

UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

UNIVERSITY OF MODENA AND REGGIO EMILIA

"Enzo Ferrari" Department of Engineering

Master's Degree in Artificial Intelligence Engineering (LM-32)

Diffusion Language Models: An Experimental Analysis

Supervisor:
Prof. Lorenzo Baraldi
Prof. Marcella Cornia
Co-supervisor:
Davide Bucciarelli
Leonardo Zini

Candidate:
Thomas Bertolani
Student ID 203142

ACADEMIC YEAR 2025/2026

Abstract

Diffusion Language Models: An Experimental Analysis

Large Language Models (LLMs) have revolutionized language modeling through autoregressive generation, enabling strong performance across a wide range of tasks. Recently, Diffusion Language Models (DLMs) have emerged as an alternative paradigm that generates text through iterative denoising rather than next-token prediction, allowing parallel refinement of entire sequences. While numerous diffusion-based architectures have been proposed, differences in evaluation protocols, datasets, inference budgets, and generation hyperparameters make it difficult to compare their capabilities and understand the trade-offs they offer. In this work, we present a systematic experimental analysis of modern DLMs. Specifically, we evaluate eight state-of-the-art DLMs across eight benchmarks spanning reasoning, coding, translation, knowledge, and structured problem solving, while explicitly considering both generation quality and computational efficiency. Beyond downstream evaluation, we analyze the impact of key inference-time factors, including denoising steps, context length, block size, and parallel unmasking strategies, and complement large-scale experiments with controlled comparisons of smaller models trained under identical conditions. Our analysis highlights the strengths and limitations of diffusion-based language modeling across different tasks, architectures, and inference budgets. We show that the behavior of DLMs is strongly influenced by generation-time design choices, leading to distinct trade-offs between performance and computational efficiency. Overall, our study provides practical insights into the capabilities and deployment characteristics of contemporary DLMs.

Keywords: Diffusion LM, Non-Autoregressive, Discrete Diffusion, Large Language Model, Parallel Generation

Contents

Abstract **ii**

1 Introduction **1**

1.1 Motivation 1

1.2 The Evaluation Problem 2

1.3 Research Questions and Scope 3

1.4 Contribution and Overview of the Findings 4

1.5 Thesis Organization 5

2 Background **6**

2.1 Language Modeling 6

2.1.1 Text Representation 6

2.1.2 Learning from Text 7

2.1.3 Evaluation and Perplexity 8

2.2 Transformers and Large Language Models 9

2.2.1 Token Embeddings and Positional Information 10

2.2.2 Self-Attention and Multi-Head Attention 10

2.2.3 Transformer Blocks and Attention Patterns 12

2.2.4 Modern Decoder-Only Models 14

2.2.5 Pretraining and Post-Training 15

2.3 Autoregressive Language Modeling 16

2.3.1 Training and Left-to-Right Generation 16

2.3.2 Prefill and Cached Incremental Decoding 17

2.4 Diffusion as a Generative Paradigm 18

2.4.1 Gaussian Corruption 18

2.4.2 Learning to Reverse the Process 19

2.5 Discrete Diffusion 20

2.5.1 Categorical Transitions . . . 20
2.5.2 Uniform and Absorbing Corruption . . . 21
2.5.3 Reverse Transitions and Learning . . . 22
2.6 Masked Diffusion for Language . . . 23
2.6.1 Training with Masked Sequences . . . 23
2.6.2 Conditional Generation and Denoising Steps . . . 24
2.7 Block Diffusion . . . 26
2.7.1 Generation Within and Between Blocks . . . 26
2.7.2 Block Size, Step Budget, and Reuse . . . 27
2.7.3 Computational Implications . . . 28
2.8 Flow Matching as a Generative Paradigm . . . 29

3 Diffusion Language Models and Related Work 31
3.1 From Continuous to Discrete Language Diffusion . . . 32
3.2 Discrete Diffusion Language Models . . . 33
3.3 Scaling Diffusion Language Models . . . 34
3.4 Block and Hybrid Diffusion Models . . . 35
3.5 Further Developments and Related Approaches . . . 37
3.5.1 Scaling and Decoding . . . 38
3.5.2 Multimodal Extensions . . . 38
3.5.3 Flow-Based Language Models . . . 39
3.6 Evaluation in Existing Literature . . . 40

4 Experimental Methodology 41
4.1 Evaluation Design . . . 41
4.2 Evaluated Models . . . 42
4.3 Benchmarks . . . 42
4.3.1 Knowledge and Commonsense . . . 43
4.3.2 Mathematics and Code . . . 44
4.3.3 Translation and Structured Constraints . . . 44
4.4 Evaluation Protocol . . . 45

4.4.1 Candidate Scoring and Generated Answers . . . 45
4.4.2 Likelihood-Based Evaluation . . . 46
4.4.3 Selecting the Reported Configuration . . . 46
4.5 Inference-Time Analysis Design . . . 46
4.6 Computational-Cost Methodology . . . 48
4.6.1 Workload and Aggregation . . . 48
4.6.2 FLOP Estimation . . . 48
4.7 Small-Scale Matched-Corpus Methodology . . . 49

5 Experiments and Results **51**
5.1 Overall Downstream Evaluation . . . 51
5.2 Inference-Time Hyperparameter Analysis . . . 53
5.2.1 Generation Length and Steps Varied Together . . . 53
5.2.2 Steps at Fixed Generation Length . . . 55
5.2.3 Effect of Block Size . . . 56
5.2.4 Denoising Steps Within a Block . . . 57
5.3 Computational-Cost Analysis . . . 58
5.4 Small-Scale Matched-Corpus Analysis . . . 60
5.5 Cross-Experiment Discussion . . . 61
5.5.1 Task Performance Depends on the Model and Its Configuration . . . 61
5.5.2 Denoising Steps and Computational Cost . . . 62
5.5.3 Implications for Selecting an Inference Configuration . . . 62

6 Conclusions **63**
6.1 Answers to the Research Questions . . . 63
6.2 Overall Contribution . . . 64
6.3 Future Directions . . . 64

References **66**

List of Figures

2.1 The original Transformer encoder and decoder. Each stack repeats its attention and feed-forward sublayers; the decoder also attends to the encoder output. Figure from [68]. 9

2.2 Attention computation in the Transformer. The diagrams show the weighting of values within one head and the learned projections and combination of multiple heads. Figure from [68]. 12

2.3 Permitted attention connections for a sequence of four tokens. Rows correspond to queries and columns to keys. A value of 1 permits attention, while 0 marks a connection whose score is set to $-\infty$ before the softmax. Padding is omitted. Causal attention restricts each row to the current and preceding tokens; bidirectional attention permits access to the entire input. 13

2.4 Forward corruption q and learned reverse transitions p_θ in a Gaussian diffusion model. The forward process progresses from $\mathbf{x}_0$ to $\mathbf{x}_T$; generation starts at $\mathbf{x}_T$ and follows the reverse process toward $\mathbf{x}_0$. Figure from [33]. 18

2.5 Two illustrative text corruption trajectories: (a) absorbing-state masking and (b) uniform corruption. The trajectories illustrate their contrasting effects rather than the exact transition rules defined above. Figure from [5]. 22

2.6 Training and sampling in LLaDA. Panels show random masking for pretraining, response masking for supervised fine-tuning, and repeated prediction and remasking during generation. Figure from [49]. 24

2.7 Block-causal attention for four tokens arranged in two blocks of two. Rows represent queries and columns represent keys, following Figure 2.3; the prompt is omitted. Each block has bidirectional access internally and can attend to earlier blocks. The first block cannot attend to the second, so changes in the latter do not affect the first block representations. 28

2.8 Illustrative paths between a source and a target distribution. The diffusion example follows a stochastic trajectory, whereas the flow example follows a trajectory defined by a velocity field. Figure from [43]. 29
3.1 Selected diffusion language models in chronological order, grouped by discrete, continuous, and multimodal approaches. Figure from [40]. 31
3.2 Block decoding in Fast-dLLM v2. Decoded blocks are cached as context, while decoding and cache reuse continue within the active block. Figure from [70]. . . . 37
3.3 Image-conditioned text generation in LaViDa. A vision encoder and projector provide visual context to a diffusion language model, which predicts masked response tokens. Figure from [39]. 39
5.1 Dream and LLaDA as generation length and denoising steps increase together. Generation length counts output tokens only. Panels show WMT16 En–De, MBPP, HumanEval, and GSM8K from top left to bottom right. 54
5.2 Dream and LLaDA with a fixed generation length of 1,024 output tokens and different numbers of denoising steps. Panels show WMT16 En–De, MBPP, HumanEval, and GSM8K from top left to bottom right. 55
5.3 Scores as block size changes from 8 to 128 tokens. LLaDA and LLaDA 1.5 use one denoising step per output token; Fast-dLLM v2 follows its own decoding procedure without this constraint. Panels show WMT16 En–De, MBPP, HumanEval, and GSM8K from top left to bottom right. 57
5.4 LLaDA and LLaDA 1.5 as the number of denoising steps per block token increases from $1/8$ to 1. Panels show WMT16 En–De, MBPP, HumanEval, and GSM8K from top left to bottom right. 58

1. Introduction

Large language models (LLMs) have achieved strong performance on tasks such as question answering, reasoning, code generation, and translation. Most of these models use autoregressive generation, predicting each new token from the preceding text [13, 72]. Despite its effectiveness, ordinary decoding follows a fixed left-to-right order: each prediction depends on the output already produced. This limits parallel generation and prevents the use of later context when making an earlier decision. These constraints motivate the investigation of alternative generation procedures.

Diffusion language models (DLMs) approach text generation through iterative denoising. Their development builds on the success of diffusion models in image generation [33], where a model learns to recover data from progressively corrupted versions. Early approaches to language generation applied this idea to continuous token representations [41]. Discrete formulations [5] instead define corruption directly over categorical states, for example by replacing tokens with masks. Generation then starts from a noisy or masked sequence and progressively produces a completed output.

This formulation offers a different way to use context during generation. A denoising model can make several predictions in one pass, using the visible context on both sides of each mask. These properties make diffusion relevant to parallel generation, infilling, and tasks with constraints across the output. Recent work has extended masked diffusion to the scale of large language models [49, 75], enabling comparisons with autoregressive systems across a broad range of language tasks.

The growing range of diffusion architectures raises a practical question: how do their different generation procedures affect the quality and computational requirements of their outputs? This thesis investigates that question through an experimental comparison of diffusion language models and autoregressive baselines.

1.1 Motivation

The central motivation is to understand how the flexibility of diffusion generation translates into useful language-modeling capabilities. Bidirectional context is particularly interesting when an out-

put must satisfy several related constraints. In Sudoku, for example, a value must agree with entries in its row, column, and subgrid. Code contains a different kind of dependency: expressions must be consistent with names and control flow elsewhere in the program. Evaluating such tasks alongside knowledge and translation benchmarks helps establish where different generation procedures are effective.

A second motivation concerns the relationship between generation quality and computational requirements. Diffusion models expose a choice about how many denoising steps to perform. Fewer steps allow more tokens to be generated in parallel, but those tokens are chosen from the same partially completed context. Additional steps allow later predictions to use more of the model earlier decisions. The useful balance between these choices needs to be established experimentally for different tasks.

The amount of work performed at each step also matters. A full-sequence denoiser can repeatedly process the entire reserved output. Block-based approaches organize generation around smaller groups of tokens and reusable context [2, 70]. Comparing these procedures through task scores, inference sweeps, and profiling provides a clearer account of their practical requirements and the performance they can achieve.

1.2 The Evaluation Problem

Comparing diffusion language models requires accounting for how they were trained as well as how they generate text. Some are trained from scratch, while others are adapted from an autoregressive checkpoint; training corpora and post-training procedures also differ. Comparisons between released models therefore assess systems with distinct training histories, while changing inference settings within one model allows those settings to be studied with its learned parameters held fixed.

Published scores introduce another difficulty: studies can use different tasks, prompt formats, scoring procedures, and generation budgets. A strong score may depend on a particular inference configuration, and that configuration may not suit another task. A useful comparison must therefore document how scores were obtained and examine whether performance remains strong as the generation settings change.

The breadth of the suite is important. Answering a multiple-choice question requires choosing

between supplied candidates. Generating a program requires producing code that passes tests. Translation is assessed against reference text, while Sudoku requires satisfaction of explicit constraints. A model can be strong on one of these tasks and weaker on another. Reporting the results separately preserves these differences and avoids reducing unlike measures to a single score.

1.3 Research Questions and Scope

The study addresses three research questions:

RQ1 How do diffusion language models compare with one another and with autoregressive models across language tasks?

RQ2 How do inference-time hyperparameters affect performance?

RQ3 How much GPU memory and estimated computation do the models require during generation?

The first question is addressed through eight downstream benchmarks covering knowledge, commonsense, mathematics, code, translation, and structured constraint satisfaction. The comparison includes five large diffusion models and two autoregressive baselines, reporting the best result observed for each model and benchmark among the tested settings.

The second question concerns the inference procedure applied to a fixed model. Four experiments examine generation length and steps varied together, steps at a fixed generation length, block size, and the number of steps within a block. These comparisons distinguish the configured number of output tokens from the procedure used to generate them. Throughout the thesis, generation length counts output tokens and excludes the prompt.

The third question is addressed through separate profiling runs that measure peak allocated GPU memory and estimate floating-point operations (FLOPs). Chapter 4 defines the workloads, aggregation, and estimation procedure. These measurements concern storage and computational work; the study does not measure elapsed generation time.

A complementary analysis compares four smaller diffusion models pretrained on OpenWebText with a GPT-2 reference. It evaluates perplexity on a mixture of texts and computational requirements for unconditional generation. This extends the study to reference-text prediction under a shared training-corpus setting.

The experimental focus is text-only diffusion language modeling. Modern autoregressive families, newer diffusion systems, multimodal extensions, and flow-based language models appear in the background and related work to place the study within the wider field.

1.4 Contribution and Overview of the Findings

The thesis develops the experimental work presented in *Diffusion Language Models: An Experimental Analysis* [7]. The contribution has four parts:

1. **Identifying task-specific strengths.** No single model leads across all benchmarks. The evaluated architectures offer different advantages depending on the task, and the comparison identifies where each achieves strong results relative to the other diffusion models and autoregressive baselines.

2. **Separating the effects of inference choices.** The experiments show that longer outputs can stop helping even when additional denoising steps at a fixed length remain useful. Changing block size often has a smaller effect than reducing the number of steps within a block, particularly for code generation. Model rankings can also change with the available step budget.

3. **Characterizing generation requirements.** Profiling shows that repeated full-sequence denoising in the studied high-step setting has much larger estimated FLOP counts than the evaluated autoregressive and block-oriented implementations, while peak-memory differences are smaller. The analysis explains these results through the work accumulated during generation.

4. **Comparing predictive quality at smaller scale.** Duo has the lowest reported perplexity among the smaller diffusion models and comes closest to GPT-2. The diffusion models have similar estimated generation FLOPs in the studied workload, showing that comparable generation work can accompany different predictive quality.

1.5 Thesis Organization

Chapter 2 introduces language modeling, Transformers, autoregressive inference, and the diffusion mechanisms needed to understand the experiments. It progresses from continuous and discrete corruption to masked and block generation, and introduces flow matching as a related generative framework.

Chapter 3 surveys diffusion language models through their representations, training routes, and generation procedures. It positions the evaluated models within this development and covers additional models as field context before motivating the experimental comparison.

Chapter 4 defines the evaluation design, models, benchmarks, scoring protocols, inference sweeps, and computational measurements. It also describes the smaller-model analysis.

Chapter 5 presents and interprets the results in the same order, then connects the findings across experiments. Chapter 6 answers the research questions, summarizes the contribution, and identifies directions for future work.

2. Background

This chapter develops the concepts needed to compare autoregressive and diffusion language models. It first introduces language modeling and the Transformer architecture, then explains how autoregressive models learn and generate text. The diffusion foundations then progress from continuous corruption to discrete tokens, masked generation, and generation in blocks. A final section introduces flow matching as context for related approaches surveyed in Chapter 3. The experimental focus remains the evaluation of diffusion language models; the background explains the inference controls and computational mechanisms needed to interpret that evaluation.

2.1 Language Modeling

Language modeling aims to learn a probability distribution over text. Given a sequence, a language model assigns a probability that reflects how well it agrees with the patterns learned from data. The same probabilistic model can support both the evaluation of existing text and the generation of new text. Learning useful probabilities requires generalization: a model must assign probability to sequences that it has never encountered, rather than simply memorize a collection of sentences. Neural language models address this problem by learning token representations and using the same model parameters to process many different sequences. The model can therefore use what it has learned from one sequence to make predictions about other sequences [6].

2.1.1 Text Representation

Before text can be processed by a model, it is converted into a sequence of discrete symbols called *tokens*. A tokenizer specifies how a string is segmented and maps the resulting units to entries in a finite vocabulary $\mathcal{V}$. A token may represent a word, part of a word, a character, or a byte, depending on the tokenization scheme. Thus, the number of tokens in a sentence need not coincide with its number of words. Subword tokenization allows words to be represented as combinations of smaller units, enabling the representation of rare or previously unseen words without requiring a separate vocabulary entry for every word [60].

We denote a tokenized sequence by $\mathbf{x} = (x_1, \ldots, x_n)$, with $x_i \in \mathcal{V}$, where n is its length and i indexes a token position. The vocabulary may also contain special symbols that mark sequence boundaries or serve other model-specific purposes. For example, an end-of-sequence token can indicate that a generated sequence is complete. The integer identifier assigned to a token is an index into the vocabulary; it does not itself express semantic similarity.

A probabilistic language model with parameters θ assigns a joint probability $p_\theta(\mathbf{x})$ to the sequence. Autoregressive and diffusion models differ in how they represent and learn this distribution.

2.1.2 Learning from Text

Language models can learn from text without manually annotated targets. The training task is constructed from the text itself, an approach known as *self-supervised learning*. Two common tasks are next-token prediction and masked-token reconstruction.

In next-token prediction, the model receives the tokens preceding a target token and learns to predict that token. The original sequence supplies the correct answer for each prediction. Applying this task throughout the corpus teaches the model how text continues from a given prefix.

In masked-token reconstruction, some tokens are hidden and the model learns to recover their original values from the remaining text. Unlike next-token prediction, this task can use information from both sides of a missing token. BERT [22] uses this form of training to learn representations from unlabeled text. Both tasks obtain their targets from the corpus, but they differ in the information available to make each prediction.

Learning from these tasks requires an objective that guides how the model parameters are updated. For a generative model with sequence probabilities $p_\theta(\mathbf{x})$, a standard goal is to make the observed training text as probable as possible. This principle is known as *maximum-likelihood estimation* [6].

The same goal can be expressed by minimizing the *negative log-likelihood* (NLL): a loss that penalizes the model when it assigns low probability to the training text. For a training corpus $\mathcal{D}$ containing N tokens in total, the loss per token is

$$\mathcal{L}_{\mathrm{NLL}}(\theta; \mathcal{D}) = -\frac{1}{N} \sum_{\mathbf{x} \in \mathcal{D}} \log p_\theta(\mathbf{x}). \tag{2.1}$$

The sum includes each sequence in the corpus, while division by N normalizes the loss by the total

number of tokens. Training adjusts θ to reduce this loss, increasing the likelihood of the corpus.

At the level of an individual token prediction, the corresponding loss is often expressed as *cross-entropy*. If $q(v)$ is a target distribution over vocabulary entries and $r_\theta(v)$ is the predicted distribution, their cross-entropy is

$$H(q, r_\theta) = -\sum_{v \in \mathcal{V}} q(v) \log r_\theta(v). \tag{2.2}$$

For an observed target token y, the target distribution places all its mass on y, and Equation 2.2 reduces to $-\log r_\theta(y)$. Increasing the probability assigned to the correct token therefore reduces the loss. The relation between token prediction losses and the sequence objective depends on the model probabilistic formulation. In particular, models for which exact likelihood is intractable can require a bound or another tractable training objective.

2.1.3 Evaluation and Perplexity

Language-modeling performance is evaluated on held-out text to assess generalization beyond the training corpus. For an evaluation set $\mathcal{D}_{\text{eval}}$ containing N_{eval} scored tokens, the average negative log-likelihood and its associated *perplexity* [6] are

$$\begin{aligned} \overline{\mathcal{L}}_{\text{eval}} &= -\frac{1}{N_{\text{eval}}} \sum_{\mathbf{x} \in \mathcal{D}_{\text{eval}}} \log p_\theta(\mathbf{x}), \\ \text{PPL}(\mathcal{D}_{\text{eval}}) &= \exp\left(\overline{\mathcal{L}}_{\text{eval}}\right). \end{aligned} \tag{2.3}$$

Lower perplexity corresponds to greater probability assigned to the observed tokens under the evaluation protocol. Its scale can be understood as an effective number of equally likely alternatives: a predictor that assigns probability $1/k$ to each observed token has perplexity k. This is an interpretation of the prediction loss, not the number of tokens in the model vocabulary.

For a corpus of sequences with different lengths, Equation 2.3 aggregates their log-probabilities before normalizing by the total token count; it is not an unweighted average of sequence perplexities.

Perplexity comparisons require consistent evaluation data, tokenization, and scoring conventions. Changing tokenization changes both the events being predicted and the token count used for normalization. Moreover, perplexity evaluates the probability of reference text, whereas downstream benchmarks assess properties such as answer correctness or satisfaction of task constraints.

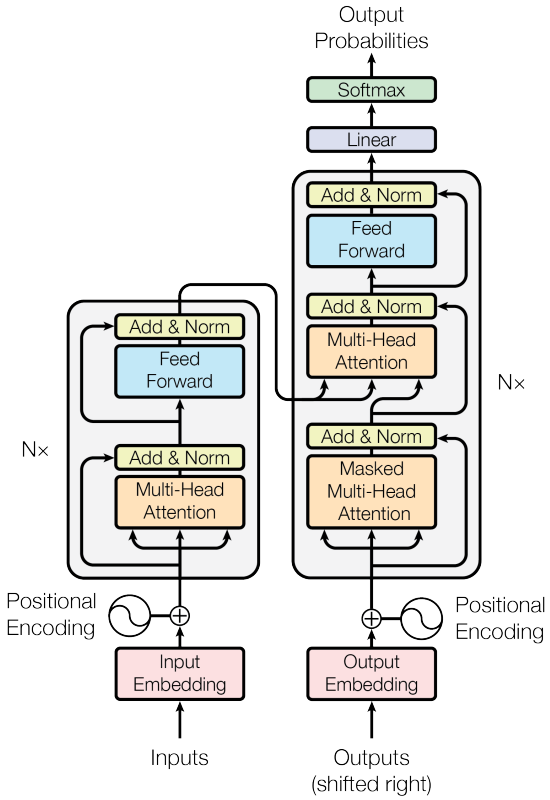


Figure 2.1: The original Transformer encoder and decoder. Each stack repeats its attention and feed-forward sublayers; the decoder also attends to the encoder output. Figure from [68].

2.2 Transformers and Large Language Models

The Transformer [68] provides a neural architecture for processing token sequences through attention. Its layers build a representation for each token by combining information from other tokens in the sequence, subject to the restrictions imposed by an attention mask. This makes attention relevant to both next-token prediction and masked-token reconstruction: the architecture can support different prediction tasks by changing which connections are available between tokens.

Figure 2.1 gives an overview of the original encoder–decoder architecture. Its components are introduced below before the chapter turns to decoder-only language models.

2.2.1 Token Embeddings and Positional Information

A token must first be mapped to a continuous vector that the network can process. An embedding matrix $\mathbf{E} \in \mathbb{R}^{|\mathcal{V}| \times d}$ associates each vocabulary entry with a learned vector of dimension d, called the model dimension. Looking up token x_i selects the row $\mathbf{E}[x_i, :]$. The same token initially receives the same embedding wherever it appears; subsequent layers adapt its representation using the surrounding text.

Token identity alone does not specify sequence order. A model must also distinguish where a token appears, since changing the order of the same tokens can change the meaning of a sentence. One way to supply this information is to add a positional vector $\mathbf{p}_i$ to each embedding:

$$\mathbf{h}_i^{(0)} = \mathbf{E}[x_i, :] + \mathbf{p}_i. \tag{2.4}$$

Stacking these row vectors gives the input matrix $\mathbf{H}^{(0)} \in \mathbb{R}^{n \times d}$; the superscript denotes the input to the first layer. Two choices are fixed sinusoidal encodings and learned positional embeddings. Sinusoidal encodings represent each position using sine and cosine functions with different frequencies, without introducing trainable positional parameters. Learned embeddings instead assign a trainable vector to each position, updated alongside the other model parameters. The original Transformer uses the sinusoidal approach and also evaluates learned embeddings. Both supply absolute position information through the additive formulation above. Some later architectures encode sequence order by modifying the queries and keys used in attention, rather than adding a vector to each input embedding. Section 2.2.4 introduces this approach after the attention mechanism has been explained.

2.2.2 Self-Attention and Multi-Head Attention

Self-attention lets each token combine information from other tokens in the same sequence. The combination depends on the current input: a token can assign different weights to other tokens in different sentences. To compute these weights, the layer forms three projections of its input $\mathbf{H} \in \mathbb{R}^{n \times d}$:

$$\mathbf{Q} = \mathbf{H}\mathbf{W}_Q, \qquad \mathbf{K} = \mathbf{H}\mathbf{W}_K, \qquad \mathbf{V} = \mathbf{H}\mathbf{W}_V, \tag{2.5}$$

where $\mathbf{W}_Q, \mathbf{W}_K \in \mathbb{R}^{d \times d_k}$ and $\mathbf{W}_V \in \mathbb{R}^{d \times d_v}$ are learned matrices. Bias terms are omitted for clarity. The rows of $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ are called *queries*, *keys*, and *values*.

The query for token x_i is compared with the key for token x_j to determine how strongly x_i should attend to x_j. The value for x_j contains the information that token contributes. Queries and keys therefore determine the weights, while values determine what is combined. These are learned projections rather than manually assigned linguistic roles.

Scaled dot-product attention is defined in the original Transformer as

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}. \tag{2.6}$$

The softmax is applied independently to each row, converting the scores into weights over the input tokens. Multiplication by $\mathbf{V}$ then forms a weighted sum of the value vectors for each query. Dividing by $\sqrt{d_k}$ controls the scale of the dot products, helping prevent the softmax from becoming excessively concentrated as the key dimension increases.

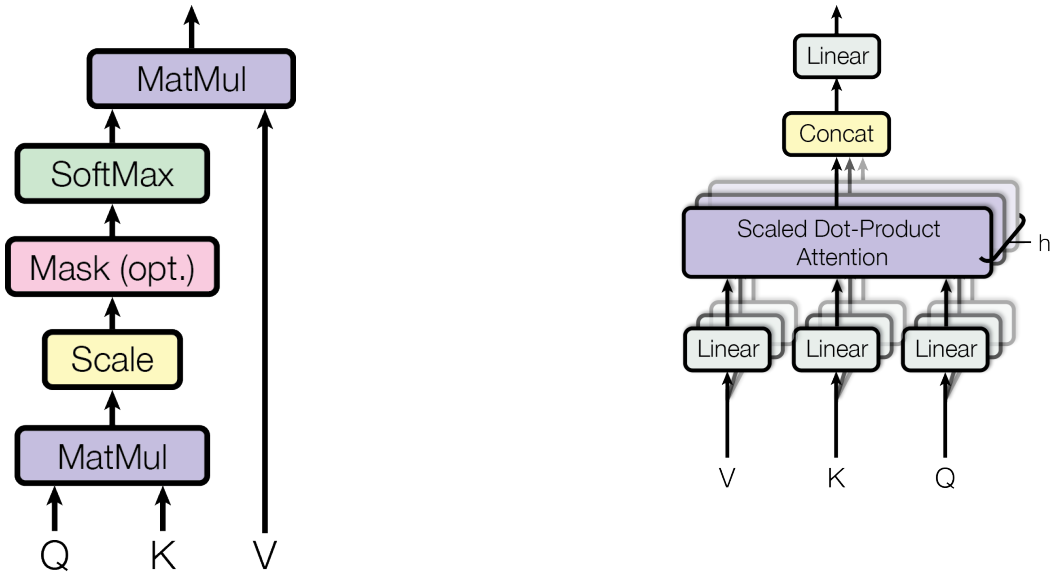
When attention must exclude certain tokens, the original Transformer sets their scores to $-\infty$ before applying the softmax. Those tokens then receive zero weight. The output for each token remains a weighted sum of the value vectors, using only the tokens that the attention pattern allows it to attend to.

A single attention head produces one set of weights for combining the values. Multi-head attention uses h separate sets of projection matrices, allowing several such combinations to be learned in parallel:

$$\begin{aligned} \mathbf{O}_a &= \text{Attention}\left(\mathbf{H}\mathbf{W}_Q^{(a)}, \mathbf{H}\mathbf{W}_K^{(a)}, \mathbf{H}\mathbf{W}_V^{(a)}\right), \qquad a = 1, \ldots, h, \\ \text{MHA}(\mathbf{H}) &= \text{Concat}(\mathbf{O}_1, \ldots, \mathbf{O}_h)\mathbf{W}_O, \end{aligned} \tag{2.7}$$

with $\mathbf{W}_O \in \mathbb{R}^{hd_v \times d}$. This expression shows the projections and combination of heads; when an attention pattern restricts access, each head masks its scores before the softmax. Concatenation joins the head outputs along their feature dimension, and the final projection returns them to dimension d. Different heads can learn different ways of relating tokens, although no particular interpretation is imposed on an individual head.

Figure 2.2 shows how the operations above are arranged: each head computes attention over its projected queries, keys, and values, then the head outputs are combined.



(a) Scaled dot-product attention

(b) Multi-head attention

Figure 2.2: Attention computation in the Transformer. The diagrams show the weighting of values within one head and the learned projections and combination of multiple heads. Figure from [68].

2.2.3 Transformer Blocks and Attention Patterns

The attention mask determines which information can reach each token. With *causal attention*, token x_i can attend only to itself and earlier tokens. Ignoring padding, attention from x_i to x_j is permitted exactly when $j \leq i$. The representation of token x_i can then be used to predict token x_{i+1} without access to that token. When training on a complete sequence, the mask prevents information from later tokens from entering earlier representations, even though those tokens are processed in the same forward pass.

With *bidirectional attention*, each token can attend to all available tokens when no padding or other restrictions are present. BERT [22] uses bidirectional attention to reconstruct masked tokens from their surrounding text.

Figure 2.3 illustrates this difference in connectivity, its entries indicate which connections are permitted.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

(a) Causal attention

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

(b) Bidirectional attention

Figure 2.3: Permitted attention connections for a sequence of four tokens. Rows correspond to queries and columns to keys. A value of 1 permits attention, while 0 marks a connection whose score is set to $-\infty$ before the softmax. Padding is omitted. Causal attention restricts each row to the current and preceding tokens; bidirectional attention permits access to the entire input.

Attention is followed by a feed-forward network that transforms each token representation separately. A basic form is

$$\mathrm{FFN}(\mathbf{H}) = \phi(\mathbf{H}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2, \tag{2.8}$$

where $\mathbf{W}_1 \in \mathbb{R}^{d \times d_{\mathrm{ff}}}$, $\mathbf{W}_2 \in \mathbb{R}^{d_{\mathrm{ff}} \times d}$, the bias vectors are broadcast over tokens, and ϕ is an element-wise nonlinear activation. The original Transformer uses ReLU. The same network is applied to every token: attention combines information across the sequence, while the feed-forward network transforms the features of each token.

Residual connections add each sublayer input to its output, and normalization controls the scale of the representations. In the original Transformer encoder, each block applies layer normalization after the residual addition for both its self-attention and feed-forward sublayers:

$$\begin{aligned} \widetilde{\mathbf{H}} &= \mathrm{LayerNorm}(\mathbf{H} + \mathrm{MHA}(\mathbf{H})), \\ \mathbf{H}' &= \mathrm{LayerNorm}\left(\widetilde{\mathbf{H}} + \mathrm{FFN}(\widetilde{\mathbf{H}})\right). \end{aligned} \tag{2.9}$$

Layer normalization operates on the features of each token, normalizing them and applying learned scale and offset parameters. Dropout is omitted from Equation 2.9 for clarity. Stacking blocks repeatedly combines information from the permitted tokens and transforms the resulting representations. Other arrangements place normalization before each sublayer; the ordering in this equation describes the original encoder rather than every Transformer implementation.

The original Transformer has an encoder and a decoder. The encoder processes the input with bidirectional self-attention. The decoder uses causal self-attention and an additional *cross-attention*

sublayer, whose queries come from the decoder and whose keys and values come from the encoder output. This allows an output sequence to depend on a separate input sequence, as in translation.

Encoder-only models retain the bidirectional stack, as in BERT, whereas autoregressive decoder-only language models use a causal stack without a separate encoder or encoder-decoder cross-attention. GPT-3 [13] is an example of the latter and uses normalization before its sublayers. In a decoder-only language model, the final representation of token x_i is projected to vocabulary scores, or *logits*, and a softmax produces the distribution used to predict the next token. The output projection connects the continuous representations computed by the network to the discrete token predictions introduced in Section 2.1.

2.2.4 Modern Decoder-Only Models

The original Transformer remains the foundation of large language models, but later systems often change parts of its blocks. For example, RMSNorm [79] scales a representation using its root mean square instead of subtracting its mean as LayerNorm does. SwiGLU [61] replaces the basic feed-forward activation with a gated transformation. These choices change how a block processes information while leaving the autoregressive next-token objective intact.

Rotary positional embeddings (RoPE) [64] incorporate position into the query and key projections. Pairs of query and key coordinates are rotated by angles that depend on their token positions. The dot product between a rotated query and key then depends on their relative positions as well as their content. This lets attention use sequence order without adding positional vectors to the input embeddings. The same principle applies in causal and bidirectional attention: position encoding indicates where a token is, while the attention mask determines which tokens it may attend to.

Grouped-query attention (GQA) [1] lets several query heads share each key and value head. This reduces the key and value states stored during generation, making the KV cache described in Section 2.3.2 smaller than it would be with a separate key and value head for every query head. Another design is mixture-of-experts (MoE): a routing mechanism selects a subset of feed-forward networks for each token, so the model can have more total parameters than it uses for one token computation.

Representative autoregressive families illustrate different combinations of these choices. Llama 3 [29] and Gemma 2 [66] provide dense decoder-only models, while DeepSeek-V3 [44] uses a

sparse MoE design. The Qwen3 family [72] includes both dense and MoE models; the Qwen3-4B and Qwen3-8B baselines evaluated in this thesis are dense members of that family. These families illustrate the range of architectures used in modern autoregressive language modeling.

These architectures are applied beyond general-purpose text generation, including structured graphics [82] and multimodal tasks such as visual understanding [47], image question answering, and retrieval-supported reasoning [16, 21, 48]. Work on vision–language systems also improves cross-modal alignment, limits hallucinated content, and addresses unsafe concepts [15, 20, 53].

2.2.5 Pretraining and Post-Training

Large language models apply these architectures at substantial parameter and data scales. During *pretraining*, the model learns from a large text corpus using a self-supervised objective. GPT-3 provides an example of large-scale autoregressive pretraining and demonstrates how a pretrained model can perform tasks from instructions or examples supplied in its input. Supplying examples in a prompt, often called *in-context learning*, does not itself update the model parameters.

A pretrained model can then undergo *post-training* to adapt its behavior. In supervised instruction tuning, training examples pair a user request with a desired response. The model parameters are updated to increase the probability of those responses, teaching the model to follow instructions in the intended format.

Preference-based training provides another source of supervision by comparing alternative responses. InstructGPT [51] combines supervised fine-tuning with reinforcement learning from human feedback: human rankings train a reward model, and that reward is then used to further optimize the language model. This procedure illustrates how models with similar underlying architectures can differ because of the training stages applied after pretraining.

Preference data can also be used without training a separate reward model. Direct Preference Optimization (DPO) [57] updates a model so that it favors a preferred response over a rejected one. Reasoning-oriented post-training can instead use rewards based on whether an answer can be checked, as illustrated by DeepSeek-R1 [31]. Qwen3 [72] combines instruction tuning and reinforcement learning and supports both responses that include an explicit reasoning process and shorter responses without one.

These distinctions matter when interpreting model comparisons. A difference in downstream

performance can reflect the training corpus, initialization, or post-training procedure as well as the architecture. The architecture determines how information is processed; the training history determines which parameters and behaviors have been learned.

2.3 Autoregressive Language Modeling

An autoregressive language model represents the probability of a sequence through a succession of next-token predictions. Each prediction is conditioned on the tokens that precede it. For the sequence $\mathbf{x} = (x_1, \ldots, x_n)$ introduced in Section 2.1, the chain rule gives

$$p_\theta(\mathbf{x}) = \prod_{i=1}^{n} p_\theta(x_i \mid x_{<i}), \qquad x_{<i} = (x_1, \ldots, x_{i-1}). \tag{2.10}$$

For $i = 1$, the context is empty or contains a designated beginning-of-sequence token, depending on the model. This factorization provides an exact sequence probability when the model supplies a normalized conditional distribution for each token given its preceding context. For a variable-length completed sequence, its terminating end-of-sequence token is also included in the product. Taking the negative logarithm turns the product into a sum of token-level losses. Thus, the corpus objective in Equation 2.1 can be written as

$$\mathcal{L}_{\mathrm{AR}}(\theta; \mathcal{D}) = -\frac{1}{N} \sum_{\mathbf{x} \in \mathcal{D}} \sum_{i=1}^{|\mathbf{x}|} \log p_\theta(x_i \mid x_{<i}), \tag{2.11}$$

where N is the number of scored tokens in the corpus. Tokens excluded from scoring, such as those in a supplied prompt in a conditional evaluation, are omitted from both the inner sum and N.

2.3.1 Training and Left-to-Right Generation

During training, the complete target sequence is already known. The model uses the preceding ground-truth tokens to predict each target token, a procedure called *teacher forcing*. A causal attention mask prevents the prediction from using the target token or subsequent tokens, so the conditional predictions in Equation 2.11 can be computed together in one forward pass through a Transformer [68]. GPT-3 [13] follows this autoregressive training approach at large scale.

Generation follows those dependencies explicitly. Given a prompt $\mathbf{c}$, the model predicts a distribution for the first output token, selects a token y_1, and appends it to the context. The next

prediction then conditions on $(\mathbf{c}, y_1)$, and the process continues:

$$p_\theta(\mathbf{y} \mid \mathbf{c}) = \prod_{i=1}^{L} p_\theta(y_i \mid \mathbf{c}, y_{<i}), \qquad \mathbf{y} = (y_1, \ldots, y_L). \tag{2.12}$$

Here, L counts output tokens and excludes the prompt. Each new token depends on the previously selected outputs. In ordinary autoregressive decoding, the next generation step therefore begins after the current token has been chosen. This execution pattern can be accelerated: speculative decoding [38], for example, verifies several proposed tokens together while preserving the target sampling distribution. Such methods retain the autoregressive model factorization.

The rule for choosing a token from each conditional distribution is the *decoding strategy*. Greedy decoding selects the token with the highest predicted probability. Sampling instead draws a token from a distribution derived from the model probabilities; settings such as temperature or a restricted candidate set can alter that distribution. Generation can stop when the model selects an end-of-sequence token or when a specified output-token limit is reached.

2.3.2 Prefill and Cached Incremental Decoding

A decoder-only Transformer first processes the prompt, often called the *prefill* phase. Each layer forms keys and values for the prompt tokens as part of its self-attention computation. During subsequent *incremental decoding*, a straightforward implementation could run the entire growing sequence through the model again after every token. However, causal attention ensures that representations of earlier tokens do not depend on later tokens. For a fixed prompt and fixed previously generated tokens, their keys and values can therefore be reused.

A *key–value cache*, or *KV cache*, stores those keys and values at each attention layer. At the next step, the model processes the new token to obtain its current query, key, and value. Its query attends to the stored keys and values as well as the new ones; the new key and value are then appended to the cache. This avoids recomputing the earlier tokens' projections and representations through the stack. The cache grows with the processed context and consumes memory, while attention for each new token still depends on the available context length.

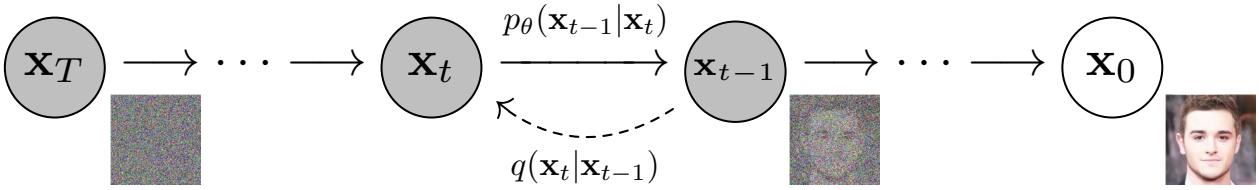


Figure 2.4: Forward corruption q and learned reverse transitions p_θ in a Gaussian diffusion model. The forward process progresses from $\mathbf{x}_0$ to $\mathbf{x}_T$; generation starts at $\mathbf{x}_T$ and follows the reverse process toward $\mathbf{x}_0$. Figure from [33].

2.4 Diffusion as a Generative Paradigm

Autoregressive generation builds a sequence by predicting its next token. Diffusion models instead define a process that progressively corrupts a data point, then learn to reverse that process [62]. The reverse process starts from a simple distribution and produces a sample through a succession of denoising steps. This section introduces the idea for continuous data, where Gaussian noise makes the construction particularly clear. Section 2.5 adapts the same general structure to discrete tokens.

Let $\mathbf{x}_0 \in \mathbb{R}^d$ denote a data point, and let $\mathbf{x}_1, \ldots, \mathbf{x}_T$ be its progressively corrupted versions. The subscript here indexes diffusion steps, rather than positions in a token sequence. A fixed *forward process* q specifies how noise is added. A learned *reverse process* p_θ aims to undo the corruption. The forward process supplies training examples at different noise levels; generation uses the reverse process.

Figure 2.4 shows the two directions in a Gaussian diffusion model. Corruption moves from the clean sample at the right toward noise at the left, and generation follows the learned reverse transitions from left to right.

2.4.1 Gaussian Corruption

In the Gaussian formulation of Ho et al. [33], one forward transition is

$$q(\mathbf{x}_t \mid \mathbf{x}_{t-1}) = \mathcal{N}\left(\mathbf{x}_t; \sqrt{1-\beta_t}\,\mathbf{x}_{t-1}, \beta_t \mathbf{I}\right), \qquad t = 1, \ldots, T, \tag{2.13}$$

where $0 < \beta_t < 1$ controls the amount of noise added at step t and $\mathbf{I}$ is the identity matrix. Each step retains a scaled version of the previous state and adds Gaussian noise. The values $\beta_1, \ldots, \beta_T$

form a *noise schedule*. The forward trajectory is a Markov chain: $\mathbf{x}_t$ depends directly on $\mathbf{x}_{t-1}$, while its earlier history is summarized by that state.

Define $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s$. The state at any step can be sampled directly from the original data point:

$$q(\mathbf{x}_t \mid \mathbf{x}_0) = \mathcal{N}\left(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\,\mathbf{x}_0, (1 - \bar{\alpha}_t)\mathbf{I}\right). \tag{2.14}$$

Equivalently, one may draw $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and form $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\,\mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t}\,\boldsymbol{\epsilon}$. This direct form makes it possible to train on a chosen noise level without simulating every earlier transition. If the schedule makes $\bar{\alpha}_T$ sufficiently small, $\mathbf{x}_T$ is approximately standard Gaussian, with only a small remaining dependence on $\mathbf{x}_0$. That dependence vanishes exactly only when $\bar{\alpha}_T = 0$.

2.4.2 Learning to Reverse the Process

The generative model starts with a simple prior, commonly $p(\mathbf{x}_T) = \mathcal{N}(\mathbf{0}, \mathbf{I})$, and defines transitions toward less corrupted states. Its joint distribution is

$$p_\theta(\mathbf{x}_{0:T}) = p(\mathbf{x}_T) \prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t). \tag{2.15}$$

Here, $\mathbf{x}_{0:T}$ denotes the complete trajectory. In the Gaussian formulation, the learned reverse transitions are typically Gaussian distributions whose parameters are produced by a neural network. To generate a sample, draw $\mathbf{x}_T$ from the prior and apply the learned transitions in the order $T, T-1, \ldots, 1$. An alternative sampler can reduce the number of reverse steps and accelerate generation [63].

The forward trajectory is known, but the reverse transitions that would recover the data distribution are not. Training uses samples from the forward process to fit p_θ. A variational objective connects the forward trajectory and learned reverse model to a lower bound on the data log-likelihood. Intuitively, it rewards reverse transitions that can recover less corrupted states from more corrupted ones, together with agreement between the terminal state and the simple prior.

A commonly used simplified objective [33] trains a network to predict the noise that produced $\mathbf{x}_t$:

$$\mathcal{L}_{\text{noise}}(\theta) = \mathbb{E}_{\mathbf{x}_0, t, \boldsymbol{\epsilon}}\left[\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\|_2^2\right], \tag{2.16}$$

where $\mathbf{x}_0$ comes from the data distribution, t is sampled uniformly from $\{1, \ldots, T\}$, and $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. Here $\mathbf{x}_t$ is sampled using Equation 2.14. The network receives the noisy state $\mathbf{x}_t$ and the timestep

t, which identifies the noise level through the schedule. Its target is the sampled Gaussian noise $\boldsymbol{\epsilon}$ used to construct that state. It is not given the clean sample $\mathbf{x}_0$ or the noise itself as input. This estimate determines the mean of the Gaussian reverse transition, linking the prediction task to the generative process. The unweighted form shown here is a simplified training loss, rather than the complete variational bound.

Image generation is the setting in which diffusion models became established before their adaptation to language. Visual applications also include restoration and super-resolution [23, 55]. Research on image diffusion models examines internal representations, inference-time candidate selection, and safe generation [67, 14, 54].

2.5 Discrete Diffusion

Tokens belong to a finite vocabulary, so adding a real-valued Gaussian perturbation to a token identifier does not directly define a useful corruption process over text. Discrete diffusion instead changes a token from one categorical state to another according to known probabilities [5]. The forward process gradually reduces information about the original sequence, while the learned reverse process uses a corrupted sequence to predict less corrupted states. The overall structure is the same as in Section 2.4; the states and transition rules are different.

2.5.1 Categorical Transitions

Consider a token at a fixed sequence index and let s_t be its state at diffusion step t. The subscript t indicates a noise level, rather than the token position in the sequence. For a state set $\mathcal{S}$ of size K, the forward transition can be represented by a $K \times K$ matrix $\mathbf{Q}_t$:

$$[\mathbf{Q}_t]_{ab} = q(s_t = b \mid s_{t-1} = a), \qquad a, b \in \mathcal{S}. \tag{2.17}$$

Each row is a probability distribution and therefore sums to one. The matrix may vary with t, allowing corruption to become stronger as the process advances. If the same transition is applied independently at each token, the forward probability of a complete corrupted sequence is the product of its per-token transition probabilities. This independence describes the corruption mechanism; a denoising model can still use context from the entire corrupted sequence.

The product of successive transition matrices gives the distribution at an arbitrary noise level without sampling all preceding steps. Writing $\overline{\mathbf{Q}}_t = \mathbf{Q}_1\mathbf{Q}_2 \cdots \mathbf{Q}_t$, we have

$$q(s_t = b \mid s_0 = a) = [\overline{\mathbf{Q}}_t]_{ab}. \tag{2.18}$$

The order of multiplication follows the direction from the clean state s_0 toward s_t. As with the Gaussian marginal in Equation 2.14, direct sampling from this distribution makes it practical to train on randomly selected noise levels.

2.5.2 Uniform and Absorbing Corruption

Two transition choices are particularly useful for understanding diffusion over text. In *uniform corruption* [34], the state set is the vocabulary $\mathcal{V}$, so $K = |\mathcal{V}|$. At step t, a token is retained with probability $1 - \beta_t$ or resampled uniformly from the vocabulary with probability β_t:

$$\mathbf{Q}_t^{\text{uniform}} = (1 - \beta_t)\mathbf{I} + \frac{\beta_t}{K}\mathbf{1}\mathbf{1}^\top. \tag{2.19}$$

Here, $\mathbf{1}$ is a vector of K ones. Uniform resampling can return the original token, so its total probability of remaining unchanged is $1 - \beta_t + \beta_t/K$. For this transition family, the cumulative matrix has the same form:

$$\overline{\mathbf{Q}}_t^{\text{uniform}} = \overline{\alpha}_t\mathbf{I} + \frac{1 - \overline{\alpha}_t}{K}\mathbf{1}\mathbf{1}^\top, \qquad \overline{\alpha}_t = \prod_{r=1}^{t}(1 - \beta_r). \tag{2.20}$$

As $\overline{\alpha}_t$ becomes small, the state approaches a uniform categorical distribution, making it harder to infer the original token.

In *absorbing-state corruption* [5], the state set also contains a special [MASK] symbol. Let $\mathbf{e}_m$ be the one-hot vector for that symbol, and let K now denote the size of this extended state set. A token either stays as it is or changes to [MASK]:

$$\mathbf{Q}_t^{\text{mask}} = (1 - \beta_t)\mathbf{I} + \beta_t\mathbf{1}\mathbf{e}_m^\top. \tag{2.21}$$

Once a token is masked, later forward steps leave it masked; this is why the mask is called an absorbing state. With sufficient corruption, the terminal distribution concentrates on [MASK] rather than on uniformly sampled vocabulary tokens. Uniform and absorbing processes therefore present different information to the denoising model: a uniformly replaced token appears to be an ordinary token, while a mask explicitly identifies a token whose original value is hidden.

a)	T = 0	The great brown fox hopped over the lazy dog.
	T = 10	The great [MASK] fox hopped over [MASK] lazy dog.
	T = 20	The [MASK][MASK] [MASK] ship over [MASK] lazy the.
	T = 25	[MASK] [MASK] [MASK] [MASK] [MASK] [MASK] [MASK] [MASK] [MASK]
b)	T = 0	The great brown fox hopped over the lazy dog.
	T = 10	The vast black fox hopping over the lazy cat.
	T = 20	Their vast tripped this jumping upon walked organizations.
	T = 25	Bunk scamper tripped this Sanchez walked organizations.

Figure 2.5: Two illustrative text corruption trajectories: (a) absorbing-state masking and (b) uniform corruption. The trajectories illustrate their contrasting effects rather than the exact transition rules defined above. Figure from [5].

Figure 2.5 illustrates how a text sequence changes under two discrete corruption rules. The upper example uses masking, while the lower example replaces words with other visible words through uniform sampling. In both cases, the rows show successive forward corruption states, not generated text.

2.5.3 Reverse Transitions and Learning

For a corrupted sequence $\mathbf{s}_t$, a denoising network defines a reverse distribution $p_\theta(\mathbf{s}_{t-1} \mid \mathbf{s}_t)$. Starting from a sample drawn from a simple terminal prior $p(\mathbf{s}_T)$, generation repeatedly samples or selects a less corrupted sequence. The generative trajectory factorizes as

$$p_\theta(\mathbf{s}_{0:T}) = p(\mathbf{s}_T) \prod_{t=1}^{T} p_\theta(\mathbf{s}_{t-1} \mid \mathbf{s}_t). \tag{2.22}$$

The appropriate terminal prior follows the corruption scheme: it may be uniform over states or concentrated on mask symbols. A finite schedule only approximates that prior unless its final transition reaches it exactly.

Training uses the known corruption process to construct targets for the reverse model [5]. Given both a clean token and its corrupted state, one can calculate the probabilities of possible intermediate

states. These conditional probabilities provide a training target even though the unconditional reverse process is unknown. A variational objective encourages the learned reverse transitions to agree with those targets and bounds the data log-likelihood from below.

An equivalent implementation can predict the original clean token and use that prediction, together with the known corruption rule, to form a reverse transition. This separates two roles: the network predicts token content from context, while the transition rule controls how much noise is removed at a step. The absorbing-state case makes this connection particularly simple and leads to the masked reconstruction objective introduced next.

2.6 Masked Diffusion for Language

Masked diffusion applies the absorbing-state process from Section 2.5.2 to token sequences. At a selected noise level, some tokens are replaced by [MASK] while the others remain visible. A denoising network receives the entire corrupted sequence and predicts the original values of the masked tokens. The denoiser can use the Transformer stack from Section 2.2 with bidirectional attention and a vocabulary projection for each token. Visible tokens on either side of a mask provide context, and one forward pass produces predictions for all masked tokens.

This reconstruction task resembles masked language modeling, but generation requires a defined route from an initially masked sequence to a completed one. A masked diffusion model supplies that route through a sequence of noise levels and reverse transitions. The formulation below predicts clean tokens and uses them to define reverse unmasking transitions.

Figure 2.6 illustrates these stages in LLaDA [49]: random masking during pretraining, masking of response tokens during supervised fine-tuning, and iterative sampling from an initially masked response. The figure shows one model procedure; the mechanisms below explain the general masked diffusion formulation used in this chapter.

2.6.1 Training with Masked Sequences

Masked diffusion training samples a noise level and corrupts a clean sequence accordingly. In LLaDA [49], a continuous timestep $t \sim \mathcal{U}(0, 1)$ is sampled, and each token is independently replaced by [MASK] with probability t. Unlike the discrete step index used in the preceding

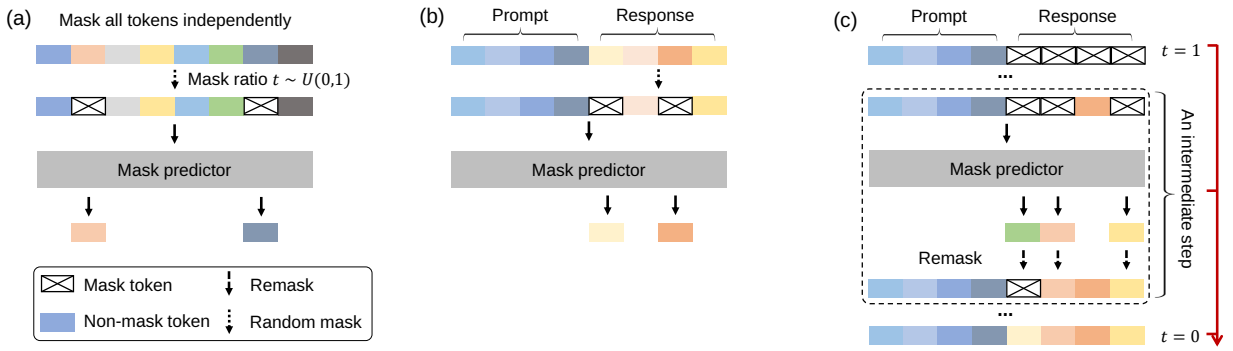


Figure 2.6: Training and sampling in LLaDA. Panels show random masking for pretraining, response masking for supervised fine-tuning, and repeated prediction and remasking during generation. Figure from [49].

sections, t here directly specifies the masking probability. Small values leave most of the sequence visible, while values close to one mask most of it.

Let $\mathbf{x} = (x_1, \ldots, x_n)$ be a clean training sequence and $\widetilde{\mathbf{x}}_t$ its masked version. The model predicts the original values of the masked tokens using the remaining text as context. LLaDA denoiser takes the corrupted sequence as input without requiring the timestep explicitly.

For absorbing diffusion with a reverse parameterization that carries visible tokens forward unchanged and excludes mask symbols from clean-token predictions, the variational objective simplifies to a weighted reconstruction loss [58]. With the masking procedure above, this takes the form

$$\mathcal{L}_{\text{mask}}(\theta) = \mathbb{E}_{\mathbf{x}, t, \widetilde{\mathbf{x}}_t} \left[\frac{1}{t} \sum_{i=1}^{n} \mathbf{1}\{\widetilde{x}_{t,i} = [\text{MASK}]\} \left[-\log p_\theta(x_i \mid \widetilde{\mathbf{x}}_t) \right] \right]. \tag{2.23}$$

Here, $\widetilde{x}_{t,i}$ is the i-th token of the corrupted sequence. The indicator restricts the loss to masked tokens, and $1/t$ is the weighting prescribed by the variational objective. Sampling different timesteps trains the model to reconstruct text from contexts with different amounts of missing information.

2.6.2 Conditional Generation and Denoising Steps

To generate a response to a prompt $\mathbf{c}$, the model places L mask tokens after the prompt and gradually replaces them with output tokens. Throughout this thesis, *generation length* is the configured number L of output tokens; prompt tokens are excluded. The prompt stays fixed while the model denoises the response. LLaDA [49] is one example of this approach.

Generation uses S denoising steps. At each step, the model predicts tokens for the remaining masks. The sampler chooses which predictions to keep and which tokens to leave masked for another step. In monotonic unmasking, retained tokens stay fixed: a later step adds context by generating other tokens. Revising a retained token requires a separate editing or remasking rule.

Let u_s be the number of new tokens revealed at step s. If each token is revealed once and never masked again,

$$\sum_{s=1}^{S} u_s = L, \qquad \frac{L}{S} = \text{average number of tokens revealed per step.} \tag{2.24}$$

With $S = L$, a schedule can reveal one token at each step. If $S < L$, some steps must reveal several tokens. This is the source of parallel generation in the experiments: fewer denoising steps mean more tokens must be produced together.

Tokens chosen in the same step are predicted from the same current context. They cannot use one another as context until the next step. When more tokens are generated together, this can reduce quality on tasks where tokens depend strongly on one another. Additional steps give the model more chances to use its earlier predictions when generating the remaining tokens [69].

A common selection heuristic ranks candidate tokens by the model confidence in its predictions. Committing high-confidence candidates first allows later steps to condition on them. LLaDA [49], for example, remasks low-confidence proposals for later prediction. The selection rule determines which predictions become context for later steps. The configured generation length specifies the number of output tokens; an end-of-sequence convention or stopping rule may lead to a shorter visible response in a particular implementation.

Unlike the autoregressive factorization in Equation 2.10, the probability of a completed output generally requires summing over unobserved denoising trajectories. Exact likelihood evaluation is therefore usually intractable for diffusion language models. A practical approach is to use a variational bound and estimate its expectation by Monte Carlo sampling, as in LLaDA [49]. This estimates the bound, which need not equal the exact log-likelihood. The quantity used for scoring or reporting perplexity must therefore be specified; the evaluation procedure is described in Chapter 4.

2.7 Block Diffusion

Full-sequence masked diffusion treats all output tokens within one denoising process. Block diffusion [2] instead divides the output into consecutive blocks and generates those blocks from left to right. Within the active block, tokens are still produced through iterative denoising. This combines sequential dependence between blocks with the possibility of updating several tokens in a block during one denoising step.

Let $\mathbf{c}$ be the prompt and let $\mathbf{y} = (\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(B)})$ be the output partitioned into B ordered blocks. The resulting conditional distribution can be written as

$$p_\theta(\mathbf{y} \mid \mathbf{c}) = \prod_{b=1}^{B} p_\theta\left(\mathbf{y}^{(b)} \mid \mathbf{c}, \mathbf{y}^{(<b)}\right), \tag{2.25}$$

where $\mathbf{y}^{(<b)}$ denotes the completed earlier blocks. Each block-conditional term represents a generative process over the tokens in that block; it need not factorize autoregressively over individual tokens. Its likelihood may still involve unobserved denoising trajectories, as in Section 2.6.

This factorization describes a model whose conditional distributions are defined at the block level. A block generation schedule can also be applied to a denoiser trained over full sequences, by restricting which output tokens are generated at each stage. Such a schedule does not itself change the denoiser attention pattern or guarantee that only the active block is processed. The distinction matters when comparing different sampling configurations of the same trained model.

2.7.1 Generation Within and Between Blocks

At the start of block b, its output tokens are initialized as [MASK]. The prompt and completed blocks provide context, while the model repeatedly predicts tokens in the active block. An inference rule decides which predictions to retain or remask before the next step. Once the block is complete, generation advances to block $b + 1$. Thus, a later block can depend on all earlier output blocks, but tokens within one block can be predicted in parallel.

Only the active block is generated at each stage. Its update and remasking rules determine whether selected tokens remain fixed or can be revised before generation advances to the next block.

2.7.2 Block Size, Step Budget, and Reuse

For generation length L and nominal block size L_{block}, the number of blocks is $B = \lceil L/L_{\text{block}} \rceil$; the final block may be shorter. Let S_b be the denoising steps used for block b. If blocks are generated sequentially and each step requires one model call, the number of denoising calls is

$$S_{\text{total}} = \sum_{b=1}^{B} S_b. \tag{2.26}$$

For a uniform allocation of S_{block} steps per block, this becomes BS_{block}. Some configurations instead specify a total step budget and distribute it across blocks, so a reported "number of steps" must be interpreted with its schedule. Under a monotonic schedule that commits exactly one new token per step, S_b equals the length of block b and $S_{\text{total}} = L$. Extra forward passes may be needed to prepare or update cached context; for example, the block diffusion sampler of Arriola et al. [2] processes a completed block to obtain the keys and values used by later blocks.

Block size changes both the scope of parallel prediction and the number of sequential block transitions. Larger blocks offer more tokens that could be updated together. Smaller blocks impose more left-to-right dependencies and allow each new block to use a completed prefix. When computation is restricted to the active block and cached context, smaller blocks also reduce the number of tokens whose representations must be recomputed on each denoising pass.

Exact reuse requires the completed prefix representations to remain unchanged. A block-causal attention pattern supports this by preventing earlier blocks from attending to the active or future blocks, while allowing bidirectional attention within each block, as illustrated in Figure 2.7. With fixed prefix inputs and conditioning, the cached keys and values remain valid as later blocks change. The active block is recomputed as its tokens are updated.

For a fully bidirectional denoiser, fixing earlier token identities is insufficient: their representations can still change when the active block changes. Reusing those states without recomputation is then an approximation. Fast-dLLM [69] uses approximate reuse of this kind. A block sampling schedule and a reusable prefix are therefore distinct properties.

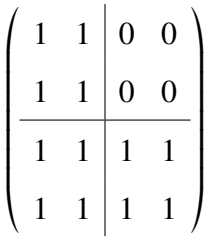
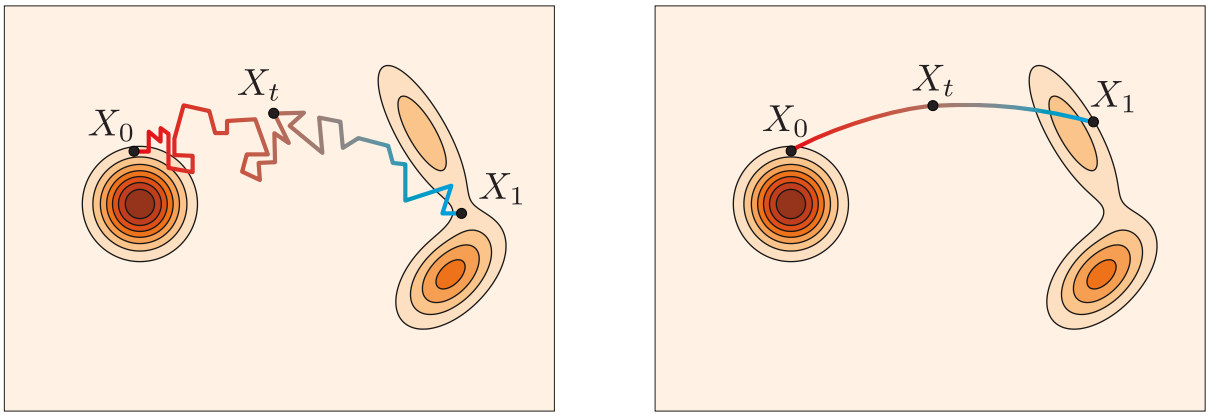


Figure 2.7: Block-causal attention for four tokens arranged in two blocks of two. Rows represent queries and columns represent keys, following Figure 2.3; the prompt is omitted. Each block has bidirectional access internally and can attend to earlier blocks. The first block cannot attend to the second, so changes in the latter do not affect the first block representations.

2.7.3 Computational Implications

The generation schedules above differ in both the number of model calls and the work performed by each call. A full-sequence denoiser can process the prompt and all output tokens, including masks, repeatedly, even when committing only one token per step. After prefill, cached autoregressive decoding computes a new token representations while attending over stored context. Block generation can reduce repeated processing when the implementation combines an active block with reusable prefix states. Equal denoising-step counts therefore need not imply equal floating-point operation counts (FLOPs).

Memory requirements reflect another aspect of inference. In addition to model parameters, the device stores intermediate tensors and any caches retained between calls. Caching saves recomputation but requires storage that grows with the retained context. Peak memory also depends on batch size, numerical precision, and how attention is implemented. Operation counts and peak memory describe computational requirements; neither directly specifies elapsed generation time. Chapter 4 defines the measurements used in this thesis.



(a) Diffusion path.

(b) Flow path.

Figure 2.8: Illustrative paths between a source and a target distribution. The diffusion example follows a stochastic trajectory, whereas the flow example follows a trajectory defined by a velocity field. Figure from [43].

2.8 Flow Matching as a Generative Paradigm

The diffusion mechanisms above supply the foundations for the experiments. Figure 2.8 illustrates one path through each paradigm. Flow matching is included here to explain a related direction in language generation that appears in the literature survey. It provides another way to learn a path from a simple distribution to data [42]. Instead of learning reverse transitions for a prescribed corruption process, a continuous flow model learns a velocity field: at each time and state, it predicts a direction in which the state should move. Generation starts from a simple sample and follows that field toward the data distribution. Diffusion and flow matching are closely related families of generative methods, and some probability paths can be described in both frameworks. The distinction here concerns what the model is trained to predict and how its generative path is followed.

One simple example joins a continuous noise sample $\mathbf{z}^{\text{noise}}$ to a continuous data sample $\mathbf{z}^{\text{data}}$ by a straight path:

$$\mathbf{y}_t = (1-t)\mathbf{z}^{\text{noise}} + t\mathbf{z}^{\text{data}}, \qquad 0 \le t \le 1. \tag{2.27}$$

The velocity along this particular path is $\mathbf{z}^{\text{data}} - \mathbf{z}^{\text{noise}}$. A network $\mathbf{v}_\theta(\mathbf{y}_t, t)$ is trained on many noise–data pairs by minimizing, for example,

$$\mathcal{L}_{\text{FM}}(\theta) = \mathbb{E}_{\mathbf{z}^{\text{noise}}, \mathbf{z}^{\text{data}}, t}\left[\left\|\mathbf{v}_\theta(\mathbf{y}_t, t) - (\mathbf{z}^{\text{data}} - \mathbf{z}^{\text{noise}})\right\|_2^2\right]. \tag{2.28}$$

In this example, the noise and data samples are drawn independently, and t is uniform on $[0, 1]$. At a given state and time, different training pairs can supply different target velocities. Minimizing the squared error learns their conditional average, which defines the velocity field used for generation. At generation time, starting from noise, an ordinary differential equation follows the learned field toward a sample:

$$\frac{\mathrm{d}\mathbf{y}_t}{\mathrm{d}t} = \mathbf{v}_\theta(\mathbf{y}_t, t). \tag{2.29}$$

The straight path is illustrative; flow matching also allows other paths. The learned sampling trajectory need not be the straight line of any one training pair. A numerical solver approximates the differential equation with a finite number of updates, so the number of sampling steps is an inference choice here as well.

A *flow map* [11] takes a state at time s directly to its state at a later time t. The velocity field describes local change, while the map captures the change over an interval. A learned flow map can therefore replace many numerical solver updates with one or a few model calls. It can be learned by distilling a trained flow model, or by self-distillation, which trains the map to make a direct update agree with a sequence of shorter updates without a separately trained teacher. This offers a route to few-step text generation when the flow acts on continuous representations of tokens.

Text introduces the same issue encountered by continuous diffusion: tokens are discrete, while Equation 2.29 acts on continuous states. A flow-based language model may therefore work with token embeddings or another continuous representation and map back to tokens at the end. Discrete flow matching [24] takes a different route. It defines a path of probability distributions over token states and learns how probability moves between those states. Its sampling process changes tokens through discrete transitions rather than treating token identifiers as real-valued coordinates. Chapter 3 surveys representative language models built from both routes.

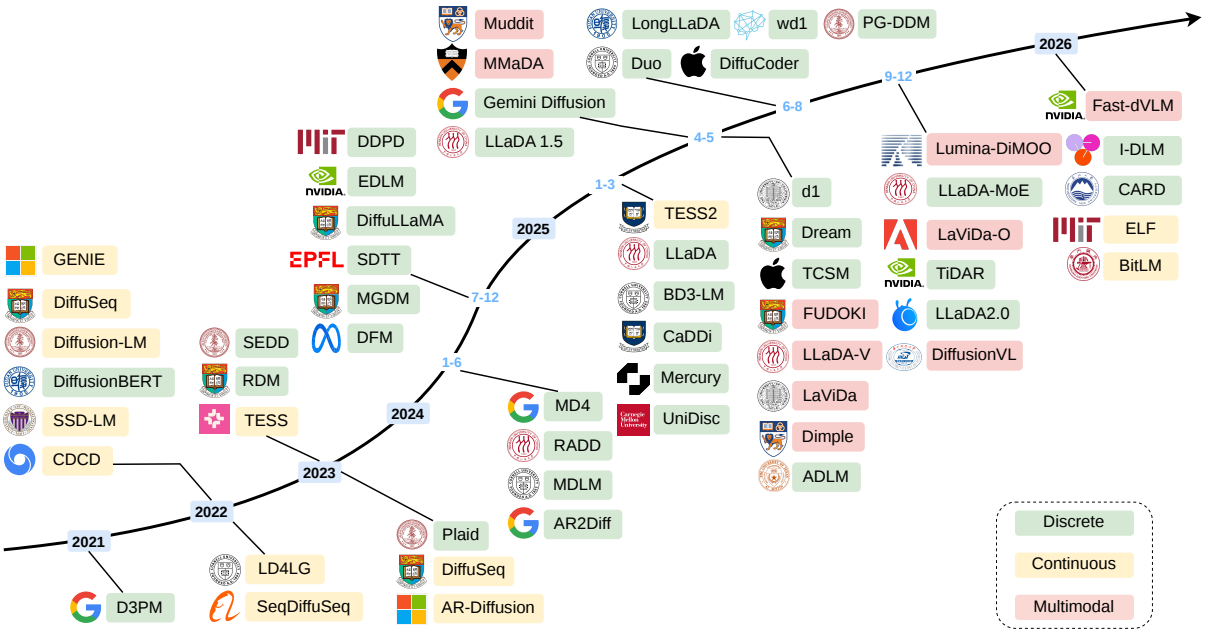


Figure 3.1: Selected diffusion language models in chronological order, grouped by discrete, continuous, and multimodal approaches. Figure from [40].

3. Diffusion Language Models and Related Work

Chapter 2 introduced the mechanisms shared by autoregressive and diffusion language models. This chapter follows the development of diffusion approaches to text: where corruption takes place, how denoising is conditioned, and how models organize generation over a sequence. It also places the evaluated models alongside newer text, flow-based, and multimodal work. These additional models provide context for the field. The experimental contribution is the evaluation of the selected text-only diffusion models in Chapters 4 and 5. Figure 3.1 places selected contributions along this development and distinguishes discrete, continuous, and multimodal work.

Three questions guide the survey. First, does corruption act on continuous vectors or discrete tokens? Second, is the model trained from scratch or adapted from an autoregressive checkpoint? Third, does generation denoise a complete sequence or proceed through blocks? These choices can be combined in different ways. Keeping them separate helps explain why two models called diffusion language models may have very different training histories and computational requirements.

3.1 From Continuous to Discrete Language Diffusion

The Gaussian process in Section 2.4 acts on continuous vectors, while text is represented by discrete tokens. Early diffusion language models connected these spaces by assigning vectors to tokens and applying noise in the resulting continuous representation. Generation then required a way to turn the denoised vectors into a token sequence. This design made methods developed for continuous diffusion available for text, while introducing decisions about both the representation and the final conversion back to words.

Diffusion-LM [41] maps tokens to word vectors and learns to denoise a sequence of continuous states before converting it back to text. Its continuous intermediate states also support gradient-based guidance during generation: a control signal can steer the denoising trajectory toward a requested property, such as a syntactic structure. The work therefore illustrates a distinctive benefit of the continuous space, beyond simply adapting Gaussian noise to language. At the same time, its embedding and conversion steps are necessary parts of the language model, since a continuous output is not itself a sequence of tokens.

DiffuSeq [27] develops a continuous diffusion approach for conditional sequence-to-sequence generation. It denoises a target sequence in the presence of an observed input sequence, placing the emphasis on tasks where an output must respond to or transform a supplied text. Whereas Diffusion-LM demonstrates how a continuous trajectory can be guided toward a chosen property, DiffuSeq incorporates the source sequence into the conditional generation task. Both approaches must ultimately map their continuous representations back to tokens.

Plaid [30] is another continuous diffusion language model, with an emphasis on likelihood-based training and evaluation. Its authors study how model size and training compute affect likelihood, then train a billion-parameter model. Plaid shows that continuous diffusion can be studied as a language model in its own right, alongside work focused on controlled generation.

The early continuous approaches also exposed a difficulty. Text is made of discrete tokens, and valid token embeddings occupy only a small part of the continuous space. A denoised vector may lie between token embeddings, making the final conversion back to text difficult [41]. Discrete diffusion avoids this conversion by corrupting tokens directly. Categorical replacement and absorbing masks, introduced in Section 2.5, are two examples. The model then predicts tokens during denoising.

The following sections examine how diffusion language models developed their objectives and sampling procedures, and how later systems scaled or reorganized those mechanisms.

3.2 Discrete Diffusion Language Models

In discrete diffusion, the forward process changes token states rather than continuous representations. The choice of transition determines what information survives at each noise level and what the reverse model must recover. It also affects how a denoising model can be trained. The distinction introduced in Section 2.5.2 provides a way to organize the main approaches: some replace tokens with ordinary vocabulary items, while others replace corrupted tokens with an absorbing mask.

D3PM [5] provides a general framework for defining categorical forward transitions through matrices. Its experiments include transition rules with different structures, including absorbing-state masking. The framework makes the corruption rule an explicit modeling choice rather than assuming one fixed kind of noise. Multinomial Diffusion [34] explores uniform categorical replacement for language modeling, in which a corrupted token can still be an ordinary vocabulary item. These works establish ways to corrupt tokens that later language models build on. The corruption rule does not by itself determine how the model is trained or how it generates text.

SEDD [46] develops a different way to learn the reverse process. Instead of predicting clean tokens with the weighted masked reconstruction objective in Equation 2.23, it learns ratios between the probabilities of discrete states using a score-entropy loss. The method is evaluated with both absorbing and uniform corruption. It illustrates that a discrete diffusion model need not be a masked language model, and that the corruption process and the learned quantity are separate design decisions.

MDLM [58] focuses on absorbing-state corruption. Its reverse model predicts the original values of masked tokens. This leads to a weighted reconstruction loss derived from a bound on likelihood, giving masked reconstruction a role in generative training. MDLM also studies sampling procedures that can generate in segments, showing that the training objective does not mandate a single full-sequence sampling schedule.

Duo [59] returns to uniform-state corruption. It exploits a relationship between uniform categorical diffusion and an underlying Gaussian process to develop a training curriculum and methods for

generation with fewer steps. A uniformly corrupted token can change value as denoising continues, allowing the process to revise a previous value. This behavior differs from monotonic unmasking, where a committed token remains fixed. Duo therefore provides a useful counterpoint to MDLM in the small-model comparison.

Taken together, these models show why "discrete diffusion" is a broader category than masked diffusion. Both the corruption rule and the quantity predicted by the network matter, while sampling can add another layer of variation. Later large diffusion language models predominantly use absorbing masks, but differ in their initialization, training scale, and post-training. The next section examines those differences.

3.3 Scaling Diffusion Language Models

The earlier models established ways to learn and sample from discrete diffusion processes, but did not by themselves show how masked diffusion would behave at the scale of contemporary language models. Scaling raises two related questions: whether a diffusion objective can support the capabilities associated with large-scale pretraining, and whether such a model should be trained from scratch or adapted from an existing autoregressive model. Both routes produce models that generate tokens through denoising, but they start from different training conditions.

Scaling up Masked Diffusion Models on Text [50] studies how the masked diffusion objective responds to increased model size and training compute. Its experiments extend masked diffusion to the billion-parameter range and compare it with autoregressive models on language tasks. The work supplies evidence that masked diffusion is not confined to small-scale demonstrations, while also making clear that model quality must be considered alongside the cost of iterative sampling. It provides a bridge from the objectives in the previous section to the later large-model systems.

LLaDA [49] follows the training-from-scratch route. It uses random masking for corruption and a Transformer to reconstruct masked tokens, then applies supervised fine-tuning to obtain an instruction-following model. It shows that the masked diffusion objective can be used for both large-scale pretraining and post-training. The resulting model also retains the ability to condition on visible tokens on either side of a missing span; generation need not be restricted to a left-to-right order.

Another route starts from an autoregressive model instead of training a diffusion model from scratch. DiffuGPT and DiffuLLaMA [28] are obtained by continuing to train GPT-2 and LLaMA checkpoints with a diffusion objective. This reuses knowledge from autoregressive pretraining and reduces the training needed to build a large diffusion model.

Dream [75] also starts from an autoregressive checkpoint. It adapts the model to masked diffusion and trains it at the 7B scale. Dream is therefore an example of how a strong autoregressive initialization can support a full-sequence diffusion model. Its performance reflects both the initial checkpoint and the subsequent diffusion training, a distinction that matters when comparing it with LLaDA training-from-scratch route.

Post-training introduces another source of variation. LLaDA 1.5 [80] extends LLaDA with preference optimization. The authors address the variance of likelihood estimates used for preference-based training under a masked diffusion objective. Their method illustrates that diffusion models can be adapted after supervised fine-tuning, while requiring care because their likelihood terms are estimated rather than obtained from a direct autoregressive factorization. The details of likelihood evaluation were introduced in Section 2.6; here the relevant point is that the training objective, initialization, and post-training stage can all change independently of the basic masking mechanism.

Consequently, the label "large diffusion language model" does not describe a single training recipe. LLaDA, Dream, and LLaDA 1.5 provide examples of models shaped by different combinations of pretraining, adaptation, and post-training. Chapter 4 describes the evaluated variants. Before that, the next section considers another independent design choice: whether denoising acts on the full generated sequence or proceeds through smaller blocks.

3.4 Block and Hybrid Diffusion Models

Full-sequence denoising repeatedly processes the output tokens, even if a sampling step resolves only a few of them. Block diffusion changes the scope of this process: completed blocks provide context for later blocks, while tokens within the current block are denoised together. Section 2.7 introduced the factorization and step accounting. The models surveyed here differ in whether they train for that factorization, apply a block schedule at inference, or reorganize the computation around the denoising process.

Block Discrete Denoising Diffusion Language Models (BD3-LMs) [2] define an autoregressive distribution over blocks and a diffusion model within each block. Their block-causal architecture permits reuse of a completed prefix while retaining bidirectional interactions inside the active block. The method therefore makes block structure part of the model itself, rather than treating block size solely as a decoding parameter. Its training and sampling procedures target both flexible output length and the cost of repeated denoising.

Fast-dLLM [69] approaches efficiency from the inference side. It applies blockwise decoding to existing large diffusion models, reusing past key and value states approximately and selecting multiple confident predictions when possible. Because the original denoiser was not trained with block-causal attention, fixing earlier tokens does not make their hidden states independent of the active block. Its cache therefore represents an inference approximation, not the exact reuse available under a suitable block-causal architecture. This distinction is relevant when comparing a trained block model with a full-sequence model run under a block schedule.

Fast-dLLM v2 [70] instead adapts an autoregressive model for block diffusion through additional training. Its attention pattern supports bidirectional prediction within a block while preserving reusable context across blocks. A hierarchical decoding scheme subdivides each block and caches computations within partially generated blocks. It thus combines the adaptation route discussed in Section 3.3 with a block-oriented architecture. The distinction between Fast-dLLM and Fast-dLLM v2 is substantive: the former changes the inference procedure of an existing diffusion model, whereas the latter produces a model trained for block generation.

Figure 3.2 shows the inference procedure of Fast-dLLM v2 [70]. Completed blocks supply cached context, while the active block is refined through smaller sub-blocks. The diagram describes the model decoding procedure; the evaluation in Chapter 4 varies its block size without imposing one denoising step per output token.

These examples show three ways to use blocks. BD3-LM trains with block structure, Fast-dLLM applies a block schedule to an existing model, and Fast-dLLM v2 trains an adapted model for block generation. Their different attention patterns determine how earlier blocks can be reused during generation.

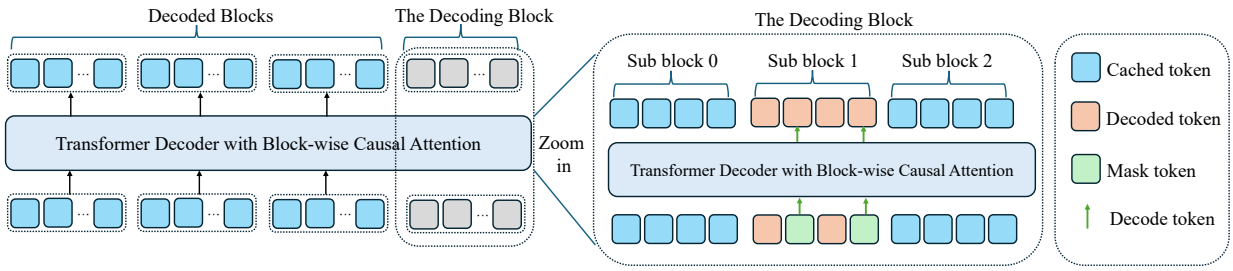


Figure 3.2: Block decoding in Fast-dLLM v2. Decoded blocks are cached as context, while decoding and cache reuse continue within the active block. Figure from [70].

Fixed blocks are not the only way to combine left-to-right progress with parallel prediction. SDLM [45] predicts a window of future tokens and advances by a variable-length contiguous prefix selected from that window. This pseudo-autoregressive strategy can adjust how far generation moves at each step rather than committing a predetermined block in full.

Another route changes the architecture rather than the ordering of outputs. Efficient Encoder-Decoder Diffusion (E2D2) [3] separates representation of clean context from iterative denoising: an encoder processes observed tokens and a lighter decoder repeatedly updates corrupted tokens. The work also applies this separation to block diffusion. This addresses the repeated cost of processing clean context without requiring every diffusion model to share the same block schedule or attention pattern.

These systems illustrate that generation order, learned factorization, and reuse of computation are related but separate choices. A block schedule alone does not establish that cached context is exact or that the model was trained blockwise. The evaluation in this thesis therefore describes both the model and its inference configuration before comparing quality and computational cost.

3.5 Further Developments and Related Approaches

The following work extends the survey beyond the evaluated models. It provides context for current research on scaling, decoding, and new modalities, without extending the experimental scope of the thesis.

3.5.1 Scaling and Decoding

DiffuCoder [26] trains a masked diffusion model specifically for code. It studies how the order in which code tokens are generated changes with decoding settings and uses reinforcement learning to improve code generation. This is a task-focused extension of the general text models discussed earlier.

LLaDA-MoE [81] combines masked diffusion with a sparse mixture-of-experts architecture and trains the model from scratch. Only a subset of its experts is active for each token.

LLaDA 2.0 [9] demonstrates that diffusion language models can scale to 100B total parameters, extending substantially beyond the 7–8B scale of LLaDA and Dream. It converts pretrained autoregressive MoE models through a staged training procedure that moves between block and full-sequence diffusion. The resulting instruction-tuned models contain 16B and 100B total parameters. This makes large-scale adaptation an important complement to training diffusion models from scratch.

LLaDA 2.1 [10] builds on LLaDA 2.0 by allowing the sampler to edit a token that has already been generated, as well as fill masked tokens.

SDAR [18] adapts pretrained autoregressive models for blockwise diffusion, aiming to retain their learned capabilities while enabling parallel prediction within a block.

I-DLM [77] follows this line of work but changes the attention pattern and decoding procedure to check previously generated tokens while producing new ones. Its consistency checks address whether earlier predictions remain acceptable as generation continues.

DiffusionGemma [65] also adapts an autoregressive MoE model and generates text by refining blocks of tokens. These models continue the block-oriented line of work, while using different training and decoding procedures from the block models evaluated here. DiffusionGemma also retains multimodal input support, connecting text generation with the extensions discussed next.

3.5.2 Multimodal Extensions

Diffusion language models can also be connected to visual inputs. One common design encodes an image and gives the resulting features to a diffusion language model, which then generates text conditioned on the image. LaViDa [39] and LLaDA-V [76] follow this broad pattern for vision-

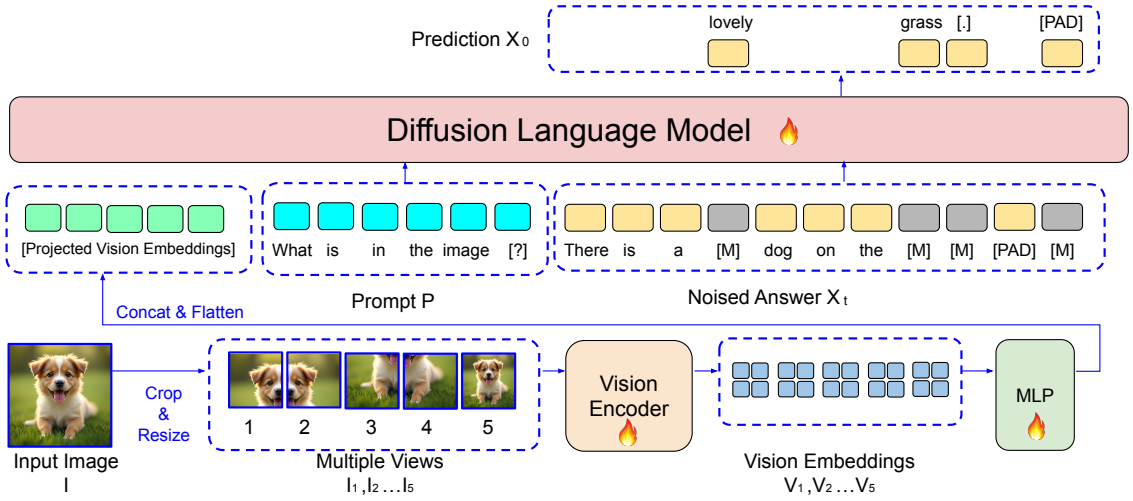


Figure 3.3: Image-conditioned text generation in LaViDa. A vision encoder and projector provide visual context to a diffusion language model, which predicts masked response tokens. Figure from [39].

language understanding. They show how masked text generation can be used for image questions and related tasks. Figure 3.3 shows the image-conditioned text-generation path in LaViDa [39]: image features join the prompt as context for a diffusion language model that predicts masked answer tokens.

MMaDA [73] takes a broader approach: its unified diffusion model handles text, visual understanding, and image generation. Fast-dVLM [71] instead focuses on efficient text generation from visual input by converting an autoregressive vision-language model to block diffusion. The distinction between these systems concerns what is generated: a model can generate only text conditioned on an image, or support generation in more than one modality. In either case, multimodal input changes the source of conditioning information; it does not by itself specify whether text is produced autoregressively or by denoising.

3.5.3 Flow-Based Language Models

Section 2.8 introduced flow matching as another generative framework. The following language models show two ways to apply it to text: learning transitions between discrete token states or following a path through a continuous representation. The terminology overlaps with diffusion in some of these works; the useful distinction is the probability path, training target, and sampling

procedure, rather than a strict division into unrelated model families.

Discrete Flow Matching [24] defines paths of distributions over tokens and learns how to move between token states along those paths. Few-Step Discrete Flow Matching (FS-DFM) [36] builds on discrete flows with training designed for generation using a small number of steps. Both work directly with discrete states, but focus on different sampling regimes.

Flow Map Language Models [37] instead learn a map through continuous states whose endpoints represent one-hot tokens. The map can generate text in very few steps. ELF [35] works in a continuous token-embedding space and converts its result to tokens at the end. These models illustrate why a flow-based text model must specify both its representation of language and how that representation becomes a token sequence. They also revisit the representation question at the start of this chapter. Continuous representations remain an active research direction, even though the models evaluated in this thesis operate directly on discrete tokens.

3.6 Evaluation in Existing Literature

The papers discussed above emphasize different evaluation goals. LLaDA [49] examines scaling and downstream capabilities, BD3-LM [2] studies language modeling and block generation, and Fast-dLLM [69] investigates inference efficiency. A comparison across these systems therefore benefits from a shared set of tasks and an explicit account of generation settings.

The survey identifies two particularly useful comparisons for this thesis. Dream and LLaDA both support full-sequence masked generation, but reach it through different training routes. Comparing their responses to the same inference changes reveals whether those changes have similar effects across models. LLaDA and LLaDA 1.5 can also be examined under block schedules alongside Fast-dLLM v2, whose training and attention pattern are designed for block generation. This connects the distinction between an inference schedule and a trained block architecture to the experimental study.

The evaluation that follows combines downstream benchmarks with inference sweeps and computational profiling. A separate comparison of smaller models sharing an OpenWebText training-corpus basis examines perplexity. Chapter 4 defines the protocols for these comparisons.

4. Experimental Methodology

The preceding chapters established the modeling choices that distinguish autoregressive, full-sequence diffusion, and block-oriented generation. This chapter specifies the experimental design of *Diffusion Language Models: An Experimental Analysis* [7], the study on which this thesis is based. It first sets out the evaluation design, then describes the models and benchmarks, explains the scoring protocol, and documents the inference and computational-cost measurements. The final section describes the complementary comparison of smaller models sharing a training corpus.

4.1 Evaluation Design

The design follows the three research questions introduced in Section 1.3: performance across language tasks (RQ1), sensitivity to inference settings (RQ2), and memory and estimated computation during generation (RQ3).

To answer RQ1, the large models are evaluated on the same eight benchmarks. Each model score on each benchmark is its best result among the inference settings tested for that model. This shows the performance reached during the study. RQ2 is addressed by changing generation length, denoising steps, or block size and observing the resulting scores. RQ3 uses separate profiling runs to measure peak GPU memory and estimate FLOPs during generation.

The headline scores and profiling results serve different purposes. The former summarize the best task performance observed in the tested configurations; the latter describe a specified generation workload. They are not measurements of quality and cost at a single common operating point. The inference sweeps connect the two questions by showing how the generation schedule affects performance.

The study also compares smaller models trained on OpenWebText. This part reports perplexity and generation cost, adding a language-modeling view to the downstream results. Chapter 5 presents the results in the same order: overall scores, inference settings, computational cost, and the smaller models.

4.2 Evaluated Models

The large-model comparison includes Qwen3 [72] 4B and 8B as autoregressive baselines. The diffusion models are LLaDA [49], Dream [75], SDLM [45], LLaDA 1.5 [80], and Fast-dLLM v2 [70]. These models cover full-sequence and block generation, training from scratch, and adaptation from pretrained models. Chapter 3 explains their main ideas. The paper tables shorten the name Fast-dLLM v2 to "Fast-dLLM"; both names refer to the same evaluated model here.

A second comparison uses the smaller MDLM [58], BD3-LM [2], E2D2 [3], and Duo [59] checkpoints, with GPT-2 [56] as the autoregressive baseline. The experiments use released checkpoints pretrained on OpenWebText.

Table 4.1 reproduces the model overview from the paper. Its underlined data counts refer to post-training examples, while the other counts refer to pretraining tokens. These figures describe the training stages reported for each model; an adaptation data count does not include the full pretraining history of its initial checkpoint. The "Block" column records block use in the paper model overview. It should be read together with the model descriptions: LLaDA 1.5 uses a block sampling configuration here, whereas models such as BD3-LM incorporate block structure into training.

The parameter groups provide two useful scales of comparison. SDLM has roughly three billion parameters and is placed alongside Qwen3-4B; the remaining large diffusion models have seven or eight billion parameters and are placed alongside Qwen3-8B. These are comparisons between available models of broadly similar size. The table also makes visible their different initializations and training data, which are part of the systems being evaluated. Later inference sweeps instead compare different settings of the same checkpoint.

4.3 Benchmarks

The benchmark suite covers both selecting an answer from supplied candidates and producing a complete output. This distinction matters for diffusion models because scoring a known answer and generating an answer use different procedures. It also broadens the comparison: knowledge questions, executable code, translations, and constrained grids demand different kinds of output.

Table 4.1: Summary of architectural and training configurations for the evaluated DLMs, categorized into large-scale and small-scale regimes. The # Tokens column details the training data scale. Note that underlined values indicate the number of fine-tuning samples or instruction pairs, as opposed to number of tokens.

Model	Venue	# Tokens	DS Training	Parameters	Block	Initialization	Denoising type
Large scale models							
LLaDa [49]	NeurIPS 2025	2.3T	Proprietary	8B	✗	From Scratch	Absorbing
Dream [75]	arXiv 2025	580B	Open Source	7B	✗	AR	Absorbing
SDLM [45]	arXiv 2025	2.3B	Open Source	3B	✓	AR	Absorbing
LLaDa-1.5 [80]	arXiv 2025	350K	Proprietary	8B	✓	Diffusion	Absorbing
Fast-dLLM [70]	ICLR 2026	30M	Open Source	7B	✓	AR	Absorbing
Small scale models							
MDLM [58]	NeurIPS 2024	9B	OpenWebText	200M	✗	From Scratch	Absorbing
BD3-LM [2]	ICLR 2025	9B	OpenWebText	200M	✓	From Scratch	Absorbing
E2D2 [3]	NeurIPS 2025	9B	OpenWebText	170M	✓	From Scratch	Absorbing
Duo [59]	ICML 2025	9B	OpenWebText	200M	✗	From Scratch	Uniform

4.3.1 Knowledge and Commonsense

MMLU [32] contains multiple-choice questions across academic and professional subjects. It tests whether a model can identify the correct answer using the information available in the question and its learned knowledge. MMLU Redux [25] revisits MMLU with expert review and correction of problematic questions and annotations. Including both provides two views of performance on broad knowledge tasks. Their scores must be interpreted with their respective protocols: this study scores candidates for MMLU but generates an option label for MMLU Redux.

HellaSwag [78] asks the model to select a plausible continuation of a described situation. The alternatives can be linguistically plausible while differing in whether they make sense in context. This makes the task complementary to factual question answering. Its candidate endings can have different lengths, so the scoring procedure normalizes the likelihood-based score by length.

4.3.2 Mathematics and Code

GSM8K [19] contains mathematics word problems whose solutions require combining information and carrying out several arithmetic steps. The evaluated output is a generated response, from which the final numerical answer is extracted. A response can contain intermediate reasoning, but correctness is determined by the extracted answer. The task is therefore useful for studying whether a generation budget supports a correct solution, rather than simply a longer response.

HumanEval [17] provides programming tasks with function specifications and tests. MBPP [4] contains short programming problems, also evaluated through executable tests. In both cases, success requires functional code, and different valid implementations can receive the same score. This contrasts with matching the wording of a reference answer. The two benchmarks also supply different task descriptions and prompt examples, so they provide complementary evidence about code generation.

4.3.3 Translation and Structured Constraints

WMT16 English–German [12] evaluates translation from a supplied English sentence into German. The translation is compared with a reference using chrF [52], which measures character n-gram overlap through precision and recall. Unlike a binary correctness metric, chrF assigns partial credit for overlap. The task also constrains the intended content through the source sentence: additional output is useful only when it helps produce the translation.

Sudoku [74] evaluates structured constraint satisfaction. A generated grid must retain the supplied clues and satisfy the row, column, and subgrid rules. Correctness depends on relationships across many output tokens. A locally plausible digit is insufficient if it conflicts with another part of the grid. Sudoku therefore adds a different test of generation from knowledge recall or reference overlap.

The suite deliberately retains these different metrics. Comparisons are made between models within a benchmark; averaging all eight columns would mix functional correctness, answer accuracy, and translation overlap into a number with no single interpretation.

4.4 Evaluation Protocol

All downstream tasks were run with `lm-evaluation-harness`, using the evaluation framework described by Biderman et al. [8]. The models receive the same task prompts and number of examples in the prompt, with model-specific tokenization and formatting where needed. We report the best result observed for each model and benchmark over the tested inference settings. A k-shot prompt contains k examples before the item being evaluated. These examples condition the response without updating model parameters; zero-shot evaluation still supplies a task instruction or question. Unless an inference setting is explicitly varied, the study follows the original model scoring and generation configuration.

4.4.1 Candidate Scoring and Generated Answers

MMLU uses five examples in the prompt and selects the answer with the highest estimated log-likelihood. HellaSwag uses no examples and selects among candidate endings using length-normalized estimated log-likelihood. MMLU Redux uses five examples, but the model generates an option label, which is read from the first generated answer token, instead of scoring all candidate answers. Thus, MMLU and HellaSwag are likelihood-based evaluations, while MMLU Redux is a generation task.

GSM8K uses four examples in the prompt; the final number is extracted from the generated answer and compared with the reference. HumanEval uses no examples and MBPP uses three; both report pass@1 after checking whether the generated code passes the task tests. WMT16 English–German uses no examples and reports chrF on its conventional 0–100 scale. This value is not the percentage of sentences translated correctly. Sudoku also uses no examples; a generated grid counts as correct when it preserves the clues and satisfies the puzzle rules.

For the code tasks, pass@1 describes the success of a single candidate under the task tests. It does not require the generated program to match a reference implementation. Output extraction and filtering follow the evaluation framework. This separates the intended task answer from surrounding generated material before the metric is applied.

4.4.2 Likelihood-Based Evaluation

Autoregressive models can obtain a candidate log-likelihood from their next-token probabilities. Diffusion models instead use the scoring procedures defined for each model. Dream, LLaDA, and LLaDA 1.5 estimate candidate likelihoods with Monte Carlo sampling. For Fast-dLLM v2 and SDLM, the target tokens are masked and scored in one forward pass. In both cases, the resulting score selects between candidates within a model; it is not an exact sequence likelihood shared by all diffusion models.

To see the role of these scores, consider a question with several supplied answers. Evaluation compares a score for each answer conditioned on the same question and chooses the largest. A score need not be an exact sequence log-likelihood to serve this ranking procedure. Nevertheless, its construction remains part of the evaluation protocol: a single-pass masked score and a Monte Carlo estimate of a variational bound are different quantities. The comparison reports the resulting task accuracy, not a comparison of raw likelihood values across these scoring methods.

4.4.3 Selecting the Reported Configuration

A model entry in the overall results table is the highest score observed for that model and benchmark among the tested settings. The selected settings may differ across benchmarks. This answers how well each model performed during the study and allows the following sweeps to explain the sensitivity behind that score. It does not represent one fixed configuration used for every task.

Selection uses the evaluated benchmark scores themselves, so these entries should be read as best observed results in the tested range. The study does not use a separate validation search followed by a new test evaluation. Each plotted configuration is an observed result, and the headline table summarizes the best configurations considered in the broader evaluation.

4.5 Inference-Time Analysis Design

The inference experiments vary three settings: generation length, the number of denoising steps, and block size. Generation length L counts output tokens only; prompt tokens are excluded. We use S for the total number of denoising steps. When generation uses blocks, L_{block} is the number of

tokens in a block and S_{block} is the number of steps used for that block.

The four sweeps form two pairs. The first pair examines full-sequence generation and separates a joint increase in output length and steps from a change in steps alone. The second examines block generation and separates changing block boundaries from changing the number of steps within them. All four use GSM8K, HumanEval, MBPP, and WMT16 English–German, covering mathematics, code, and translation.

Joint variation of generation length and denoising steps. The first experiment increases generation length and denoising steps together for Dream and LLaDA. Both L and S take the values $\{64, 128, 256, 512, 1024, 1536\}$. Each setting therefore allows one denoising step per output token.

Denoising steps at a fixed generation length. The second experiment keeps $L = 1024$ and varies S over $\{64, 128, 256, 512, 1024\}$. Because generation length stays fixed, this second experiment shows the effect of changing the number of denoising steps more directly. With fewer steps, the model must generate more tokens in parallel at each step.

For example, at $L = 1024$, a monotonic schedule with $S = 128$ commits eight tokens per step on average, whereas $S = 1024$ allows one commitment per step. Both use the same configured number of output tokens. This example explains why step count controls parallel unmasking independently of the configured generation length.

Block size. The third experiment tests block sizes of 8, 16, 32, 64, and 128 tokens for LLaDA, LLaDA 1.5, and Fast-dLLM v2. For LLaDA and LLaDA 1.5, the total number of denoising steps remains one per output token. Fast-dLLM v2 uses its own decoding procedure at each tested block size, without enforcing one token per step. The experiment therefore measures the effect of block size under each model sampling procedure. SDLM is not included because it advances by variable-length segments rather than a fixed block size.

Denoising steps within a block. The fourth experiment selects the best-performing block size for each model from the preceding analysis. For LLaDA and LLaDA 1.5, each selected size is then held fixed while the number of denoising steps within a block varies. The ratio $S_{\text{block}}/L_{\text{block}}$ takes the values $\{1/8, 1/4, 1/2, 1\}$. A lower ratio means fewer steps and more tokens generated in parallel

per step. Fast-dLLM v2 is not included because its decoder does not set a fixed number of steps per block in the same way. Chapter 5 presents the four experiments in this order.

4.6 Computational-Cost Methodology

The cost comparison reports peak GPU memory and estimated floating-point operations (FLOPs), both for a reference forward pass and for generation on GSM8K. These measurements answer complementary questions: memory describes the storage required at a given moment, whereas FLOPs accumulate the computation performed during a run. A forward pass is one model call; generation may require many calls.

Profiling used NVIDIA A100 GPUs.

4.6.1 Workload and Aggregation

For full-sequence diffusion, a reference forward pass processes the prompt together with the masked output. For autoregressive models, it processes the prompt alone, corresponding to prefill. The single-pass measurements therefore describe the inputs used by each generation procedure.

Full-sequence diffusion profiling uses a configured generation length of 1,024 output tokens and 1,024 denoising steps. Autoregressive generation allows up to 1,024 new tokens and can stop when an end-of-sequence token is produced. A common output limit therefore does not enforce an equal realized output length. Each model follows its implementation stopping and caching behavior.

The profiling utility skips five warmup runs and averages the next 100 runs. The resulting FLOP value is a mean per generation, rather than the sum for all 100 runs. For memory, it records the peak PyTorch-allocated memory of each run and then averages those peaks. This is distinct from both the largest peak across the dataset and the total memory shown by a device monitor. The measurement includes resident model tensors; it is not just the additional memory used by generation.

4.6.2 FLOP Estimation

The reference forward pass is measured with the FLOP accounting provided by `torch.profiler`. Full generation is estimated from that measurement and the input lengths of intercepted forward

calls. If the reference uses n_{ref} input tokens and has measured cost F_{ref}, the estimator is

$$\widehat{F}_{\text{gen}} = \frac{F_{\text{ref}}}{n_{\text{ref}}} \sum_{j=1}^{C} n_j, \tag{4.1}$$

where C is the number of intercepted calls and n_j is the input length of call j. The reference supplies an estimated cost per input token, which is then accumulated over generation. Repeated full-sequence calls process many tokens each time; a cached autoregressive call can supply only the newly processed token. Early stopping also reduces the accumulated total.

This approximation makes the treatment of repeated computation explicit. It does not separately scale attention work with the length of cached context, and its reference value includes only operations covered by the profiler. The reported FLOPs are therefore estimates under this procedure. In the reproduced tables, "TFLOPS" denotes trillions of operations, 10^{12} FLOPs. No latency or throughput measurement is inferred from these values.

4.7 Small-Scale Matched-Corpus Methodology

The second branch compares the released MDLM, BD3-LM, E2D2, and Duo checkpoints in Table 4.1 with a GPT-2 autoregressive reference. The selected models share an OpenWebText training-corpus basis. This provides a more focused comparison of predictive quality and cost at smaller scale than the large-model downstream evaluation, while retaining differences in architecture, objective, and training procedure. Sharing a corpus does not make those factors identical. The model designs were surveyed in Chapter 3.

Predictive quality is evaluated on a 1,000-example mixture drawn from GSM8K, MBPP, HumanEval, WMT16 English-to-German, MMLU, and HellaSwag. The mixture is a collection of evaluation texts, not an ensemble of model predictions; this is the meaning of "ensemble" in the paper original table caption. The reported metric is token-normalized perplexity, as introduced in Equation 2.3. Autoregressive likelihood follows a causal factorization; diffusion-model scores require an estimate because the denoising trajectory is not directly observed. Thus, the perplexity table reports each model score under the study evaluation procedure. This evaluates the assigned probability of reference text. It is separate from generating a solution and checking whether it is correct: a model can assign a favorable score to a reference without its chosen sampler producing

that reference.

The small-model analysis consequently has two roles. It examines whether different corruption and architectural choices lead to similar predictive scores on the same evaluation mixture, and it compares the work performed in a separate generation task. It complements the large-model experiments without assuming that perplexity determines the downstream ranking.

The cost comparison measures a reference forward pass and unconditioned generation of a configured 1,024-token sequence. It reports peak allocated GPU memory and FLOP counts for each workload. Together, the predictive and computational results show how the smaller models compare when evaluated on the same text mixture and a fixed output-length task. Chapter 5 presents these results after the large-model downstream and inference-time analyses.

5. Experiments and Results

This chapter presents the experimental results of the study described in Chapter 4. The result tables and four experimental plots reproduce the pape results [7]; the accompanying analysis develops their interpretation. We first compare the models across eight benchmarks, then examine how generation length, denoising steps, and block size affect performance. Finally, we compare computational cost and analyze the smaller models.

5.1 Overall Downstream Evaluation

Table 5.1 shows the best result obtained for each model on each benchmark among the settings tested. Scores can be compared across models within a column. The WMT16 column reports chrF; the other columns report percentages, including pass@1 for code and the valid-solution rate for Sudoku, following the metrics defined in Section 4.4.

Knowledge and commonsense. Qwen3-8B has the highest scores on MMLU, MMLU Redux, and HellaSwag. Dream is the strongest diffusion model on all three and exceeds Qwen3-4B on each of them. The diffusion models do not all behave the same way: Fast-dLLM v2 reaches 71.18% on MMLU Redux but only 30.82% on HellaSwag. Evaluating several tasks reveals these differences. Dream gaps to Qwen3-8B are 3.05 percentage points on MMLU, 2.42 on MMLU Redux, and 1.20 on HellaSwag. These results place its performance close to the larger autoregressive baseline on this group of tasks.

The difference between MMLU and MMLU Redux is also useful when reading the table. Their evaluation differs in both question set and answer procedure, so a higher Redux score does not directly show an improvement in the same task. For Fast-dLLM v2, the contrast between its knowledge scores and HellaSwag establishes an uneven task profile; the table alone does not identify whether that contrast comes from training, candidate scoring, or another model-specific choice.

Table 5.1: Comparison of peak benchmark accuracy (%) across autoregressive and DLMs. Autoregressive baselines (Qwen3) are shaded in gray, while bold text indicates the highest performance achieved among the diffusion-based architectures.

Model	GSM8K	HumanEval	MBPP	MMLU	MMLU-Redux	HellaSwag	Sudoku	WMT16 En–De
Qwen3-4B	81.19	71.95	62.60	70.06	74.80	68.48	2.00	55.37
SDLM	62.00	63.41	56.60	64.96	62.55	69.70	2.00	50.11
Qwen3-8B	87.41	63.41	64.80	74.78	78.42	74.97	8.00	58.21
Dream	77.75	57.92	**57.00**	**71.73**	**76.00**	**73.77**	**75.00**	45.61
LLaDa	65.42	33.53	40.60	65.84	61.33	70.91	46.00	51.31
LLaDa-1.5	82.41	50.00	42.40	64.14	61.38	69.70	28.00	**54.85**
Fast-dLLM	**83.39**	**69.51**	49.00	68.00	71.18	30.82	1.00	43.62

Mathematics and code. On GSM8K, Qwen3-8B leads with 87.41% accuracy. Fast-dLLM v2 is close at 83.39%, the best result among diffusion models, followed by LLaDA 1.5 at 82.41%. HumanEval shows a similarly strong result for Fast-dLLM v2: its 69.51% pass@1 is above Qwen3-8B 63.41% and close to Qwen3-4B 71.95%. MBPP gives a different ordering. Dream is the strongest diffusion model at 57.00%, while Qwen3-8B leads at 64.80%. The two code benchmarks therefore show different strengths across the models. In particular, Fast-dLLM v2 strong HumanEval result does not extend to MBPP, where its 49.00% pass@1 is below both Dream and SDLM. This makes a single code benchmark an incomplete description of coding performance.

SDLM also deserves attention at the smaller parameter scale. It matches Qwen3-8B on HumanEval at 63.41%, despite having roughly three billion parameters, and reaches 56.60% on MBPP. Its GSM8K score is lower than both Qwen baselines. Parameter count helps place these results in context, but it does not by itself predict a model ranking: even the two Qwen baselines reverse their order on HumanEval.

Translation and Sudoku. Qwen3-8B leads WMT16 English–German with 58.21 chrF. LLaDA 1.5 is the strongest diffusion model at 54.85, close to Qwen3-4B 55.37. Sudoku gives the clearest diffusion advantage: Dream solves 75% of the puzzles, compared with 8% for Qwen3-8B. LLaDA also performs well at 46%. The 67-percentage-point gap between Dream and Qwen3-8B makes this the strongest task-specific advantage for diffusion in the comparison. Because grid validity

depends on relationships across the output, this result motivates interest in flexible generation order and bidirectional conditioning. These mechanisms allow the model to predict masked tokens using information from already completed parts of the grid, even when previously committed tokens remain fixed.

The LLaDA variants illustrate another useful contrast. LLaDA 1.5 improves over LLaDA by 16.99 points on GSM8K and 16.47 points on HumanEval, but has a lower Sudoku score, 28% rather than 46%. Their differences in post-training and inference configuration therefore accompany improvements on some tasks rather than a uniform increase across the suite. The per-task view preserves this information.

Overall, Dream leads the diffusion models on five benchmarks, Fast-dLLM v2 on two, and LLaDA 1.5 on translation. The table shows that diffusion models are competitive, but their strengths depend on the task. The next section examines how changing their inference settings affects these results.

5.2 Inference-Time Hyperparameter Analysis

The overall table reports each model best score. The following experiments show how scores change when generation length, denoising steps, or block size changes. They use GSM8K, HumanEval, MBPP, and WMT16 English–German. In the plots, accuracy and pass@1 are shown as fractions, while the overall table expresses them as percentages; chrF retains the same scale. Curves connect the tested settings to make their trends visible. The analysis concerns the observed changes, without treating small differences as statistically established effects.

5.2.1 Generation Length and Steps Varied Together

Figure 5.1 increases generation length and denoising steps together for Dream and LLaDA. Both range from 64 to 1,536, with one step per output token. This experiment shows what happens when both the maximum output length and the number of steps increase.

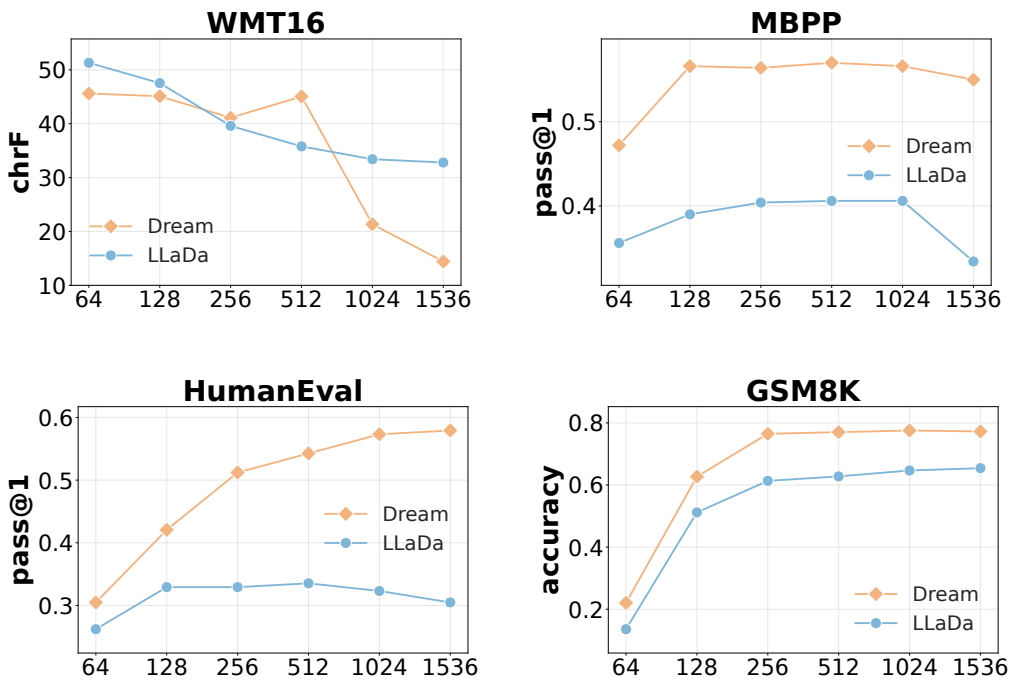


Figure 5.1: Dream and LLaDA as generation length and denoising steps increase together. Generation length counts output tokens only. Panels show WMT16 En–De, MBPP, HumanEval, and GSM8K from top left to bottom right.

GSM8K and MBPP improve sharply at first, then gain less as the settings grow. Dream continues to improve on HumanEval, while LLaDA levels off and then declines. Translation behaves differently: LLaDA loses quality as generation length and steps increase, and Dream falls sharply at the two largest settings.

The MBPP curves make the saturation especially visible: Dream obtains most of its improvement by 128 output tokens, while LLaDA improves more gradually before dropping at 1,536. GSM8K gains are concentrated at the shorter settings, particularly for Dream. Allocating more output tokens and denoising calls beyond that region produces little further gain.

This experiment changes two aspects of inference together. It provides more output space and more denoising calls, and a full-sequence call also processes the larger reserved output. The result is therefore a comparison of joint generation budgets. The next experiment holds output length fixed to examine the effect of denoising steps separately.

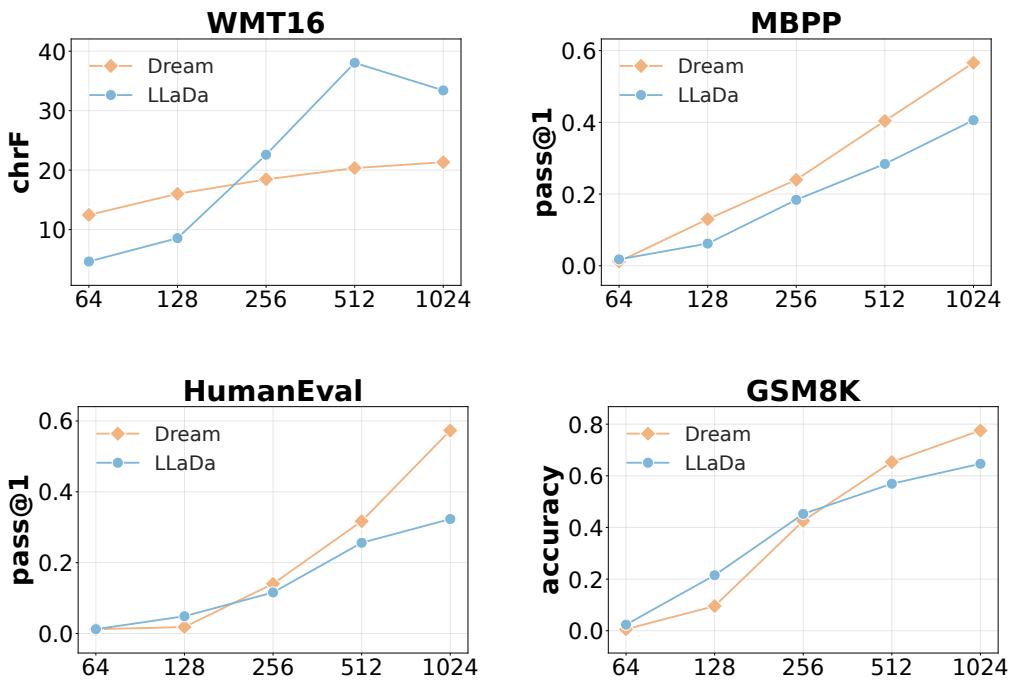


Figure 5.2: Dream and LLaDA with a fixed generation length of 1,024 output tokens and different numbers of denoising steps. Panels show WMT16 En–De, MBPP, HumanEval, and GSM8K from top left to bottom right.

5.2.2 Steps at Fixed Generation Length

Figure 5.2 keeps generation length at 1,024 tokens and changes the number of denoising steps from 64 to 1,024. With fewer steps, the model must generate more tokens at each step. This changes the average number of newly committed tokens per pass from sixteen at 64 steps to one at 1,024 steps under a monotonic schedule. With fewer passes, more tokens are selected using the same partially completed context.

On GSM8K, MBPP, and HumanEval, quality generally rises as more steps are added. The effect is especially clear for code generation: performing fewer steps means generating more tokens in parallel, and the resulting code passes fewer tests. Translation is less uniform. Dream improves gradually, while LLaDA reaches its best plotted score at 512 steps and drops slightly at 1,024.

The model ordering also changes with the step budget. At low budgets, LLaDA is ahead of Dream on GSM8K; Dream overtakes it as more steps are allowed. On HumanEval, both are close

to zero at the smallest setting, while Dream gains more strongly toward the largest settings. This is why the best-score table and the full curves answer different questions: the model with the highest observed score need not lead when only a small number of denoising calls is available.

Read together, the first two experiments show that high parallelism and long generation are separate choices. For code, giving the model more successive contexts at a fixed length is beneficial throughout much of the tested range. Increasing length at the same time can reach a plateau or reverse the gain. The evidence supports choosing an output length suited to the task and then considering the step budget, rather than treating both settings as one measure of generation quality.

5.2.3 Effect of Block Size

Figure 5.3 varies block size from 8 to 128 tokens for LLaDA, LLaDA 1.5, and Fast-dLLM v2. The LLaDA models retain one denoising step per output token. Fast-dLLM v2 uses its own decoding procedure, with block size as the varied parameter.

Scores change little on WMT16, and the LLaDA models also remain fairly stable on HumanEval. On GSM8K, Fast-dLLM v2 stays near its strongest scores, while both LLaDA variants lose some accuracy toward larger blocks. MBPP is more sensitive: Fast-dLLM v2 peaks at a block size of 16, while the LLaDA models vary more modestly. Fast-dLLM v2 also declines on HumanEval as blocks become larger. Overall, block size has less effect than denoising steps in most of the tested tasks, though the best choice still depends on the model and benchmark.

The ordering is stable on several tasks, but not everywhere. LLaDA 1.5 stays above the other two models on translation across all five block sizes. Fast-dLLM v2 leads HumanEval throughout, even though its score falls at the largest block size. On MBPP, however, its advantage at small blocks narrows and LLaDA 1.5 slightly exceeds it at 128 tokens. The latter crossover would be lost in a description of the experiment as complete invariance to block size.

For the LLaDA models, keeping one step per output token means that larger blocks change the grouping of tokens without increasing the number of parallel commitments. They change which output tokens can be chosen at each step and the context available during that process. For example, dividing 128 output tokens into blocks of eight or sixteen changes the number of block transitions, even if both configurations allocate 128 denoising steps in total. This isolates a different practical choice from the preceding step-count sweep.

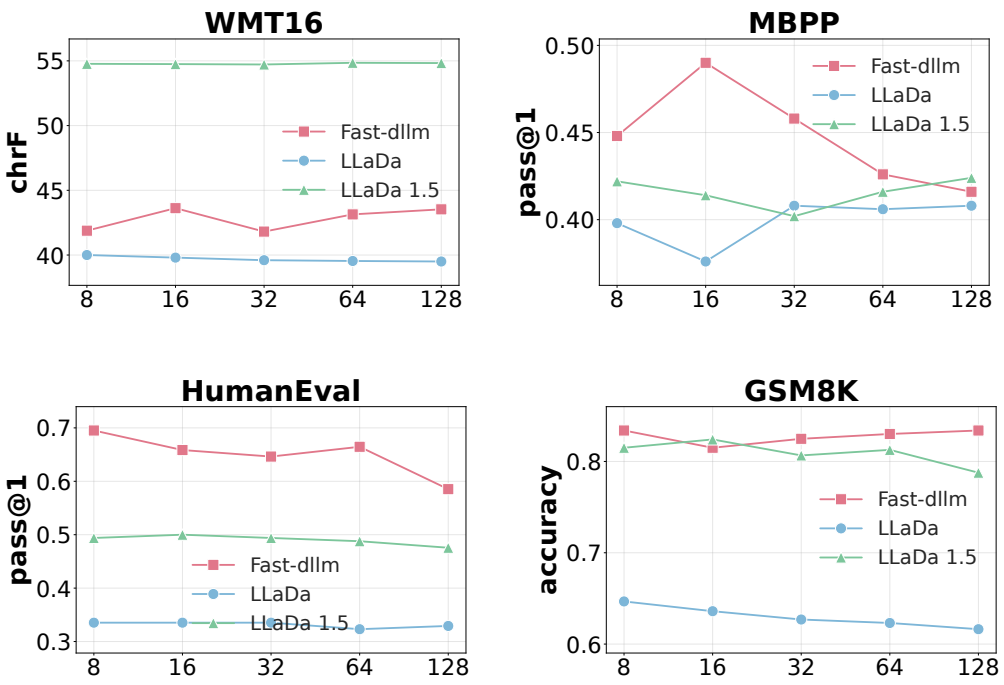


Figure 5.3: Scores as block size changes from 8 to 128 tokens. LLaDA and LLaDA 1.5 use one denoising step per output token; Fast-dLLM v2 follows its own decoding procedure without this constraint. Panels show WMT16 En–De, MBPP, HumanEval, and GSM8K from top left to bottom right.

5.2.4 Denoising Steps Within a Block

The last experiment fixes each model block size at its best setting from the preceding sweep and varies denoising steps within each block for LLaDA and LLaDA 1.5. Figure 5.4 shows the ratio of steps to block length, from $1/8$ to 1. At a lower ratio, more tokens are generated in parallel in each step.

Both models improve strongly on MBPP and HumanEval as the ratio increases: code generation benefits from more steps within each block. GSM8K also improves, but gains become smaller near a ratio of one. For translation, LLaDA 1.5 keeps improving, whereas LLaDA peaks near $1/2$. These results confirm that the number of steps matters inside blocks as well as across the full sequence.

The relative gains also depend on the task. LLaDA 1.5 has a clear advantage on HumanEval and GSM8K, especially when more steps are available. On MBPP, the two models become nearly equal

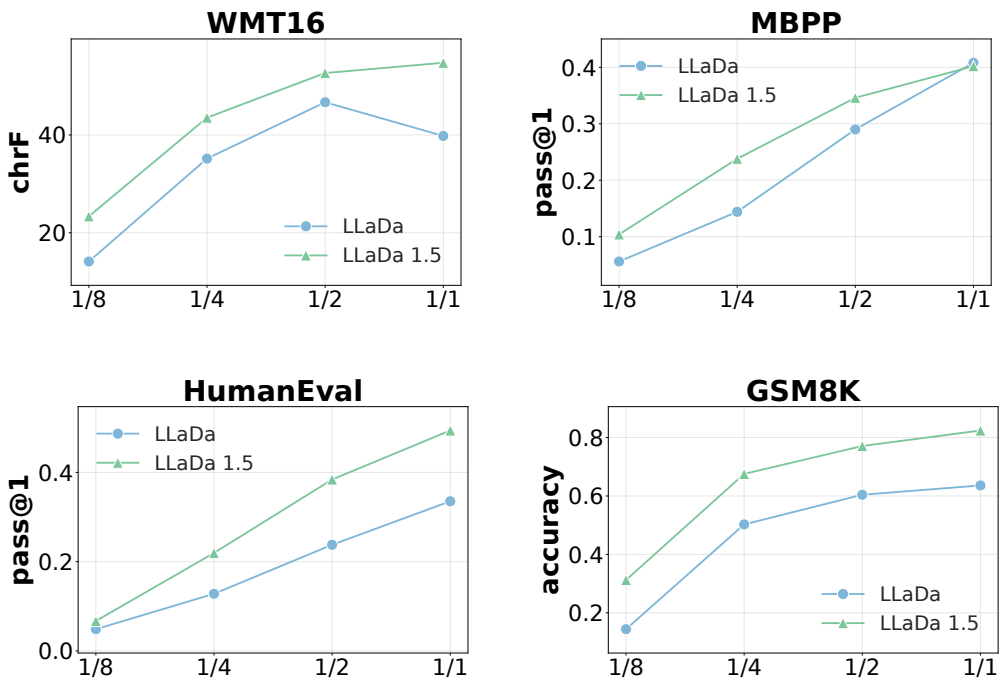


Figure 5.4: LLaDA and LLaDA 1.5 as the number of denoising steps per block token increases from $1/8$ to 1. Panels show WMT16 En–De, MBPP, HumanEval, and GSM8K from top left to bottom right.

at the one-step-per-token setting even though LLaDA 1.5 is stronger at lower ratios. For translation, the best ratio differs between the two models. A sampling choice that improves one checkpoint therefore need not be the best choice for a related checkpoint on every benchmark.

The comparison with the block-size sweep is instructive. Changing which tokens share a block often preserves much of the achieved quality; reducing the number of successive predictions inside that block can produce large losses. In these experiments, retaining block structure is not enough to preserve code quality under aggressive parallel unmasking. Later predictions still benefit from conditioning on tokens committed by earlier passes.

5.3 Computational-Cost Analysis

Table 5.2 compares peak VRAM and FLOPs for one forward pass and for complete GSM8K generation. The full-sequence models use one denoising step per output token, a setting in which

Table 5.2: Computational cost comparison across the evaluated models. Metrics represent peak GPU memory (VRAM) and floating-point operations (TFLOPS) for a single forward pass versus full generation on GSM8K, with autoregressive baselines shaded in gray.

	1 Forward Pass		GSM8K Generation	
Model	Peak VRAM (GB)	TFLOPS	Peak VRAM (GB)	TFLOPS
Qwen3-4B	8.36	5.04	8.36	9.85
SDLM	7.49	4.15	7.51	6.72
Qwen3-8B	16.68	9.47	16.68	19.47
Dream	15.80	23.23	20.45	23726.95
LLaDa	16.48	24.82	17.31	25357.94
LLaDa 1.5	16.48	25.01	17.32	25557.57
Fast-dLLM	15.50	9.73	15.55	33.38

they achieve strong task scores. FLOPs are estimated computational operations, not elapsed time.

A single forward pass costs about 23–25 TFLOPs for Dream, LLaDA, and LLaDA 1.5, compared with 9.47 TFLOPs for Qwen3-8B and 9.73 TFLOPs for Fast-dLLM v2. These values describe different inputs: full-sequence diffusion processes the prompt and masked output, whereas autoregressive prefill processes only the prompt. The generation columns are more useful for seeing how the cost accumulates while producing an answer.

During generation, Dream, LLaDA, and LLaDA 1.5 each reach roughly 24,000–26,000 TFLOPs. Qwen3-8B records 19.47 TFLOPs, Fast-dLLM v2 33.38 TFLOPs, and SDLM 6.72 TFLOPs. These totals are consistent with the repeated full-sequence processing visible in the profiling procedure. Dream, LLaDA, and LLaDA 1.5 process the reserved output again at each denoising step. Qwen3 reuses earlier computations through KV caching, while Fast-dLLM v2 and SDLM organize generation around a growing prefix. The cost of one pass alone cannot explain the generation totals.

For the full-sequence models, generation requires roughly a thousand times the estimated work of a reference pass, consistent with the 1,024-step schedule and repeated processing of similarly sized inputs. For the autoregressive baselines, the estimate is about twice the prefill value: subsequent cached calls process one new token at a time, and generation may stop before reaching the output limit. These ratios follow the estimation procedure described in Section 4.6.

The VRAM results tell a different story. Dream reaches 20.45 GB during generation, while LLaDA and LLaDA 1.5 use about 17.3 GB. Qwen3-8B uses 16.68 GB and Fast-dLLM v2 15.55 GB. The models thus differ much more in repeated computation than in peak memory. This is relevant because a model can fit in GPU memory yet still require substantial work to generate an answer. Repeated calls reuse working memory, so their FLOPs accumulate even when their intermediate tensors do not all remain resident. Conversely, caches retain tensors between calls and can increase storage while saving repeated computation. This explains why the memory and FLOP columns need not scale together.

The block-oriented models occupy a different region of the reported cost comparison. Fast-dLLM v2 has a generation estimate much closer to Qwen3-8B than to the full-sequence models, while SDLM has the lowest generation estimate in the table. Under the estimator and stopping rules documented in Section 4.6, repeated full-sequence denoising has a large computational requirement in this high-step setting. The relatively close memory figures show why both measurements are needed to characterize these generation workloads.

5.4 Small-Scale Matched-Corpus Analysis

Table 5.3 compares four smaller diffusion models with GPT-2. The models were trained on OpenWebText, and perplexity was evaluated on the 1,000-example mixture described in Section 4.7. The cost columns describe one forward pass and unconditioned generation of 1,024 tokens.

Table 5.3: Ensemble perplexity and computational cost comparison for small-scale architectures. The table reports raw predictive performance (PPL) alongside peak VRAM and TFLOPS for both a single forward pass and unconditioned generation over a fixed 1024-token sequence. The autoregressive baseline (GPT-2) is shaded in gray.

		1 Forward Pass		Generation	
Model	PPL↓	Peak VRAM (GB)	TFLOPS	Peak VRAM (GB)	TFLOPS
GPT-2	20.98	2.39	0.292	2.34	1.75
E2D2	36.82	5.45	1.03	5.45	263.15
BD3-LM	36.16	1.20	0.253	1.20	259.04
MDLM	28.45	1.07	0.253	1.07	259.16
Duo	**24.36**	1.08	0.253	1.08	259.22

GPT-2 has the lowest reported perplexity at 20.98. Among the diffusion models, Duo performs best at 24.36, followed by MDLM at 28.45. BD3-LM and E2D2 both score around 36. Duo therefore comes closest to the autoregressive baseline in this smaller-model comparison. As explained in Chapter 4, diffusion perplexity relies on likelihood estimation, so these are the scores under the study evaluation procedure.

For one forward pass, MDLM, Duo, and BD3-LM each use 0.253 TFLOPs. E2D2 uses 1.03 TFLOPs and has the highest peak VRAM at 5.45 GB. During unconditioned generation, all four diffusion models use about 259–263 TFLOPs, compared with 1.75 TFLOPs for GPT-2. Their perplexity scores differ more than their generation FLOP estimates. This comparison shows that the cost of diffusion generation depends on the actual sampling procedure, as well as the model design.

5.5 Cross-Experiment Discussion

The preceding sections examined each experiment in turn. Their connections help explain why some generation choices are useful across tasks while others require more careful selection.

5.5.1 Task Performance Depends on the Model and Its Configuration

The downstream evaluation establishes competitive diffusion results on several tasks and a pronounced advantage for Dream on Sudoku. It also shows that full-sequence and block-oriented models have different strengths. No single diffusion model leads across mathematics, code, knowledge, translation, and structured constraints. Reporting a suite of tasks is therefore central to the contribution: it reveals differences that an aggregate ranking would hide.

The step sweeps add a second part to that conclusion. The model position in a ranking can change with its step budget, as Dream and LLaDA show on GSM8K. A best observed score describes the performance reached after choosing settings; it does not fully describe behavior across those settings. The curves make that sensitivity visible and identify when additional generation budget produces little benefit.

This also helps interpret the role of bidirectional context. Diffusion can generate output tokens in a flexible order, so later commitments can use already resolved tokens on either side. Sudoku provides a strong empirical reason to study it, while code and translation show that the useful

schedule depends on the task.

5.5.2 Denoising Steps and Computational Cost

The inference sweeps and profiling highlight two separate choices: how many times the model is called, and how much of the sequence it processes on each call. Parallel unmasking can reduce the number of calls by generating several tokens together. Caching can reduce the work within a call by reusing representations that have already been computed.

For example, a full-sequence denoiser generating 1,024 tokens in 1,024 steps repeatedly processes the complete output. A block model can use many denoising steps while processing a smaller active block and reusing its completed prefix.

This distinction connects the quality and cost findings. The code experiments show the benefit of retaining more denoising steps, including within a block. The profiling results motivate reducing the work done at each step through block generation and reuse. Together, they suggest that useful efficiency improvements can come from preserving successive predictions while making each prediction less expensive.

5.5.3 Implications for Selecting an Inference Configuration

The paired sweeps show that increasing generation length does not consistently improve performance: gains can saturate or reverse, especially in translation. Increasing the number of denoising steps at a fixed length generally improves performance over the tested range, particularly on mathematics and code. The results therefore favor allocating sufficient denoising steps without assuming that a longer configured output will produce a better answer.

Block size generally has a smaller influence on performance than the number of denoising steps within a block. This relative stability suggests room to choose a block size that suits the available memory and the implementation computational requirements. Retaining enough denoising steps remains important for quality, especially for code; the best block size can still differ across models and tasks.

The smaller-model results reinforce the value of reporting predictive quality alongside cost, since similar generation FLOP estimates accompany different perplexities.

6. Conclusions

This thesis evaluated diffusion language models across eight downstream benchmarks and examined how their performance changes with generation length, denoising steps, and block size. Profiling of peak GPU memory and estimated FLOPs added a computational perspective, while a complementary study of smaller models pretrained on OpenWebText examined reference-text prediction through perplexity.

6.1 Answers to the Research Questions

RQ1: Performance across language tasks. No single diffusion model leads across all evaluated tasks. Dream leads the diffusion models on five of the eight benchmarks, with its strongest advantage over the autoregressive baselines on Sudoku: 75% valid solutions compared with 8% for Qwen3-8B. Fast-dLLM v2 gives the strongest diffusion results on GSM8K and HumanEval, while LLaDA 1.5 leads on translation. The Qwen baselines remain strong across the suite, but the comparison identifies tasks on which diffusion models are competitive or substantially better.

These results support evaluating several kinds of language task. Even the two code benchmarks produce different model rankings, showing that a result on one benchmark need not represent a model performance across a whole category. In the smaller-model study, Duo has the lowest reported perplexity among the diffusion models and comes closest to GPT-2, which has the lowest value overall.

RQ2: The effect of inference-time hyperparameters. Increasing generation length does not consistently improve performance: gains can saturate or reverse, particularly for translation. By comparison, increasing the number of denoising steps at a fixed length generally improves mathematics and code performance over the tested range. The main implication is to allocate sufficient denoising steps while choosing an output length suited to the task.

Block size generally has a smaller effect than the number of steps within a block. This leaves flexibility to select it according to available memory and implementation requirements, although

the best setting can differ across models and tasks. Reducing within-block steps can still produce large losses on code benchmarks. The number of successive denoising steps therefore remains important for quality in both full-sequence and block generation. Changes in model rankings across step budgets further support reporting performance curves alongside best observed scores.

RQ3: Memory and estimated computation. Full-sequence denoising in the studied high-step setting has much larger estimated generation FLOPs than the evaluated autoregressive and block-oriented implementations. The differences in peak memory are considerably smaller. Repeatedly processing the reserved output can therefore accumulate substantial computation without a proportional increase in memory requirements.

The block-oriented implementations have lower generation FLOP estimates in the workload defined in Chapter 4. Together with the inference sweeps, this motivates reducing the work performed at each step through block generation and reuse, while retaining enough denoising steps for good task performance. The smaller-model study adds a related observation: the diffusion models have similar generation FLOP estimates despite their different perplexities.

6.2 Overall Contribution

The contribution is an experimental account of where the evaluated diffusion models perform well and which generation choices affect that performance. The benchmark comparison identifies task-specific strengths; the paired sweeps separate the effects of output length, steps, and block organization; and profiling explains the computational requirements of the studied generation procedures. Together, the results show why inference configuration deserves a central place in the evaluation of diffusion language models. They also provide concrete starting points for choosing configurations and designing further comparisons.

6.3 Future Directions

Adapting the generation budget. The varying benefit of additional steps motivates procedures that allocate them according to the state of the output. A model could commit more tokens when predictions are reliable and retain more masked tokens when dependencies are difficult.

The relevant question is whether such a procedure preserves the quality of a fixed schedule while reducing work. Output-length selection is also worth investigating, especially for translation, where longer configured outputs did not consistently help.

Understanding generation order on constrained tasks. The Sudoku result motivates examining how token commitment order interacts with global constraints. Tracing partial outputs could reveal which tokens are generated first in successful and unsuccessful solutions. A related comparison could distinguish flexible unmasking order from explicit editing of already committed tokens, examining how each uses information acquired during generation.

Extending the comparison to newer models. The survey identifies developments in scale, post-training, editing, and block generation beyond the evaluated set. Extending the same experimental questions to these models would show which patterns persist as the field develops. Multimodal models would require an extension of the benchmark suite and accounting for their additional inputs.

Continuous diffusion and flow language models. Continuous representations have recently drawn renewed attention in language generation. They can retain graded information about token alternatives during generation, while continuous flows also admit learned flow maps for one-step or few-step sampling [37]. Comparing continuous diffusion and flow models with discrete models under matched data and inference budgets would clarify when these representational and sampling choices improve text quality or cost.

The study establishes diffusion language modeling as a useful alternative for text generation with clear task-specific strengths. Its results place particular emphasis on the generation procedure: choosing how much output to reserve, how many steps to use, and how to organize those steps is central to obtaining the capabilities that these models offer.

Bibliography

[1] Joshua Ainslie et al. “GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints”. In: *EMNLP*. 2023.

[2] Marianne Arriola et al. “Block Diffusion: Interpolating Between Autoregressive and Diffusion Language Models”. In: *ICLR*. 2025.

[3] Marianne Arriola et al. “Encoder-Decoder Diffusion Language Models for Efficient Training and Inference”. In: *NeurIPS*. 2025.

[4] Jacob Austin et al. “Program Synthesis with Large Language Models”. In: *arXiv preprint arXiv:2108.07732* (2021).

[5] Jacob Austin et al. “Structured Denoising Diffusion Models in Discrete State-Spaces”. In: *NeurIPS*. 2021.

[6] Yoshua Bengio et al. “A Neural Probabilistic Language Model”. In: *JMLR* (2003).

[7] Thomas Bertolani et al. “Diffusion Language Models: An Experimental Analysis”. In: *arXiv preprint arXiv:2606.19475* (2026).

[8] Stella Biderman et al. “Lessons from the Trenches on Reproducible Evaluation of Language Models”. In: *arXiv preprint arXiv:2405.14782* (2024).

[9] Tiwei Bie et al. “LLaDA2.0: Scaling Up Diffusion Language Models to 100B”. In: *arXiv preprint arXiv:2512.15745* (2025).

[10] Tiwei Bie et al. “LLaDA2.1: Speeding Up Text Diffusion via Token Editing”. In: *arXiv preprint arXiv:2602.08676* (2026).

[11] Nicholas M Boffi, Michael S Albergo, and Eric Vanden-Eijnden. “Flow map matching with stochastic interpolants: A mathematical framework for consistency models”. In: *arXiv preprint arXiv:2406.07507* (2024).

[12] Ondřej Bojar et al. “Findings of the 2016 Conference on Machine Translation”. In: *WMT*. 2016.

[13] Tom Brown et al. “Language Models are Few-Shot Learners”. In: *NeurIPS*. 2020.

[14] Davide Bucciarelli et al. “Tiny Inference-Time Scaling with Latent Verifiers”. In: *CVPR Findings*. 2026.

[15] Davide Caffagni et al. “Mind the Heads: Topological Representation Alignment for Multimodal LLMs”. In: *NeurIPS*. 2026.

[16] Davide Caffagni et al. “The revolution of multimodal large language models: A survey”. In: *ACL Findings*. 2024.

[17] Mark Chen et al. “Evaluating Large Language Models Trained on Code”. In: *arXiv preprint arXiv:2107.03374* (2021).

[18] Shuang Cheng et al. “SDAR: A Synergistic Diffusion-AutoRegression Paradigm for Scalable Sequence Generation”. In: *ACL Findings*. 2026.

[19] Karl Cobbe et al. “Training Verifiers to Solve Math Word Problems”. In: *arXiv preprint arXiv:2110.14168* (2021).

[20] Alberto Compagnoni et al. “Mitigating Hallucinations in Multimodal LLMs via Object-aware Preference Optimization”. In: *BMVC*. 2025.

[21] Alberto Compagnoni et al. “ReAG: Reasoning-Augmented Generation for Knowledge-based Visual Question Answering”. In: *CVPR*. 2026.

[22] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *NAACL*. 2019.

[23] Fabio D’Oronzio et al. “GramSR: Visual Feature Conditioning for Diffusion-Based Super-Resolution”. In: *ICPR*. 2026.

[24] Itai Gat et al. “Discrete Flow Matching”. In: *NeurIPS*. 2024.

[25] Aryo Pradipta Gema et al. “Are We Done with MMLU?” In: *NAACL*. 2025.

[26] Shansan Gong et al. “DiffuCoder: Understanding and Improving Masked Diffusion Models for Code Generation”. In: *ICLR*. 2026.

[27] Shansan Gong et al. “DiffuSeq: Sequence to Sequence Text Generation with Diffusion Models”. In: *ICLR*. 2023.

[28] Shansan Gong et al. “Scaling Diffusion Language Models via Adaptation from Autoregressive Models”. In: *ICLR*. 2025.

[29] Aaron Grattafiori et al. “The Llama 3 Herd of Models”. In: *arXiv preprint arXiv:2407.21783* (2024).

[30] Ishaan Gulrajani and Tatsunori B. Hashimoto. “Likelihood-Based Diffusion Language Models”. In: *NeurIPS*. 2023.

[31] Daya Guo et al. “DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning”. In: *arXiv preprint arXiv:2501.12948* (2025).

[32] Dan Hendrycks et al. “Measuring Massive Multitask Language Understanding”. In: *ICLR*. 2021.

[33] Jonathan Ho, Ajay Jain, and Pieter Abbeel. “Denoising Diffusion Probabilistic Models”. In: *NeurIPS*. 2020.

[34] Emiel Hoogeboom et al. “Argmax Flows and Multinomial Diffusion: Learning Categorical Distributions”. In: *NeurIPS*. 2021.

[35] Keya Hu et al. “ELF: Embedded Language Flows”. In: *arXiv preprint arXiv:2605.10938* (2026).

[36] Amin Karimi Monsefi et al. “FS-DFM: Fast and Accurate Long Text Generation with Few-Step Diffusion Language Models”. In: *ICLR*. 2026.

[37] Chanhyuk Lee et al. “Flow Map Language Models: One-step Language Modeling via Continuous Denoising”. In: *arXiv preprint arXiv:2602.16813* (2026).

[38] Yaniv Leviathan, Matan Kalman, and Yossi Matias. “Fast Inference from Transformers via Speculative Decoding”. In: *ICML*. 2023.

[39] Shufan Li et al. “LaViDa: A Large Diffusion Language Model for Multimodal Understanding”. In: *NeurIPS*. 2025.

[40] Tianyi Li et al. “A Survey on Diffusion Language Models”. In: *arXiv preprint arXiv:2508.10875* (2025).

[41] Xiang Lisa Li et al. “Diffusion-LM Improves Controllable Text Generation”. In: *NeurIPS*. 2022.

[42] Yaron Lipman et al. “Flow Matching for Generative Modeling”. In: *ICLR*. 2023.

[43] Yaron Lipman et al. “Flow Matching Guide and Code”. In: *arXiv preprint arXiv:2412.06264* (2024).

[44] Aixin Liu et al. “DeepSeek-V3 Technical Report”. In: *arXiv preprint arXiv:2412.19437* (2024).

[45] Yangzhou Liu et al. “Sequential Diffusion Language Models”. In: *arXiv preprint arXiv:2509.24007* (2025).

[46] Aaron Lou, Chenlin Meng, and Stefano Ermon. “Discrete Diffusion Modeling by Estimating the Ratios of the Data Distribution”. In: *ICML*. 2024.

[47] Federico Melis et al. “Harnessing Self-Supervised Features for Art Classification”. In: *arXiv preprint arXiv:2605.18974* (2026).

[48] Marco Morini et al. “Look Twice: Training-Free Evidence Highlighting for Knowledge-based Visual Question Answering”. In: *arXiv preprint arXiv:2604.01280* (2026).

[49] Shen Nie et al. “Large Language Diffusion Models”. In: *NeurIPS*. 2025.

[50] Shen Nie et al. “Scaling up Masked Diffusion Models on Text”. In: *ICLR*. 2025.

[51] Long Ouyang et al. “Training Language Models to Follow Instructions with Human Feedback”. In: *NeurIPS*. 2022.

[52] Maja Popović. “chrF: Character n-gram F-score for Automatic MT Evaluation”. In: *Proceedings of the Tenth Workshop on Statistical Machine Translation*. 2015.

[53] Samuele Poppi et al. “Safe-CLIP: Removing NSFW Concepts from Vision-and-Language Models”. In: *ECCV*. 2024.

[54] Tobia Poppi et al. “Do Models Share Safety Representations? Cross-Model Steering for Safe Visual Generation”. In: *arXiv preprint arXiv:2606.05290* (2026).

[55] Federico Putamorsi et al. “SPARK: Input-Conditioned Sparse Activation Modulation for Frozen DiT-based Super-Resolution”. In: *arXiv preprint arXiv:2609.03813* (2026).

[56] Alec Radford et al. “Language models are unsupervised multitask learners”. In: *OpenAI blog* (2019).

[57] Rafael Rafailov et al. “Direct Preference Optimization: Your Language Model is Secretly a Reward Model”. In: *NeurIPS*. 2023.

[58] Subham Sekhar Sahoo et al. “Simple and Effective Masked Diffusion Language Models”. In: *NeurIPS*. 2024.

[59] Subham Sekhar Sahoo et al. “The Diffusion Duality”. In: *ICML*. 2025.

[60] Rico Sennrich, Barry Haddow, and Alexandra Birch. “Neural Machine Translation of Rare Words with Subword Units”. In: *ACL*. 2016.

[61] Noam Shazeer. “GLU Variants Improve Transformer”. In: *arXiv preprint arXiv:2002.05202* (2020).

[62] Jascha Sohl-Dickstein et al. “Deep Unsupervised Learning using Nonequilibrium Thermodynamics”. In: *ICML*. 2015.

[63] Jiaming Song, Chenlin Meng, and Stefano Ermon. “Denoising diffusion implicit models”. In: *ICLR*. 2021.

[64] Jianlin Su et al. “RoFormer: Enhanced Transformer with Rotary Position Embedding”. In: *Neurocomputing* (2024).

[65] DiffusionGemma Team et al. “DiffusionGemma Technical Report”. In: *arXiv preprint arXiv:2608.00146* (2026).

[66] Gemma Team et al. “Gemma 2: Improving Open Language Models at a Practical Size”. In: *arXiv preprint arXiv:2408.00118* (2024).

[67] Evelyn Turri et al. “Few Channels Draw The Whole Picture: Revealing Massive Activations in Diffusion Transformers”. In: *NeurIPS*. 2026.

[68] Ashish Vaswani et al. “Attention Is All You Need”. In: *NeurIPS*. 2017.

[69] Chengyue Wu et al. “Fast-dLLM: Training-free Acceleration of Diffusion LLM by Enabling KV Cache and Parallel Decoding”. In: *ICLR*. 2026.

[70] Chengyue Wu et al. “Fast-dLLM v2: Efficient Block-Diffusion LLM”. In: *ICLR*. 2026.

[71] Chengyue Wu et al. “Fast-dVLM: Efficient Block-Diffusion VLM via Direct Conversion from Autoregressive VLM”. In: *arXiv preprint arXiv:2604.06832* (2026).

[72] An Yang et al. “Qwen3 Technical Report”. In: *arXiv preprint arXiv:2505.09388* (2025).

[73] Ling Yang et al. “MMaDA: Multimodal Large Diffusion Language Models”. In: *NeurIPS*. 2025.

[74] Jiacheng Ye et al. “Beyond Autoregression: Discrete Diffusion for Complex Reasoning and Planning”. In: *ICLR*. 2025.

[75] Jiacheng Ye et al. “Dream 7B: Diffusion Large Language Models”. In: *arXiv preprint arXiv:2508.15487* (2025).

[76] Zebin You et al. “LLaDA-V: Large Language Diffusion Models with Visual Instruction Tuning”. In: *CVPR*. 2026.

[77] Yifan Yu et al. “Introspective Diffusion Language Models”. In: *arXiv preprint arXiv:2604.11035* (2026).

[78] Rowan Zellers et al. “HellaSwag: Can a Machine Really Finish Your Sentence?” In: *ACL*. 2019.

[79] Biao Zhang and Rico Sennrich. “Root Mean Square Layer Normalization”. In: *NeurIPS*. 2019.

[80] Fengqi Zhu et al. “LLaDA 1.5: Variance-Reduced Preference Optimization for Large Language Diffusion Models”. In: *ACL*. 2026.

[81] Fengqi Zhu et al. “LLaDA-MoE: A Sparse MoE Diffusion Language Model”. In: *arXiv preprint arXiv:2509.24389* (2025).

[82] Leonardo Zini et al. “vHector and HeisenVec: Scalable Vector Graphics Generation Through Large Language Models”. In: *NeurIPS*. 2025.