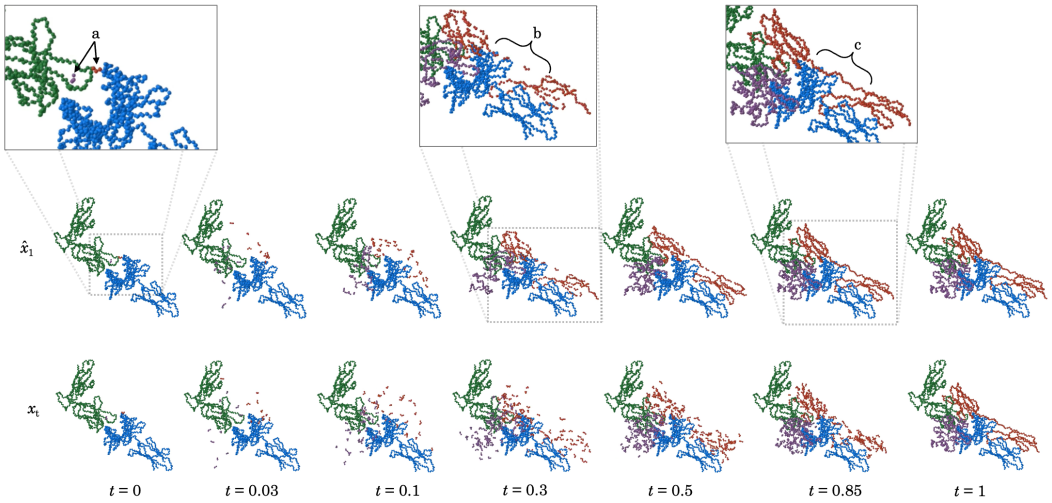


Figure 16: **Branching Flows protein sampling trajectories.** Shown are snapshots from a sampling trajectory from BF-ChainStorm, with both the current state x_t and the model's prediction of the end state, $\hat{x_1}$. The trajectories are colored by chain, and each backbone residue is shown as three spheres $(N, C\alpha, C)$. Here, two chains were fixed, and two chains were designable, with each designable chain starting from a single residue (arrow a in the first inset). The chains grow over time, and in this example the orange chain converges on two separate domains. Interestingly, during the intermediate flow state ($t = 0.3$, bracket b in the second inset), as the two domains resolve in $\hat{x_1}$, they are not connected to each other, and the model uses split events to build a complete linker between them by $t = 1$.

Chain breaks are easy to detect using bond distances, and we exclude them from refolding analysis because our folding pipeline requires intact chains.

When sampling trees $\mathcal{T}$ for protein models, when the group (described above) is the chain index, the requirement that separate chains never occur on the same tree allows the number of sampled chains in a structure to be specified at inference time (but each of their lengths stochastically sampled by the model). Similarly, any residues (even within a single chain) separated by residues fixed in the conditioning (relevant for BranchSegment) may not coalesce when sampling trees. Thus no single tree spans two designable segments.

For BranchChain, we initialize each chain to have $1 + \text{Poisson}(20)$ x_0 elements, each drawn from the ChainStorm base element distribution, and for BranchSegment each designable segment has a single x_0 element. For sampling 'to-be-deleted' elements, we use $d_r = 1.2$ for BranchChain and $d_r = 1.1$ for BranchSegment (reasoning that the more strongly informative conditioning information for BranchSegment might require less exploratory flow paths). Samples were generated with 400 steps: $t = 1 - (\cos(\pi\tau) + 1)/2$ for $\tau \in (0, 1/400, 2/400, ..., 1)$.

For illustration purposes, we also finetuned a BranchChain variant with only a single x_0 element per chain. Figure 16 shows steps along the sampling trajectory for a sample where two chains (blue and green) were fixed, and two chains were sampled. The orange chain, best seen through the lens of the model-predicted end state $\hat{x}_1$, highlights a key Branching Flows feature: at first, the designed chain converges upon two separate domains, with insufficient residues in between them for them to be linked. This is then progressively filled in by insertions, until a complete linker connects them as a single chain.

For BranchSegment we do not attempt quantitative evaluations, but we demonstrate that it is a promising solution to the 'unknown length infix sampling' problem. Taking a protein complex (PDB: 9IQP, Shcheblyakov et al. [2025]) of a nanobody and the SARS-CoV-2 receptor binding domain (RBD), we mask only the 'Complementarity Determining Region 3' (CDR3) as designable, and we generate 20 samples from the RBD-bound complex, and 20 samples from the unbound nanobody alone. The sampled CDR3s had a broad length distribution (7 to 29 residues for bound, 7 to 40 unbound). For the bound samples, even though the side chain atom positions from the RBD were not provided to BranchSegment (it only sees backbone frames and amino acid labels) the designed

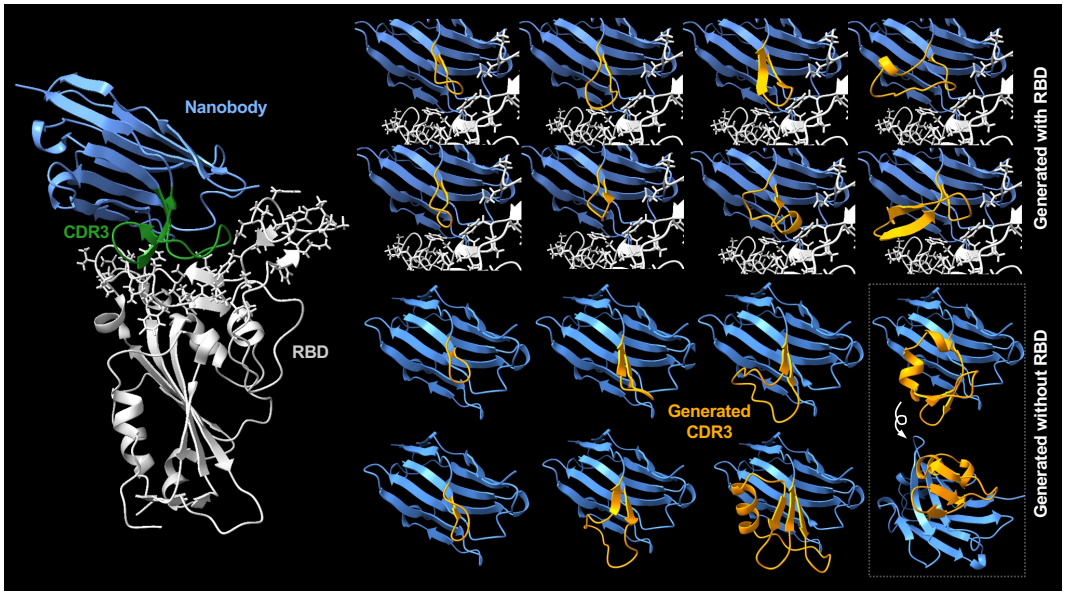


Figure 17: **Branching Flows protein 'infix' samples.** Depicted is the template PDB (9IQP, Shcheblyakov et al. [2025]) where only the CDR3 (green in the template) was designable, and the rest of the structure was fixed and conditioned upon. The top two rows of samples show generated CDR3s when they are generated in the context of the binding partner, and the bottom two rows show generations in the absence of the binding partner.

CDR3s had no atomic clashes with the RBD side chain rotamers for 15 out of 20 of the generated CDR3s. Some of the unbound CDR3s appear structurally plausible, but are extremely unlikely to occur naturally, which is not surprising since neither the base model nor BranchSegment have any antibody or nanobody-specific training.

H.4.1 Branching ChainStorm compute resources

Each ChainStorm finetune takes approximately 3 days on a single Nvidia RTX 6000 Ada GPU.

H.5 La Proteina Branching Flows fine-tuning and evaluation

H.5.1 Branching-flow architecture

We fine-tuned the unconditional La Proteina LD3 model (weights license: NVIDIA Open Model License Agreement) with Branching Flows. The base network was the 160M-parameter local-latents transformer with 14 transformer layers, token dimension 768, 12 attention heads, conditioning dimension 256, and output parameterization as vector fields for both C_α coordinates and local latent variables. The Branching Flows variant wrapped this base transformer and added only the machinery required to model insertions and deletions.

Two additional per-residue heads were attached to the final sequence embeddings of the LD3 trunk: a split head and a deletion head. Each indel head was a two-layer MLP,

$$768 \to 768 \to 1,$$

with a SiLU nonlinearity between layers. The first linear layer used a bias and the final scalar projection did not. The C_α and local-latent vector-field output heads were otherwise the original LD3 heads. Unlike the original La Proteina, the Branching Flows variant used the same scalar time t for C_α and local-latent conditioning during Branching Flows training and sampling.

H.5.2 OU bridge process and Branching Flows setup

Training bridges and Branching Flows sampling used an endpoint-conditioned Ornstein–Uhlenbeck process (as above) rather than the ODE process from the original model. Following the insight from La Proteina [Geffner et al., 2025] that co-designability improves when the latent side chains are resolved later in the conditional process, we chose OU parameters to give similar behavior. For C_α coordinates during training, we used the same OU process as ChainStorm, but for the latent side

chains we used a mean reversion rate of 5, and a constant variance of 5 (i.e. we do not decay the variance, and the termination at the endpoint is due to the conditioning drift only).

The split-time distribution was a $\mathrm{Beta}(1, 2)$ distribution rescaled to support $[0, 0.95]$ to prevent late split events (which was identified as an issue specific to La Proteina in an initial training attempt). To-be-deleted elements were sampled from a Poisson with mean 10% of the length of the training sample.

H.5.3 Fine-tuning setup

The model was initialized from the LD3 checkpoint. The (frozen) VAE from LD3 was used to generate side chain latents for the Branching Flows finetune.

Fine-tuning was run for 1,000,000 training steps on one GPU with batch size 8 (via gradient accumulation). Training used full float32 precision. Self-conditioning was enabled: before the gradient pass, a Poisson(1) number of no-gradient self-conditioning passes was sampled, and the resulting predicted endpoints were fed back as self-conditioning inputs.

The optimization used Muon for the transformer stack matrix parameters and AdamW for the remaining parameters, including the input/conditioning factories, output heads, and the new indel heads. Muon used momentum 0.95, Nesterov momentum, weight decay 0.01, $\epsilon = 10^{-7}$, and 5 Newton–Schulz steps. AdamW used learning rate 0.015 times the Muon learning rate, betas $(0.9, 0.95)$, $\epsilon = 10^{-10}$, and weight decay 0.01. Gradients were clipped to norm 1.0.

The learning-rate schedule started at 5×10^{-6}, inflated by a factor 1.05 at each learning-rate update until reaching the target 10^{-4}, and then decayed multiplicatively by 0.99995. Learning-rate updates occurred every 10 optimizer updates. A final warmdown was enabled for the last 7,810 optimizer updates, linearly decaying the learning rate to 10^{-9}.

H.5.4 Fixed-length La Proteina LD3 baselines

For comparison, we generated fixed-length samples from the original LD3 model, which is a recent strong fixed-length protein sampler, using lengths matched one-to-one to the 500 Branching Flows samples. Each fixed-length LD3 sample was indexed by the corresponding Branching Flows sample it matched. La Proteina, like most other protein models, uses annealed noise schedules to improve designability when sampling. Two LD3 noise settings were evaluated: the default La Proteina inference setting from the codebase $(0.15, 0.05)$ for C_α and local latents, and an alternative $(0.1, 0.1)$ which is what was used in the La Proteina manuscript.

The original fixed-length LD3 sampler used $N = 400$ steps and self-conditioning. The C_α and local-latent modalities used different time grids. C_α used the log schedule

$$\big(t_i\big)_{i=0}^{N} = 1 - \mathrm{logspace}(-2, 0, N+1)^{\mathrm{reversed}},$$

renormalized to span $[0, 1]$. Local latents used a power schedule $t_i = (i/N)^2$. C_α used $g(t) = 1/(t + 10^{-2})$, while local latents used

$$g(t) = \frac{\pi}{2} \frac{\sin((1-t)\pi/2)}{\cos((1-t)\pi/2) + 10^{-2}}.$$

Both modalities used stochastic score-corrected sampling mode. For $t \leq 0.98$, the update was

$$x_{t+\Delta t} = x_t + \{v_\theta(x_t, t) + g(t) s_\theta(x_t, t)\}\Delta t + \sqrt{2g(t)\lambda \Delta t}\, \epsilon,$$

where λ was the modality-specific noise scale. For $t > 0.98$, the implementation switched off stochastic noise and used a low-temperature ODE update with the inferred score scaled by 1.5. C_α coordinates were re-centered after every step; local latents were not.

The main text table shows metrics for samples from the $(0.15, 0.05)$ model (which performed better on our metrics) and table 3 shows both LD3 annealing options.

H.5.5 Branching-flow sampling and annealing

For the final Branching Flows LD3 samples reported here, we used the fine-tuned checkpoint above with 500 sampling steps, self-conditioning enabled. Sampling used the cosine time grid as in the Branching Flows ChainStorm finetune. Annealing for Branching Flows models is underexplored, and here we modified our OU noise (just as the original La Proteina modifies their SDE noise) to achieve a similar effect.

The C_α OU process used a flat annealing value $v_0 = 60$, with $v_1 = 10^{-9}$ held near zero. The local-latent OU process (calibrated to resolve later) used $\theta = 5$, $\beta = -0.001$, and linearly annealed variance from 5 at $t = 0$ to 0 at $t = 1$.

H.5.6 Designability, co-designability, and diversity metrics

All reported metric sets used $N = 500$ generated structures. MPNN1 designability was computed by running ProteinMPNN once per generated backbone, folding the single designed sequence with ESMFold, and computing C_α TM-score between the ESMFold structure and the generated structure. A sample was counted as designable when $\mathrm{scTM} > 0.5$ [Lin and AlQuraishi, 2023]

Co-designability used the original sequence sampled from the generated latent side chains rather than a ProteinMPNN-designed sequence. This sequence was folded with ESMFold and compared back to the generated structure using the same C_α TM- score criterion. A sample was counted as co-designable when $\mathrm{scTM} > 0.5$.

Diversity was computed with Foldseek [van Kempen et al., 2024] clustering on the generated PDBs. The reported structure-only diversity used Foldseek `easy-cluster` with `-alignment-type 1`, `-cov-mode 0`, `-min-seq-id 0.0`, and `-tmscore-threshold 0.5`. The reported structure-plus-sequence diversity used `-alignment-type 2`, `-cov-mode 0`, `-min-seq-id 0.1`, and `-tmscore-threshold 0.5`. For binomial designability and co- designability proportions, the table reports symmetric two-sigma Wald intervals.

Sample set	MPNN1 designability	Co-designability	Div. str	Div. str+seq
Branching Flows	0.990 ± 0.009	0.966 ± 0.016	128	215
LD3 matched, $0.15, 0.05$	0.972 ± 0.015	0.952 ± 0.019	110	173
LD3 matched, $0.1, 0.1$	0.964 ± 0.017	0.950 ± 0.019	96	147

Table 3: Designability, co-designability, and diversity for $N = 500$ samples. Intervals are symmetric two-sigma Wald intervals for binomial proportions.

H.5.7 Branching La Proteina compute resources

Finetuning Branching Flows into the La Proteina LD3 model takes approximately one day on an Nvidia RTX 6000 Blackwell.

H.6 Training and Inference Timing

For timing comparisons we compare matched optimized implementations of variable length models, including Branching Flows (with 'insertion only' variants which have shorter average state lengths, and 'deletion only' variants with longer average state lengths) as well as Oracle Length models for QM9. Edit Flows is included for discrete antibody sequences.

During training, the CPU is preparing batches in parallel while the GPU is processing them and, just as in regular flow matching (whose asymptotic cost we match), the batch preparation is much faster than the GPU training pass, so there is no overall cost to constant-factor batch preparation overhead. As can be seen in figure 18, the only thing that matters for training throughput is how many tokens the large neural network is processing.

For sampling: the insertion and deletion bookkeeping is negligible (just editing a single token sequence) compared to the large forward model pass, and can happen on-device. So, as in training, the cost is entirely dominated by the number of tokens that the model sees, and also as in training, Branching Flows can have a (slight) disadvantage when starting from a longer sequence (e.g. the 'deletion only' model), or an advantage when starting from a single element. For each task, these timings are architecture matched to the maximum extent (except where e.g. Branching Flows vs Edit Flows vs Fixed-Length differ in number of output heads).

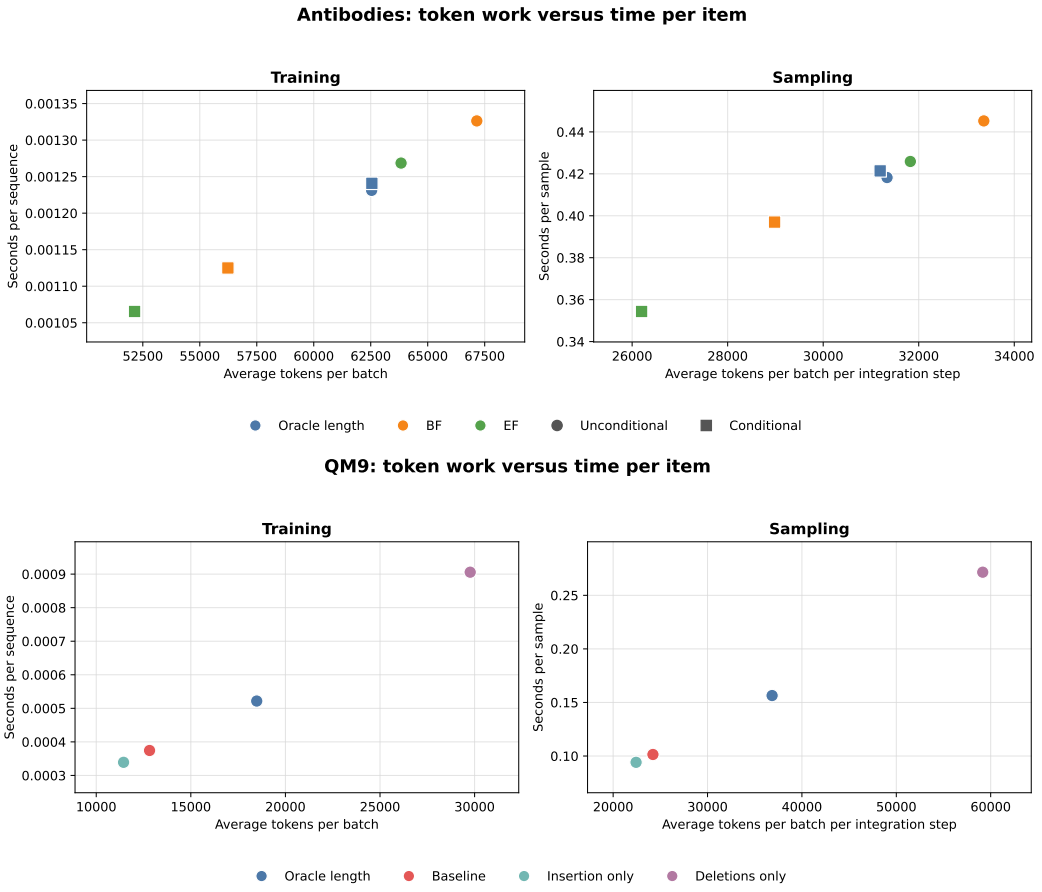


Figure 18: **Timing plots.** Training (left) and sampling (right) for antibody (top) and QM9 (bottom) models. Training and sampling times differ across approaches, but this is driven entirely by the number of tokens processed by the transformer, which can e.g. be lower on average for Branching Flows vs an "oracle length" model, since for lower values of t the number of elements in the sequence is typically lower than the observation length (during training) and the final sample length (during sampling).

Table 4: Training timing. Tokens/batch is the mean number of live tokens per optimizer update. Steps/s counts optimizer updates.

Task	Conditioning	Model	Batch	Tokens/batch	Steps/s	Samples/s	Peak memory
Antibodies	Unconditional	Oracle length	512	62539.5	1.586	812.2	20.97 GiB
Antibodies	Unconditional	BF	512	67152.8	1.473	754.0	22.45 GiB
Antibodies	Unconditional	EF	512	63830.7	1.540	788.3	21.43 GiB
Antibodies	Conditional	Oracle length	512	62548.7	1.574	805.9	21.06 GiB
Antibodies	Conditional	BF	512	56232.3	1.736	888.9	19.28 GiB
Antibodies	Conditional	EF	512	52138.7	1.833	938.6	17.97 GiB
QM9	Unconditional	Oracle length	1024	18483.3	1.872	1916.8	18.57 GiB
QM9	Unconditional	Baseline	1024	12818.7	2.608	2671.1	13.07 GiB
QM9	Unconditional	Insertion only	1024	11444.7	2.880	2949.0	11.63 GiB
QM9	Unconditional	Deletions only	1024	29767.4	1.078	1104.1	32.77 GiB

Table 5: Sampling timing. Tokens/batch-step is the mean number of live tokens processed per integration step across the batch. Antibody and QM9 flow samplers use 1,000 integration steps per sample.

Task	Conditioning	Model	Batch	Tokens/batch-step	Samples/s	Sample-steps/s	Peak memory
Antibodies	Unconditional	Oracle length	256	31336.5	2.391	2391.1	1.14 GiB
Antibodies	Unconditional	BF	256	33362.5	2.246	2246.5	1.21 GiB
Antibodies	Unconditional	EF	256	31824.5	2.348	2348.3	1.17 GiB
Antibodies	Conditional	Oracle length	256	31193.0	2.373	2373.3	1.14 GiB
Antibodies	Conditional	BF	256	28982.3	2.519	2518.7	1.16 GiB
Antibodies	Conditional	EF	256	26198.2	2.822	2821.9	1.14 GiB
QM9	Unconditional	Oracle length	2048	36841.5	6.392	6391.7	1.71 GiB
QM9	Unconditional	Baseline	2048	24214.3	9.859	9858.8	1.41 GiB
QM9	Unconditional	Insertion only	2048	22430.3	10.624	10624.0	1.39 GiB
QM9	Unconditional	Deletions only	2048	59156.5	3.682	3681.8	4.27 GiB

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract states that we develop Branching Flows, and demonstrate competent variable-length distribution matching on (multimodal) small molecules and (discrete) antibody sequences, and that the method also scales to complex domains such as protein structures. This is precisely what we show in the experiments in the manuscript.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: At the end of the main text, we discuss what we view as the key limitations of this work, both in our experimental results and in the generality of the method presented.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [Yes].

Justification: Full proofs for each new result are provided, and great care has been taken to ensure these proofs are correct, having gone through several rounds of extensive internal review. We have also taken care to state our assumptions explicitly, according to the norms in this field, either explicitly stated in this manuscript or cited (eg. in Holderrieth et al. [2024], Billera et al. [2026]).

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: In each empirical section we specify the data source, describe the model architecture, training details, and the specific Branching Flows configuration we use. The level of detail is sufficient for an independent replication of the empirical results.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [No]

   Justification: We link to a flexible implementation framework that makes it simpler to construct branching flows models, which will include links to key models from this manuscript, but we do not have a 'one push' reproducible environment that can replicate the many experiments across this publication, and so we answer No here.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: We have provided extensive details about the experimental setting in the appendix for each experiment.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: We use Wald two-sigma intervals for Bernoulli (pass/fail) metrics, and this is described in the figure captions and in appendix.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: All experiments are GPU compute-limited, and at the end of each empirical section in the appendix we describe the GPU compute required.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: This work does not involve sensitive data, human subjects research, or any other potentially controversial issues, and there are no potential societal harms to mitigate. All models and code used are properly licensed, and the work itself is otherwise consistent with the ethics guidelines.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: We include an explicit Broader Impacts section in the Appendix.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: No pre-trained language models, image generators or models for biological data with high misuse risk are introduced in this paper. All training datasets used are public, and no specific dataset is released.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We list licenses of models/assets used/referred to in the appendix text.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [N/A]

    Justification: The paper does not in its present form release new assets.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: No crowdsourcing or research on human subjects was performed.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

Justification: No crowdsourcing or research on human subjects was performed.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: Even though we do not use LLMs as non-standard components of the core methods, we prefer to explicitly declare the usage of Large Language Models in an Appendix section.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.