# LatentRecurrentDepthLM: An Open-Source Framework for Recurrent-Depth Language Models with Controllable Test-Time Compute

Ahsan Umar
GitHub: `codewithdark-git` Hugging Face: `codewithdark`

***Abstract*—LatentRecurrentDepthLM is a modular, production-ready open-source framework implementing a hybrid recurrent-depth language model that decouples effective reasoning depth from parameter count by iterating a single weight-shared block over a continuous latent state, enabling controllable test-time compute scaling without generating intermediate tokens or modifying model weights. Built in PyTorch with full Hugging Face Transformers compatibility, the framework provides end-to-end pipelines for dataset preparation, tokenization, training with randomized iteration depth and cosine scheduling, autoregressive generation with temperature and top-$k$ sampling, and one-command Hub deployment via a custom `PreTrainedModel` subclass. This paper documents the software architecture, core algorithms, training and inference workflows, practical use cases, and comparisons with related tools, connecting the framework to recent advances in recurrent-depth and latent reasoning research to serve researchers, educators, and practitioners exploring parameter-efficient sequence modeling. The repository (`codewithdark-git/LatentRecurrentDepthLM`) and Hugging Face model checkpoint are released under the MIT license.**

***Index Terms*—Recurrent-depth language models, latent reasoning, test-time compute scaling, PyTorch, Hugging Face Transformers, open-source software framework**

## I. Introduction

The development of modern natural language processing has been shaped fundamentally by the Transformer architecture introduced by Vaswani et al. [1]. By replacing sequential recurrence with parallelizable self-attention, Transformers overcame the gradient-vanishing and poor-hardware-utilization problems that had long limited earlier recurrent architectures such as Long Short-Term Memory (LSTM) networks [2]. The ability to attend over entire input contexts in parallel allowed the community to exploit modern accelerators fully, and a series of landmark scaling studies subsequently revealed a striking empirical regularity: model quality on diverse language tasks improves in a predictable, near-log-linear fashion as the number of trainable parameters, the volume of pre-training data, and the total compute budget are each increased [3], [4], [5]. This insight catalyzed a decade-long push toward ever-larger models, culminating in systems with hundreds of billions of parameters and emergent few-shot capabilities that were unimaginable a generation earlier [5].

Yet this *parameter-centric* scaling paradigm carries substantial and increasingly visible costs. Training a frontier-scale language model now requires tens of thousands of specialized accelerators operating for weeks or months, consuming energy at industrial scale. More fundamentally, once such a model is deployed its effective computational depth is permanently fixed: a 96-layer transformer always executes exactly 96 layers per token, regardless of whether the input is a simple factual lookup or a complex multi-step mathematical proof. Memory footprints grow linearly with depth, context windows must be extended at quadratic attention cost, and the inference-time compute budget is entirely determined by architectural choices made before training—not by the difficulty of the problem at hand. These structural rigidities motivate the search for architectures and inference strategies that allocate compute more adaptively.

A complementary paradigm has emerged under the name *test-time compute scaling*, which reallocates computational resources to the inference phase rather than concentrating them exclusively in pre-training. The most widely adopted incarnation of this idea is chain-of-thought (CoT) prompting [6], in which a model is encouraged to generate explicit intermediate reasoning tokens before committing to a final answer. CoT and its descendants—scratchpad reasoning, process reward models, and reinforcement-learning-from-human-feedback on reasoning traces—have produced impressive gains on mathematical, logical, and multi-step reasoning benchmarks. However, these token-based methods carry inherent overheads. Every intermediate reasoning token occupies a slot in the context window, inflates the key-value cache, adds decoding latency, and can require expensive supervised fine-tuning on curated reasoning demonstrations to elicit reliably. More subtly, externalizing cognition as natural-language tokens may not always be the most compact or accurate representation of multi-step reasoning: the intermediate text is trained to be human-readable, not necessarily to encode the most information-theoretically efficient intermediate representation.

An architecturally distinct and theoretically well-motivated alternative is *latent reasoning* achieved through *recurrent-depth* architectures, also referred to in the literature as looped, depth-recurrent, or iteratively refined models. The central idea is straightforward: instead of stacking $N$ distinct transformer blocks each with their own parameters, a single weight-shared computational block is applied $N$ times in succession, each pass refining a continuous hidden-state representation without emitting any intermediate tokens to the output stream [7], [8], [9], [10]. This design achieves two highly desirable properties

simultaneously. First, the total number of trainable parameters is independent of the number of refinement iterations: the model does not grow as $N$ increases. Second, $N$ is a free variable at inference time. A model trained with up to four iterations per forward pass can, without retraining, be queried with sixteen or fifty iterations, allocating more compute to harder problems and less to easier ones—a direct analogue of the adaptive thinking time that characterizes human expert reasoning.

The conceptual foundations of this paradigm trace back to the Universal Transformer (UT) of Dehghani et al. [7], which replaced the conventional fixed stack of heterogeneous transformer layers with repeated application of a single shared transformer block, augmented by a per-position adaptive halting mechanism based on Graves's ACT framework. Universal Transformers set new state-of-the-art results on the LAMBADA language-modeling task and on several algorithmic generalization benchmarks, demonstrating that recurrent inductive biases and the parallelism of attention are not mutually exclusive. This foundational result established that weight sharing across depth is not merely a parameter-saving trick but a qualitatively different inductive bias that favors systematic, composable computation.

Subsequent research has extended and scaled the paradigm substantially. Geiping et al. [8] revisited recurrent-depth language models in the era of modern hardware and large-scale data, introducing a 3.5-billion-parameter model that iterates a shared block in continuous latent space. When allowed to perform deeper unrolling at inference—up to compute budgets equivalent to a 50-billion-parameter static transformer—their model achieved dramatic improvements on mathematical reasoning (GSM8K, MATH), competitive programming (HumanEval), and general knowledge benchmarks, all without generating a single intermediate reasoning token. This empirical demonstration validated the core promise of latent test-time compute scaling at modern scale. Parallel lines of work have broadened the scope of applicability. Zhu et al. [9] introduced the Ouro family of Looped Language Models (LoopLM), pre-trained with an entropy-regularized objective that encourages the model to learn how much latent refinement each input requires; their 2.6-billion-parameter variant matches or surpasses models with up to 12 billion parameters on standard benchmarks. LoopFormer [11] explores elastic-depth training with shortcut modulation to make models robust across a range of compute budgets during both training and deployment. Depth-Recurrent Attention Mixtures (Dreamer) [12] combine recurrent-depth iteration with sparse expert routing to address the hidden-size bottleneck that can arise when a single shared block must represent the full diversity of layer-wise computations, reporting two-to-eight-fold gains in training token efficiency on reasoning benchmarks. Even beyond text, Recurrent-Depth Vision-Language-Action models [13] have applied iterative latent refinement in robotic control, achieving up to 80$\times$ inference speedup at constant memory usage compared to static-depth equivalents. Collectively, these works establish recurrent-depth iteration as a general-purpose mechanism for parameter-efficient scaling and implicit multi-step reasoning across modalities and domains.

Despite this accumulation of compelling evidence, the broader research and practitioner community has lacked accessible, production-ready open-source implementations that combine modern recurrent-depth architecture ideas with the tooling standards the community now expects: native compatibility with the Hugging Face ecosystem [14], simple training on custom datasets, reproducible configuration management, and frictionless model sharing. Most published works release code that is purpose-built for reproducing the specific large-scale pre-training experiments described in the accompanying paper. Such codebases are typically monolithic, poorly documented for external use, tightly coupled to specific infrastructure, and unsuitable for rapid ablation or modular experimentation. A lightweight, modular, and immediately usable implementation of the recurrent-depth paradigm has therefore been conspicuously absent from the open-source landscape.

The **LatentRecurrentDepthLM** repository (`codewithdark-git/LatentRecurrentDepthLM`) is designed to fill exactly this gap. It delivers a clean, self-contained, and fully Hugging Face-compatible open-source software framework for recurrent latent depth language modeling, released under the permissive MIT license. The framework is centered on a custom `PreTrainedModel` subclass that inherits the complete Hugging Face interface, enabling one-line model loading, controllable inference depth, direct upload to the Hugging Face Hub (model card available at `codewithdark/latent-recurrent-depth-lm`), and drop-in interoperability with the broader Transformers ecosystem [14]. The framework is built entirely in PyTorch [15] and imposes no heavy external dependencies beyond the standard scientific Python stack.

At its architectural core, LatentRecurrentDepthLM implements a three-stage hybrid design that separates initial token processing, iterative latent refinement, and final vocabulary projection into distinct, independently replaceable components.

The **PreludeBlock** (implemented in `prelude_Block.py`) performs the initial mapping from discrete token indices to continuous representations. Token embeddings are summed with learnable positional encodings that support sequence lengths up to 1 024 tokens, then passed through a single transformer layer consisting of multi-head self-attention and a feed-forward network, both wrapped in pre-norm residual connections. This single contextualization pass ensures that the subsequent recurrent loop operates on already-attended representations rather than raw embeddings, which improves convergence and representation quality.

The **RecurrentBlock** (in `recurrent_Block.py`), which is the heart of the architecture, maintains both a primary hidden state and a dedicated recurrent state vector. On each iteration, the recurrent state is projected through a learned linear layer and injected additively into the hidden state, after which multi-head attention and a feed-forward layer are applied with pre-norm residuals, and the updated hidden state is written back as the new recurrent state. Crucially, all iterations share the

same weights: the block is applied for `num_iterations` steps, where this count is randomly sampled per mini-batch during training (to build robustness to variable inference depths) and freely specified by the user at inference time. This mechanism realizes implicit multi-step latent reasoning entirely within continuous representation space, with no intermediate tokens emitted. The `LatentRecurrent` orchestration module (`latent_Recurrent.py`) wraps the block, initializes the recurrent state to zero at the start of each forward pass, and manages the iteration loop.

The **CodaBlock** (in `codaBlock.py`) applies a final layer normalization to the output of the last recurrent iteration and projects it linearly to vocabulary logits, keeping the decoding stage minimal so that all the useful computational depth resides in the recurrent stage.

This three-stage decomposition achieves true parameter efficiency: total trainable parameters depend only on the model dimension, number of attention heads, and vocabulary size—not on `num_iterations`. At the default configuration of $d_{\text{model}} = 768$ with 12 attention heads, the model contains approximately 93 million parameters, a footprint comparable to GPT-2 medium, yet it can simulate the effective depth of a much larger static model simply by iterating more at inference time.

Beyond the architectural core, LatentRecurrentDepthLM provides a complete end-to-end toolchain for practical experimentation and deployment. Data preparation and tokenization are handled by `dataset.py` and `tokenizer.py`, which wrap the Hugging Face Datasets library for general-purpose corpus loading with minimal configuration. The training engine (`trainer.py`) uses the AdamW optimizer, cosine annealing with warm-up, causal language-modeling loss, gradient clipping, and optional mixed-precision support via `torch.amp`. A high-level pipeline orchestrator (`pipeline.py`) and a command-line entry point (`train.py`) make end-to-end training fully scriptable with no manual PyTorch boilerplate. The `Inference/` directory contains ready-to-run scripts for full-sequence generation, single-token next-word prediction, and local checkpoint testing. Finally, `push_to_hub.py` provides one-command upload of trained checkpoints to the Hugging Face Hub with full custom-code compatibility, so community members can immediately use or fine-tune any shared model.

By packaging cutting-edge recurrent-depth techniques within a framework that satisfies modern open-source standards, LatentRecurrentDepthLM democratizes access to latent reasoning and test-time compute scaling. It is particularly valuable for three groups of users. Academic researchers benefit from the modular block design, which allows rapid ablation of attention variants, recurrent injection strategies, or halting mechanisms with changes to a single file. Educators and students find the clean Prelude-Recurrent-Coda decomposition an excellent conceptual entry point into depth-recurrent architectures. Practitioners and independent developers can load a pre-trained checkpoint from the Hub in one line, increase `num_iterations` to obtain qualitatively richer text with no parameter overhead, and deploy the result to resource-constrained environments where frontier-scale models are impractical.

This paper constitutes the definitive technical documentation and reference for the LatentRecurrentDepthLM framework as of February 2026. All architectural descriptions, pseudocode, and design commentary are grounded in the actual repository source files. No results or capabilities are claimed beyond what is demonstrable from the codebase. The remainder of this document is organized as follows. Section II surveys the related literature on recurrent-depth and looped architectures, test-time compute scaling, and open-source tooling. Section III describes the software at a high level, including the repository structure, installation procedure, and Hugging Face ecosystem integration. Sections IV and V detail the model architecture and core algorithms with full pseudocode. Section VI covers the training and inference pipelines. Section VII presents concrete use cases and code examples. Section VIII compares LatentRecurrentDepthLM with related open-source tools. Section IX discusses design strengths, acknowledged limitations, reproducibility practices, and directions for future development. Section X concludes the paper.

## II. Related Work

The design of LatentRecurrentDepthLM is grounded in a broad body of prior work spanning four interconnected areas: foundational sequence modeling and its limitations, recurrent and looped depth architectures, test-time compute scaling and latent reasoning, and the open-source tooling ecosystem that shapes how modern language model research is conducted and shared. This section surveys each area in turn, explaining how each line of research motivates specific design decisions in the framework and situating LatentRecurrentDepthLM within the landscape of existing work.

### A. Foundational Sequence Models and Their Limitations

The Transformer architecture introduced by Vaswani et al. [1] fundamentally changed how sequence modeling problems are approached. By replacing the sequential hidden-state updates of recurrent networks with fully parallelizable scaled dot-product attention, Transformers enabled training on hardware accelerators at scales that were simply unreachable for earlier recurrent architectures. The multi-head attention mechanism captures dependencies between any two positions in a sequence in a single layer, removing the vanishing-gradient bottleneck that had constrained the effective range of Long Short-Term Memory (LSTM) networks [2] and gated recurrent units to relatively short contexts in practice.

The practical consequences became clear quickly. A succession of scaling studies [3], [4] established that transformer-based language model performance improves in a predictable, near-log-linear fashion as the number of parameters, the volume of pre-training data, and the total compute budget are independently scaled. These empirical scaling laws provided the theoretical justification for the rapid growth in model size that characterized the years following the Transformer's

introduction, leading to models such as GPT-3 [5], which demonstrated that sufficiently large static transformers acquire strong few-shot task-solving abilities without any task-specific fine-tuning.

However, the successes of parameter-centric scaling have come with structural costs that become more acute as models grow. Inference compute is determined entirely by architectural choices made before training begins: a model with $L$ transformer layers always executes exactly $L$ layers per forward pass, regardless of whether the input is a straightforward factual query or a complex multi-step reasoning problem. Deeper models incur memory costs that grow linearly with depth and attention costs that grow quadratically with context length, making serving at scale expensive and inflexible. Earlier recurrent architectures such as LSTMs offered sequential processing with constant memory per step, but their inability to parallelize across time made them unsuitable for modern hardware, and their sequential inductive bias prevented the context-range improvements that attention achieves. Hybrid architectures that combine attention with forms of recurrence have since been proposed to capture the advantages of both [16], motivating the broader investigation of architectures that escape the fixed-depth constraint without sacrificing the parallelism of attention.

### *B. Recurrent and Looped Depth Architectures*

One of the most principled responses to the fixed-depth limitation is *weight sharing across depth*, the idea that instead of stacking $N$ independently parameterized transformer blocks, a single block with shared weights is applied $N$ times in succession. This keeps the parameter count constant as $N$ grows and endows the model with an inductive bias toward composable, iterative computation rather than memorization of layer-specific transformations.

The Universal Transformer (UT) of Dehghani et al. [7] was among the earliest demonstrations that this idea could outperform conventional stacked transformers. The UT applies a single shared transformer block repeatedly across a variable number of depth steps, augmented with a per-position Adaptive Computation Time (ACT) mechanism that learns when each token's representation has converged and no further refinement is necessary. On algorithmic generalization tasks such as copying and sorting, on the LAMBADA language modeling benchmark, and on machine translation, Universal Transformers matched or exceeded the performance of much deeper fixed-architecture baselines. The key insight was that recurrent inductive biases—the tendency to learn transformations that compose cleanly across iterations—are in fact complementary to the global receptive field provided by attention, rather than in tension with it.

The theoretical grounding for this intuition was later strengthened by Merrill et al. [17], who characterized the computational expressivity of transformers with shared parameters and showed that looped architectures can simulate classes of computation that require circuits of depth proportional to the number of iterations, even when the parameter count is held constant. Related work by Saunshi et al. [18] provided empirical evidence that looped transformers trained on reasoning tasks develop stronger inductive biases for systematic generalization than depth-matched non-looped counterparts, and offered preliminary theoretical explanations in terms of the gradient flow structure induced by weight sharing.

The most direct empirical precedent for LatentRecurrentDepthLM at modern scale is the recurrent-depth language model introduced by Geiping et al. [8]. Their 3.5-billion-parameter model iterates a single shared transformer block in continuous latent space, with `num_iterations` controllable at both training and inference time. When allowed to unroll to compute budgets equivalent to a 50-billion-parameter static transformer, the model achieved substantial improvements on GSM8K, MATH, HumanEval, and MMLU—all without generating intermediate reasoning tokens. Critically, the authors demonstrated that training with randomly sampled iteration counts improves robustness to inference-time depth changes, a finding that directly informs the randomized-iteration training strategy implemented in LatentRecurrentDepthLM.

Parallel work has explored variants of the looped architecture that address limitations of the basic weight-sharing scheme. The Ouro family of Looped Language Models by Zhu et al. [9] integrates iterative latent computation directly into pre-training through an entropy-regularized objective that trains the model to allocate depth adaptively based on input complexity. Ouro 1.4B and 2.6B models, trained on 7.7 trillion tokens, match or surpass models with up to 12 billion parameters on standard benchmarks including MMLU, HellaSwag, and ARC. The authors attribute these gains primarily to superior knowledge manipulation and compositional reasoning rather than raw capacity, consistent with the inductive bias argument. Pappone et al. [10] introduced a two-scale training framework for recurrent-depth models that decouples updates to the shared recurrent block from updates to the embedding and decoding layers, improving stability when training with large numbers of iterations.

LoopFormer [11] addresses the practical challenge of deploying recurrent-depth models under variable compute constraints by introducing elastic-depth training with shortcut modulation. During training, the model learns to produce useful outputs after any number of iterations, not just the maximum, enabling deployment across a spectrum of compute budgets without retraining. This is complementary to the randomized-iteration training used in LatentRecurrentDepthLM, and represents a natural direction for future extension.

Depth-Recurrent Attention Mixtures, or Dreamer [12], identify and address a structural limitation of basic looped architectures: when a single shared block must represent the computational roles of many distinct layers, the hidden-state bottleneck can prevent the model from expressing the full diversity of transformations needed across iterations. Dreamer combines recurrent-depth iteration with sparse mixture-of-experts routing, assigning different expert subnetworks to different iteration depths while keeping the routing overhead minimal. This achieves two-to-eight-fold improvements in training token efficiency across reasoning benchmarks and

motivates future extensions of LatentRecurrentDepthLM with expert routing.

The recurrent-depth paradigm has also proven effective beyond text. The Recurrent-Depth Vision-Language-Action (VLA) model [13] applies iterative latent refinement to robotics control, where the model must integrate visual observations, language instructions, and action history to produce motor commands. By iterating a shared block rather than decoding through a deep static stack, the architecture achieves up to 80× inference speedup relative to comparable static-depth VLAs while maintaining constant memory usage—a particularly compelling demonstration that the benefits of latent recurrence generalize across input modalities and output types.

Further refinements documented in the literature include efficient parallel samplers for recurrent-depth models [19], which address the sequential dependency between iterations in autoregressive decoding; techniques for teaching pre-trained static-depth models to perform deeper thinking through post-hoc recurrence adaptation [20], which suggests recurrent-depth inference can be retrofitted onto existing checkpoints; and analyses of emergent latent chain-of-thought structures in depth-recurrent transformers [21], which provide interpretability evidence that iterated hidden states develop structured, semantically meaningful intermediate representations rather than arbitrary attractors. Together, these works paint a rich and converging picture of recurrent-depth as a mature and versatile paradigm for parameter-efficient scaling and implicit multi-step reasoning.

### C. Test-Time Compute Scaling and Latent Reasoning

The broader question of how to scale compute at inference time rather than solely at training time has attracted significant attention across the community. The most widely adopted approach, chain-of-thought (CoT) prompting [6], instructs a language model to generate explicit intermediate reasoning steps as natural-language tokens before producing its final answer. CoT and its variants—including scratchpad prompting, least-to-most prompting, and reinforcement-learning-based process reward models—have produced substantial gains on mathematical, logical, and commonsense reasoning benchmarks and have been integrated into production systems.

However, token-based test-time scaling has fundamental overheads that become more pronounced as reasoning chains grow longer. Every intermediate token occupies space in the context window, contributes to the autoregressive key-value cache, adds to decoding latency, and must be consistent with the model's language distribution in ways that may not align with the most information-efficient intermediate representation of the reasoning process. Moreover, eliciting reliable CoT often requires fine-tuning on curated reasoning demonstrations, which are expensive to produce and may introduce distribution shift relative to the target task.

Latent reasoning through recurrent depth addresses these limitations by confining all intermediate computation to continuous hidden-state space. No intermediate tokens are generated; the model's reasoning is implicit in the sequence of hidden-state refinements produced across iterations. Geiping et al. [8] formalized this paradigm and demonstrated empirically that scaling the number of latent iterations at inference can match or exceed the performance gains obtained by generating long token-based reasoning chains, at a fraction of the output-length cost. Zhu et al. [9] showed further that models pre-trained with latent iteration develop representations whose intermediate states are more causally predictive of the final output than the intermediate tokens of CoT-trained models, suggesting that latent reasoning is not merely a compute-equivalent substitute for CoT but a qualitatively different and in some respects more direct form of multi-step computation.

Theoretical connections between looped architectures and continuous-depth dynamical systems have been explored by several authors. Merrill et al. [17] establish expressivity results showing that parameter-shared depth-recurrent models can represent computations that require depth scaling with the problem size, which fixed-depth models cannot achieve without proportional parameter growth. These results provide a formal basis for the empirical observation that increasing inference-time iterations yields qualitative improvements in reasoning, not merely quantitative refinements of the same computation.

### D. Open-Source Implementations and Ecosystem Tools

The Hugging Face Transformers library [14] has become the dominant standard for sharing, loading, and deploying language models in both research and production settings. Its `PreTrainedModel` base class provides a principled interface for custom architectures, enabling seamless `from_pretrained` loading, `generate` API compatibility, automatic device placement, mixed-precision support, and one-command upload to the Hugging Face Hub. PyTorch [15] provides the underlying tensor computation and automatic differentiation framework that nearly all modern implementations rely upon. Together, these two libraries define the de-facto ecosystem standard for open-source language model research.

Despite the growing research literature on recurrent-depth architectures, the intersection of this area with modern ecosystem standards remains sparsely populated. The original Universal Transformer codebase was implemented in TensorFlow and designed for the specific experimental setup of the 2018 paper, without Hugging Face integration or support for custom datasets. The research codebase accompanying Geiping et al. [8] is the most direct large-scale implementation of the latent recurrent depth paradigm, but is designed for multi-GPU pre-training at scale and is not structured for modular experimentation or lightweight single-GPU deployment. Similarly, the codebases accompanying Ouro [9] and LoopFormer [11] target reproduction of specific large-scale training experiments rather than providing reusable, general-purpose frameworks.

General-purpose alternatives such as the RWKV family [22] and Mamba [23] offer efficient long-context inference through state-space model (SSM) formulations that achieve linear time and constant memory per generated token. Both projects provide well-engineered Hugging Face-compatible

implementations and are strong alternatives for applications where linear-time autoregressive inference is the primary requirement. However, they differ fundamentally from recurrent-depth architectures in their design goals: RWKV and Mamba use implicit state evolution to extend context, not explicit multi-step latent iteration to deepen reasoning. They do not expose a controllable iteration count that trades inference compute for reasoning quality, and are therefore not substitutes for the paradigm that LatentRecurrentDepthLM implements.

LatentRecurrentDepthLM occupies a distinct position in this landscape. It is, to the authors' knowledge, one of the few open-source repositories that simultaneously provides a fully Hugging Face-native recurrent-depth language model, explicit and user-controllable inference-time iteration depth, complete training pipelines for custom datasets, and a lightweight enough default configuration to run on consumer-grade hardware. Its MIT license, modular block design, and emphasis on reproducibility and documentation align with the principles articulated in best-practice guides for open machine learning software **arora2021opportunities**. By lowering the barrier to entry for recurrent-depth experimentation, the framework complements the large-scale research codebases in the literature rather than competing with them, serving researchers who need a testbed for ablation studies, educators who need a pedagogically clear implementation, and practitioners who need a deployable starting point.

The remainder of this paper describes precisely how LatentRecurrentDepthLM translates the architectural insights reviewed in this section into production-ready, well-engineered code.

## III. Software Description

LatentRecurrentDepthLM is a lightweight, fully modular, and production-ready open-source software framework implemented in pure Python 3 using the PyTorch deep learning library [15]. The framework is designed specifically for researchers, educators, and practitioners who wish to experiment with, extend, or deploy recurrent-depth language models without confronting the infrastructure complexity that characterizes large-scale research codebases. Rather than providing isolated model code, it offers a complete end-to-end toolchain that spans data preparation and tokenization, model instantiation and configuration management, training with a variable-depth curriculum, controllable-depth autoregressive inference, and one-command deployment to the Hugging Face Hub. The software is released under the permissive MIT license, which imposes no restrictions on commercial or academic use, modification, or redistribution, and thereby encourages broad adoption and community contribution.

The repository is organized around two guiding engineering principles. The first is *minimal surface area*: every file and directory has a single, clearly stated responsibility, and no functionality is duplicated across modules. This makes the codebase easy to navigate for a newcomer and easy to modify safely for an experienced developer. The second is *zero mandatory infrastructure*: the framework requires only PyTorch and the Hugging Face Transformers and Datasets libraries, both of which install trivially on any Python environment. No distributed computing cluster, proprietary dataset, or pre-downloaded model weight is required to begin experimenting. The official pre-trained checkpoint is hosted publicly on the Hugging Face Hub at `codewithdark/latent-recurrent-depth-lm` and can be loaded with a single Python call, allowing immediate inference without local training. As of the latest commit (23 total commits), the codebase remains concise yet comprehensive, deliberately favoring readability and extensibility over premature optimization.

### A. Repository Organization

The project structure is intentionally transparent and self-documenting. A flat root directory holds all high-level scripts that a user interacts with directly—data loading, training, Hub upload—while deeper architectural logic is encapsulated within the `Model/` subdirectory. Inference examples live in a dedicated `Inference/` subdirectory, keeping concerns cleanly separated. Table I provides a complete map of the repository's key files and their roles.

The `Model/` directory is the architectural heart of the framework. It contains six focused modules, each implementing exactly one component of the recurrent-depth pipeline. `model.py` defines the top-level `LatentRecurrentDepthLM` class, which inherits from Hugging Face's `PreTrainedModel` and exposes the full model to the ecosystem. `latent_Recurrent.py` implements the `LatentRecurrent` orchestration wrapper, which initializes the recurrent state and manages the iteration loop. `recurrent_Block.py` defines the `RecurrentBlock`, the weight-shared computational unit that is called on every iteration. `prelude_Block.py` implements the `PreludeBlock`, responsible for the initial token embedding and first contextualization layer. `codaBlock.py` provides the `CodaBlock`, which applies final normalization and projects hidden states to vocabulary logits. Finally, `multi_head_Attention.py` contains the shared multi-head attention implementation used by both the PreludeBlock and RecurrentBlock. This one-class-per-file organization directly enforces the single-responsibility principle: a researcher who wishes to substitute a different attention kernel, modify the recurrent injection strategy, or add an adaptive halting mechanism can locate and edit the relevant file in seconds, with no risk of unintended side effects elsewhere in the codebase.

The `Inference/` directory contains three ready-to-run scripts that cover the most common practical usage patterns: full-sequence generation with configurable prompt and depth, single-token next-word prediction for interactive debugging, and a local checkpoint loader for testing a freshly trained model before Hub upload. These scripts are intentionally self-contained so that they serve as working documentation as well as functional tools.

TABLE I
REPOSITORY FILE AND DIRECTORY STRUCTURE

| Component | Purpose |
|---|---|
| `dataset.py` | Loads and preprocesses custom text corpora; supports the Hugging Face `Dataset` format |
| `tokenizer.py` | Wrapper around Hugging Face tokenizers with vocabulary handling and special token management |
| `trainer.py` | Core training engine: causal LM loss, AdamW optimization, cosine annealing, gradient clipping |
| `pipeline.py` | High-level orchestration of data loading, model instantiation, training loop, and evaluation |
| `train.py` | Command-line entry point for launching training runs with argument parsing |
| `push_to_hub.py` | Uploads trained checkpoints to the Hugging Face Hub with full custom-code compatibility |
| `Model/` | Contains all six architectural modules (detailed in Section IV) |
| `Inference/` | Ready-to-run generation scripts for testing and production inference |
| `README.md` | Installation guide, usage examples, and architectural overview |

### B. Installation and Dependencies

Installation follows standard open-source Python conventions and is designed to be fully reproducible on any CUDA-capable workstation or CPU-only machine. The recommended procedure is as follows. First, clone the repository with https://github.com/codewithdark-git/LatentRecurrentDepthLM.git. Second, create and activate a dedicated virtual environment to avoid dependency conflicts with other projects. Third, install PyTorch with the appropriate CUDA backend: for CUDA 12.1, the command is `pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu121`; for CPU-only environments, the standard PyPI wheel suffices. Fourth, install the Hugging Face stack with `pip install transformers datasets accelerate`. Optionally, `matplotlib` may be installed for automatic loss-curve visualization during training.

The total dependency footprint is deliberately minimal. Beyond PyTorch and the Hugging Face libraries, no specialized package is required. This deliberate choice keeps the framework deployable in constrained environments, including university computing clusters with locked package managers, cloud instances with limited disk quotas, and edge devices where heavyweight dependencies are impractical. One current limitation worth noting is the absence of a `requirements.txt` file, which means that dependency versions are not pinned and reproducibility relies on the user selecting compatible library versions. Adding a versioned requirements file is a straightforward and recommended community contribution that would eliminate this gap.

### C. Hugging Face Ecosystem Integration

A defining feature of LatentRecurrentDepthLM, and one that distinguishes it sharply from most research-oriented recurrent-depth codebases, is its first-class integration with the Hugging Face Transformers library [14]. This integration is not superficial. The top-level model class inherits directly from `PreTrainedModel` and satisfies the full contract that class imposes, meaning the model participates seamlessly in the entire Hugging Face ecosystem without any adapter or wrapper layer.

At the configuration level, the framework provides a custom `LatentRecurrentDepthConfig` class that subclasses `PretrainedConfig`. This class manages all hyperparameters—model dimension `d_model`, number of attention heads `num_heads`, vocabulary size `vocab_size`, dropout rate, and maximum sequence length—through a unified, serializable configuration object. Every training run writes this configuration object alongside the model weights, so any checkpoint is fully self-describing: another user loading the checkpoint does not need to remember or reconstruct the hyperparameters used during training.

At the model level, the framework supports both `from_pretrained` and `save_pretrained` with automatic custom-code handling. Because the model class is not part of the standard Transformers library, users load it by passing `trust_remote_code=True` to the loading call, after which Hugging Face downloads and executes the custom model code transparently. This mechanism is the standard pattern for community models that extend the base library, and LatentRecurrentDepthLM adopts it without modification.

At the inference level, the model registers itself with `AutoModelForCausalLM`, so it can be loaded with the standard one-line pattern used for any causal language model in the ecosystem. The `generate` API is fully supported, with the additional `num_iterations` argument passed through to the forward method to control recurrent depth on a per-call basis. This means that a user can switch between shallow, fast generation and deep, reasoning-intensive generation simply by changing a single integer argument, with no model reload or weight modification required.

The pre-trained checkpoint for the default 93-million-parameter configuration is hosted at `codewithdark/latent-recurrent-depth-lm` on the Hugging Face Hub. Any user with a Hugging Face account and an internet connection can load this checkpoint, run inference, or continue fine-tuning on a custom corpus, entirely from within a standard Python environment and without downloading or managing raw model files manually.

### D. Key Software Features and Design Practices

Several design practices permeate the codebase and collectively account for its suitability as both a research platform and a practical deployment tool.

**Modularity and single-responsibility design.** Every architectural component is implemented as a standalone

nn.Module with a clearly bounded interface. The attention module, recurrent block, prelude layer, and coda layer each live in their own file and expose only the inputs and outputs their role requires. A researcher who wishes to replace standard scaled dot-product attention with FlashAttention-2 **dao2022flashattention** for speed, or with a linear attention variant for long-sequence efficiency, needs to modify only multi_head_Attention.py and leave all other components untouched. Similarly, experimenting with multiplicative or gated recurrent state injection requires a single-line change in recurrent_Block.py. This modularity is not incidental; it is a deliberate architectural choice that prioritizes ablation speed and experimental flexibility over code minimality.

**Reproducibility and configuration management.** Reproducibility in machine learning software requires more than setting a random seed. LatentRecurrentDepthLM addresses reproducibility at multiple levels. All hyperparameters are centralized in the LatentRecurrentDepthConfig object, which is serialized alongside every checkpoint. The randomized num_iterations sampling during training uses PyTorch's random number generation, which is controlled by torch.manual_seed and can therefore be made deterministic across runs. Gradient clipping is applied at a fixed norm threshold of 1.0 in every training step, preventing loss spikes that would otherwise make runs hard to reproduce. Loss curves are saved automatically using matplotlib, providing a lightweight training record without requiring a separate experiment-tracking system.

**Controllable test-time compute.** The num_iterations parameter is the primary user-facing control knob for the framework's core capability. During training it is sampled uniformly from a small integer range per mini-batch, which exposes the shared recurrent block to a distribution of effective depths and encourages it to produce useful outputs regardless of how many iterations it is given at inference time—a technique motivated directly by findings in the recurrent-depth literature [8]. At inference time the parameter is specified freely by the user for each generation call. This design enables a natural compute-quality trade-off: a deployment with a tight latency budget can run with four iterations, while a high-stakes reasoning task can use sixteen or more, all with the same loaded model and the same memory footprint.

**Training pipeline completeness.** The Trainer class provides causal language-modeling loss computed over shifted token sequences, the AdamW optimizer with configurable $\beta$ coefficients and weight decay, a cosine annealing learning-rate schedule with optional linear warm-up, gradient norm clipping, and optional mixed-precision training through torch.amp. The higher-level pipeline.py orchestrator handles dataset loading, train/validation splitting, DataLoader construction with appropriate padding collators, and checkpoint saving at configurable intervals. Together these components mean that a user who supplies a Hugging Face dataset identifier and a configuration dictionary can run a complete training experiment with a single script invocation and no manual PyTorch boilerplate.

**Inference efficiency.** Autoregressive generation reuses the exact same forward pass that is used during training. The model is placed in evaluation mode and wrapped in torch.no_grad() for efficiency. CUDA availability is detected automatically and the model is moved to the appropriate device. The attention module's architecture is compatible with future integration of key-value caching, which would reduce the per-step cost of generation from $O(L^2)$ to $O(L)$ in sequence length—a natural extension for longer-context deployment.

**Extensibility and contribution path.** The MIT license and the clean module boundaries together create a welcoming environment for community extensions. The most natural near-term additions—adaptive halting mechanisms, FlashAttention integration, requirements.txt pinning, unit tests, and distributed training support via Accelerate—each map cleanly onto one or two files without requiring architectural redesign. The repository's GitHub Issues and Pull Request workflow provides the standard contribution path.

These practices collectively align the framework with established recommendations for open-source machine learning software quality [24], including emphasis on documentation, testability, and community accessibility. The default 93-million-parameter configuration is deliberately sized to run comfortably on a single consumer-grade GPU with 8 GB of VRAM, making the framework accessible to independent researchers and students who do not have access to institutional computing clusters. By providing complete pipelines rather than isolated model code, LatentRecurrentDepthLM removes the most common practical barriers to exploring recurrent-depth architectures and positions itself as a genuine community resource rather than a one-off research release.

The subsequent sections build on this software foundation. Section IV describes the model architecture in full detail, explaining the role and mathematical operations of each block. Section V provides formal pseudocode for the forward pass and generation algorithm. Section VI covers the training and inference pipelines in depth. Section VII presents concrete use cases and runnable code examples.

## IV. Architecture and Design

The core innovation of LatentRecurrentDepthLM is a clean three-stage modular architecture that fuses classical transformer components with iterative latent-state refinement. By sharing all weights across depth iterations, the design achieves true parameter efficiency: the total number of trainable parameters is fixed at model creation and does not grow with the number of recurrent steps applied during a forward pass. At the same time, the effective computational depth—and therefore the model's implicit reasoning capacity—remains freely adjustable at both training and inference time. The architecture maps directly onto the six files of the Model/ directory: prelude_Block.py, recurrent_Block.py, latent_Recurrent.py, codaBlock.py, multi_head_Attention.py, and model.py.

The top-level class LatentRecurrentDepthLM inherits from Hugging Face's PreTrainedModel and accepts four

primary hyperparameters: vocabulary size, hidden dimension `d_model` (default 768), number of attention heads (default 12), and dropout rate (default 0.1). Its forward method receives `input_ids` of shape $(B, L)$, a positive integer `num_iterations` that controls recurrent depth, and an optional attention mask. The three conceptual stages executed by this method are the Prelude, which maps discrete tokens to contextualized continuous representations; the Latent Recurrent Refinement, which iterates a weight-shared block to refine those representations in latent space; and the Coda, which projects the final hidden state to vocabulary logits. This decomposition preserves end-to-end differentiability while allowing each stage to be modified or replaced independently.

### A. Prelude Block

The `PreludeBlock` (in `prelude_Block.py`) maps input token indices to continuous hidden representations and applies a first round of contextualization before the recurrent loop begins. Token embeddings are summed with learnable positional encodings of shape $(1, 1024, d_{\text{model}})$ truncated to the actual sequence length, giving the model awareness of token position without hard-coding a sinusoidal pattern. A single transformer layer—consisting of pre-norm multi-head self-attention followed by a pre-norm two-layer GELU feed-forward network with expansion ratio 4—then contextualizes these representations before they enter the recurrent stage. The forward equations are:

$$\mathbf{E} = \text{Embedding}(x) + \text{PosEncoding}_{:,:L} \tag{1}$$

$$\mathbf{A} = \text{MHA}(\text{LN}(\mathbf{E}), \text{mask}) \tag{2}$$

$$\mathbf{H}_1 = \mathbf{E} + \mathbf{A} \tag{3}$$

$$\mathbf{F} = \text{FFN}(\text{LN}(\mathbf{H}_1)) \tag{4}$$

$$\mathbf{H}_{\text{out}} = \mathbf{H}_1 + \mathbf{F} \tag{5}$$

Pre-norm residuals are used throughout, consistent with the stabilization practices adopted by GPT-style and LLaMA-family architectures. By giving the recurrent block already-attended representations as input, the Prelude ensures that even a single recurrent iteration yields meaningful contextual output.

### B. Recurrent Block and Latent State Management

The `RecurrentBlock` (in `recurrent_Block.py`), orchestrated by `LatentRecurrent` (in `latent_Recurrent.py`), is the heart of the architecture. All iterations share identical weights, so the parameter count is independent of depth. The block maintains an explicit recurrent state $\mathbf{S} \in \mathbb{R}^{B \times L \times D}$, which the `LatentRecurrent` wrapper initializes to zero at the start of every forward pass and updates after each iteration. The per-iteration operations are:

$$\mathbf{S}' = \text{Linear}_{\text{proj}}(\mathbf{S}) \tag{6}$$

$$\mathbf{X} = \mathbf{H} + \mathbf{S}' \tag{7}$$

$$\mathbf{A} = \text{MHA}(\text{LN}(\mathbf{X}), \text{mask}) \tag{8}$$

$$\mathbf{F} = \text{FFN}(\text{LN}(\mathbf{X})) \tag{9}$$

$$\mathbf{H}' = \mathbf{X} + \mathbf{A} + \mathbf{F} \tag{10}$$

$$\mathbf{S}'_{\text{new}} = \mathbf{H}' \tag{11}$$

The recurrent state is first projected through a learned linear layer and then injected additively into the current hidden state. This lets information accumulated in previous iterations influence the current one entirely within continuous latent space, without emitting any intermediate tokens. After attention and feed-forward processing with pre-norm residuals, the updated hidden state is written back as the new recurrent state, closing the loop. The `LatentRecurrent` wrapper runs this block for exactly `num_iterations` steps and returns the final hidden state, enabling zero-cost depth scaling at inference time: a model trained with up to four iterations can be queried with sixteen or more without any weight change.

### C. Coda Block

The `CodaBlock` (in `codaBlock.py`) is intentionally minimal. It applies a final layer normalization followed by a linear projection to vocabulary size:

$$\mathbf{L} = \text{Linear}(\text{LN}(\mathbf{H}_{\text{final}})) \tag{12}$$

No additional attention or recurrence appears here. Keeping the Coda lightweight concentrates all the computational depth of the model within the recurrent stage, where it contributes directly to representation refinement.

### D. Multi-Head Attention

The shared `MultiHeadAttention` module (in `multi_head_Attention.py`) implements canonical scaled dot-product attention with support for both padding masks and causal masks. Using the same attention implementation in the Prelude and the RecurrentBlock keeps the parameter budget consistent and simplifies the extension path: replacing this module with FlashAttention-2 **dao2022flashattention** or a linear-attention variant requires editing a single file.

### E. Summary of Design Decisions

The architecture reflects several deliberate engineering choices. Parameter efficiency is achieved by conditioning effective depth entirely on the runtime integer `num_iterations` rather than on model size; at the default $d_{\text{model}} = 768$ with 12 heads, the model holds approximately 93 million parameters regardless of how many iterations are used. Modularity is enforced by the one-class-per-file organization, so any block can be swapped or extended without touching the rest of the codebase. Pre-norm residuals improve training stability at scale. Full `PreTrainedModel` compliance provides seamless `from_pretrained`, `generate`, and Hub integration. Table II summarizes each block and its computational role.

## V. Algorithm and Implementation Overview

This section provides a precise, implementation-grounded description of the three core algorithms in LatentRecurrentDepthLM. All pseudocode is derived directly from the actual source files in the `Model/` directory and presented using the `algorithm2e` environment. For each algorithm we describe inputs and outputs, explain key design decisions, and state time and space complexity where relevant.

TABLE II
ARCHITECTURAL BLOCKS AND RESPONSIBILITIES

| Block | File | Key Operations |
|---|---|---|
| Prelude | `prelude_Block.py` | Embedding + positional encoding + MHA + FFN (1 layer) |
| Recurrent | `recurrent_Block.py` | State projection + injection + MHA + FFN (weight-shared) |
| Latent Orchestrator | `latent_Recurrent.py` | Zero-initialize state; loop `num_iterations` times |
| Coda | `codaBlock.py` | Final layer norm + linear vocabulary projection |
| Attention | `multi_head_Attention.py` | Scaled dot-product attention with optional masking |

**Algorithm 1** Forward Pass of LatentRecurrentDepthLM

**Input:** $x \in \mathbb{Z}^{B\times S}$ (input token indices), $N \in \mathbb{N}^{+}$ (num_iterations), optional mask $M$
**Output:** logits $\mathbf{L} \in \mathbb{R}^{B\times S\times V}$

$\mathbf{H} \leftarrow$ `PreludeBlock`$(x, M)$ `// token embedding + positional encoding + one MHA+FFN layer`
$\mathbf{S} \leftarrow \mathbf{0}^{B\times S\times D}$ `// zero-initialize recurrent state`
**for** $i \leftarrow 1$ **to** $N$ **do**
  $\mathbf{H}, \mathbf{S} \leftarrow$ `RecurrentBlock`$(\mathbf{H}, \mathbf{S}, M)$
$\mathbf{L} \leftarrow$ `CodaBlock`$(\mathbf{H})$ `// final layer norm + linear projection to vocabulary`
**return** $\mathbf{L}$

**Algorithm 2** RecurrentBlock Forward (one iteration)

**Input:** $\mathbf{H} \in \mathbb{R}^{B\times S\times D}$ (hidden state), $\mathbf{S} \in \mathbb{R}^{B\times S\times D}$ (recurrent state), optional mask $M$
**Output:** updated hidden $\mathbf{H}'$, updated recurrent state $\mathbf{S}'$

$\mathbf{S}' \leftarrow \mathrm{Linear}_{\mathrm{proj}}(\mathbf{S})$ `// project recurrent state`
$\mathbf{X} \leftarrow \mathbf{H} + \mathbf{S}'$ `// inject into hidden state`
$\mathbf{A} \leftarrow \mathrm{MHA}(\mathrm{LN}_1(\mathbf{X}), M)$ `// pre-norm self-attention`
$\mathbf{F} \leftarrow \mathrm{FFN}(\mathrm{LN}_2(\mathbf{X}))$ `// pre-norm GELU FFN, expansion ratio 4`
$\mathbf{H}' \leftarrow \mathbf{X} + \mathbf{A} + \mathbf{F}$ `// residual connections`
$\mathbf{S}' \leftarrow \mathbf{X}$ `// update recurrent state with pre-FFN hidden`
**return** $\mathbf{H}', \mathbf{S}'$

**Algorithm 3** Autoregressive Generation (HF-Compatible)

**Input:** prompt $p \in \mathbb{Z}^{B\times S_0}$, max_length $L$, num_iterations $N$, temperature $T$, top_k $K$
**Output:** generated sequence $y \in \mathbb{Z}^{B\times L}$

$y \leftarrow p$ **while** $\mathrm{length}(y) < L$ **do**
  $\mathbf{L}_{\mathrm{logits}} \leftarrow$ `Model.forward`$(y, N)$ `// full forward pass with` $N$ `recurrent iterations`
  $l \leftarrow \mathbf{L}_{\mathrm{logits}}[:, -1, :] / T$ `// extract last-position logits, apply temperature`
  **if** $K \neq$ *None* **then**
    Apply top-$K$ filtering to $l$
  $t \sim \mathrm{softmax}(l)$ `// sample next token`
  $y \leftarrow \mathrm{concat}(y, t)$ **if** $t = \mathrm{EOS}$ **then break**;
**return** $y$

### A. Core Forward Pass

The primary computation is orchestrated by `LatentRecurrentDepthLM.forward`, which sequences the three architectural stages (Algorithm 1).

`input_ids` is a long integer tensor of shape $(B, S)$ with $S \leq 1024$. `num_iterations` is sampled randomly per mini-batch during training and set freely by the user at inference. The zero-initialized recurrent state gives the model a neutral starting point on every call; each subsequent iteration refines the representation in latent space without emitting any output tokens [8].

### B. Recurrent Block (Weight-Shared Core)

Algorithm 2 shows the single-iteration logic of `RecurrentBlock.forward`.

The recurrent state is written back as the pre-FFN hidden state $\mathbf{X}$ rather than the post-FFN output $\mathbf{H}'$, a deliberate choice that produces cleaner cross-iteration information flow as observed in the source. Per-iteration computational cost is $O(B \cdot S^2 \cdot D)$ for attention and $O(B \cdot S \cdot D^2)$ for the FFN, giving a total forward cost of $O(N \cdot B \cdot S^2 \cdot D + N \cdot B \cdot S \cdot D^2)$ for $N$ iterations. Crucially, the parameter count is $O(D^2 + V \cdot D)$ and is independent of $N$, so increasing test-time depth incurs compute cost but no memory overhead from additional weights.

### C. Autoregressive Generation with Controllable Depth

Algorithm 3 shows the generation loop exposed through the Hugging Face `generate` API.

The model is placed in `eval()` mode and wrapped in `torch.no_grad()` before generation. Top-$k$ filtering and temperature scaling use efficient PyTorch primitives (`torch.topk`, `F.softmax`, `torch.multinomial`). The value of $N$ can in principle vary across decoding steps, providing a natural hook for future adaptive-compute strategies where harder positions receive more iterations.

### D. Additional Implementation Notes

Both the Prelude and Recurrent blocks forward the attention mask to `MultiHeadAttention`, supporting padding masks and causal masks interchangeably. During training, `num_iterations` is sampled uniformly per mini-batch (e.g., from $\{1, 2, 3, 4\}$), which trains the shared block to produce useful representations at any depth and improves

robustness when higher iteration counts are used at inference [8]. Full `torch.amp` automatic mixed-precision and CPU execution are supported without code changes. The modular class structure means that replacing `MultiHeadAttention` with FlashAttention-2 or adding a learned halting mechanism requires modifying at most one or two files, with all pipelines remaining functional.

These algorithms faithfully implement the recurrent latent depth paradigm while maintaining production-grade efficiency and full Hugging Face compatibility. The following section covers the complete training and inference pipelines that build on this algorithmic foundation.

## VI. Training and Inference Pipelines

LatentRecurrentDepthLM provides complete, production-ready pipelines for both training and inference, enabling end-to-end experimentation with minimal configuration. The training workflow is orchestrated through `pipeline.py` and `trainer.py`, while inference is supported via native Hugging Face `generate` integration and dedicated scripts in the `Inference/` directory. All components emphasize reproducibility, modularity, and ease of use, aligning with modern open-source machine learning best practices [24].

### A. Training Pipeline

The training pipeline begins with data preparation in `dataset.py`, which loads and tokenizes text corpora using the Hugging Face Datasets library. The repository demonstrates training on the first 1 000 samples of WikiText-2-raw-v1, a standard language-modeling benchmark, but the code is fully general and accepts any dataset that the library supports. Tokenization is handled by `tokenizer.py`, which wraps Hugging Face tokenizers and manages vocabulary, padding, and special tokens consistently.

The core training logic resides in the `Trainer` class (`trainer.py`). Default hyperparameters are summarized in Table III. A critical research-aligned feature of the training loop is randomized `num_iterations` per mini-batch: during each step the iteration count is sampled uniformly from a small range (e.g., $\{1, 2, 3, 4\}$), exposing the shared recurrent block to a distribution of effective depths. This trains the model to produce useful representations at any depth and allows it to generalize gracefully to higher iteration counts at inference time without retraining [8]. Algorithm 4 shows the complete training step.

High-level orchestration is handled by `pipeline.py`, which instantiates the `LatentRecurrentDepthConfig` and model, loads and splits the dataset, constructs `DataLoader` objects with appropriate collators, initializes the `Trainer` with optimizer and scheduler, runs the training loop for a specified number of epochs, saves checkpoints, and writes `matplotlib` loss curves for visualization. The command-line entry point `train.py` exposes learning rate, batch size, and epoch count as arguments, making every training run fully scriptable. Checkpoints are saved in `save_pretrained` format, enabling seamless continuation and Hub upload via `push_to_hub.py`.

TABLE III
Training Hyperparameters and Configuration (Default / Example Values)

| Component | Value / Description |
|---|---|
| Optimizer | AdamW ($\beta_1$=0.9, $\beta_2$=0.999, $\epsilon$=$10^{-8}$) |
| LR scheduler | CosineAnnealingLR with warm-up |
| Initial LR | $3 \times 10^{-4}$ |
| Batch size | Configurable (typically 8–32) |
| Sequence length | $\leq 1024$ |
| Loss function | Causal LM cross-entropy |
| Gradient clipping | $\ell_2$ norm $\leq 1.0$ |
| Mixed precision | `torch.amp` / Accelerate |

**Algorithm 4** Training Step in `Trainer` Class

**Input:** mini-batch of tokenized sequences $x$
**Output:** updated model parameters
Sample $N \sim \text{Uniform}\{1, 2, 3, 4\}$ `// randomized depth exposure`
$\mathbf{L} \leftarrow$ `Model.forward`$(x, N)$ $\mathcal{L} \leftarrow$ `CrossEntropyLoss`$(\mathbf{L}[:, :-1, :], x[:, 1:])$
$\mathcal{L}$`.backward()` Clip gradients ($\ell_2$ norm $\leq$ 1.0) `Optimizer.step()`; `Scheduler.step()` Log loss and perplexity

### B. Inference Pipeline

Inference is designed for maximum flexibility with minimal setup. Because the model inherits the full `PreTrainedModel.generate` interface, users control reasoning depth with a single argument:

```
output = model.generate(
    input_ids,
    max_new_tokens=128,
    num_iterations=12,   # controllable latent depth
    temperature=0.7,
    top_k=50,
    do_sample=True
)
```

Increasing `num_iterations` enables deeper latent reasoning with no additional parameters or memory overhead beyond the per-iteration compute cost. The `Inference/` directory provides three ready-to-run scripts. `locally.py` loads a checkpoint from a local path or the Hub and performs generation with a user-specified depth. `Squence_Generator.py` supports full-sequence generation with configurable prompt, length, temperature, and depth, and prints post-processed results. `One_word.py` performs single-token next-word prediction, useful for debugging attention patterns and recurrent state dynamics. All scripts detect CUDA availability automatically, set the model to evaluation mode, and reuse the exact forward pass from training, ensuring consistent behavior between training and deployment.

### C. Reproducibility and Scalability

Reproducibility is treated as a first-class concern throughout the pipelines. Configuration objects are serialized alongside

every checkpoint so that any experiment is self-describing. Random seeding for `num_iterations` sampling is fully deterministic under `torch.manual_seed`. Loss curves and metrics are saved automatically at the end of each epoch. The pipelines scale naturally from a consumer GPU running the default 93-million parameter model to larger experiments; future extensions such as multi-node distributed training via Accelerate or production serving via vLLM are straightforward to add given the modular structure.

## VII. Use Cases and Examples

LatentRecurrentDepthLM is designed to be immediately usable by academic researchers exploring recurrent-depth architectures, graduate students learning advanced sequence modeling, independent developers prototyping efficient reasoning systems, and practitioners deploying adaptive language models in resource-constrained environments. This section presents four concrete, reproducible use cases drawn directly from repository patterns.

### A. Quick Inference with Controllable Latent Depth

The most common entry point is loading the pre-trained checkpoint from the Hugging Face Hub and generating text with user-specified reasoning depth.

```
from transformers import AutoTokenizer,
    AutoModelForCausalLM

model_name = "codewithdark/latent-recurrent-depth-lm
    "
tokenizer  = AutoTokenizer.from_pretrained(
    model_name)
model      = AutoModelForCausalLM.from_pretrained(
                model_name, trust_remote_code=True,
     device_map="auto")

prompt = "The future of artificial intelligence lies
     in"
inputs = tokenizer(prompt, return_tensors="pt").to(
    model.device)

for depth in [4, 8, 16]:
    out = model.generate(**inputs, max_new_tokens
    =80,
                         num_iterations=depth,
    temperature=0.75,
                         top_k=50, do_sample=True,
                         pad_token_id=tokenizer.
    eos_token_id)
    print(f"Depth {depth}: {tokenizer.decode(out[0],
     skip_special_tokens=True)}")
```

Listing 1. One-line loading and controllable-depth generation

Increasing `num_iterations` trades additional forward-pass compute for deeper implicit reasoning at constant model size, consistent with findings in the latent reasoning literature [8], [9].

### B. Training on a Custom Text Dataset

Researchers frequently need to continue pre-training or fine-tune on domain-specific corpora such as scientific papers, legal documents, or code. The framework supports this through a configuration-driven pipeline that requires no modification to source code.

```
from pipeline import run_training_pipeline

config = {
    "model_name"           : "codewithdark/latent-
    recurrent-depth-lm",
    "dataset_name"         : "your-org/your-domain-
    text",
    "max_length"           : 1024,
    "batch_size"           : 16,
    "epochs"               : 3,
    "lr"                   : 3e-4,
    "num_iterations_range" : [1, 2, 3, 4],
    "output_dir"           : "./checkpoints/domain-
    adapted-model"
}
run_training_pipeline(config)
```

Listing 2. End-to-end training on a custom dataset

The pipeline loads and tokenizes the dataset, initializes the model with a custom `LatentRecurrentDepthConfig`, trains with randomized iteration counts, saves periodic checkpoints, and plots loss curves automatically.

### C. Rapid Prototyping and Ablation Studies

The one-class-per-file module structure enables fast ablation without touching the training or inference pipelines. Replacing the attention kernel with FlashAttention-2 **dao2022flashattention** requires editing only `multi_head_Attention.py`. Changing the recurrent state injection from additive to multiplicative gating requires a single line in `recurrent_Block.py`. No other file needs to change in either case, because all blocks communicate through well-defined tensor interfaces.

### D. Model Sharing and Community Deployment

After training or fine-tuning, a checkpoint can be shared with the community via one command:

```
python push_to_hub.py \
    --model_path ./checkpoints/final-model \
    --repo_id    your-username/my-recurrent-depth-
    model
```

Listing 3. Uploading a trained checkpoint to the Hugging Face Hub

The uploaded model retains full custom-code compatibility and can be loaded by any user with `AutoModelForCausalLM.from_pretrained(..., trust_remote_code=True)`.

### E. Summary of Target Audiences

Table IV summarizes the primary user groups and the value the framework delivers to each.

## VIII. Comparison with Similar Tools

LatentRecurrentDepthLM occupies a distinctive niche in the open-source language modeling landscape: it is one of the few lightweight, fully Hugging Face-compatible implementations that natively supports recurrent-depth architectures with explicit, user-controlled test-time compute. Table V summarizes the comparison across key dimensions, and the subsections below discuss each tool in more detail.

TABLE IV
TARGET USERS AND PRIMARY VALUE PROPOSITION

| User Group | Key Benefits |
|---|---|
| Academic researchers | Modular blocks for rapid ablation; depth scaling experiments |
| Graduate students | Clean, well-documented code for learning recurrent-depth concepts |
| Independent developers | Lightweight 93M model; one-line Hub deployment |
| Resource-constrained teams | Constant-parameter depth scaling; runs on consumer GPUs |
| Open-source contributors | MIT license; clear contribution path via GitHub |

TABLE V
COMPARISON OF LATENTRECURRENTDEPTHLM WITH RELATED OPEN-SOURCE TOOLS

| Tool | HF Native | Recurrent-Depth | Controllable Iters | Full Pipeline | Lightweight |
|---|---|---|---|---|---|
| LatentRecurrentDepthLM | Yes | Explicit latent | Yes (per-step) | Yes | Yes (93M) |
| Universal Transformer [7] | No | Yes | Partial (halting) | Research-only | Yes |
| Geiping et al. [8] | Partial | Yes | Yes | Large-scale only | No |
| Ouro / LoopLM [9] | No | Yes | Learned depth | Pre-training scripts | No |
| LoopFormer [11] | No | Yes | Elastic depth | Training scripts | No |
| RWKV [22] | Yes | Implicit (SSM) | No | Yes | Yes |
| Mamba [23] | Yes | Implicit (SSM) | No | Yes | Yes |
| Transformers GPT-style [14] | Yes | No | Fixed depth | Yes | Varies |

The Universal Transformer [7] is the conceptual predecessor of the present framework. Its original TensorFlow implementation is research-oriented and focused on dynamic halting rather than user-specified iteration counts, and it predates the Hugging Face ecosystem. LatentRecurrentDepthLM simplifies the halting mechanism to a fixed integer count while gaining full modern ecosystem compatibility and a complete training pipeline.

The Geiping et al. codebase [8] is the closest conceptual relative and a direct inspiration for this work. It demonstrates impressive empirical gains at 3.5B parameters, but its infrastructure targets multi-GPU large-scale pre-training and is not designed for lightweight experimentation or custom-dataset training. The two codebases are therefore complementary: Geiping et al. for frontier-scale research, and LatentRecurrentDepthLM for accessible prototyping and deployment.

Ouro [9] and LoopFormer [11] pursue related but distinct goals. Ouro introduces entropy-regularized depth allocation learned during pre-training at 7.7T-token scale, while LoopFormer investigates elastic-depth training with shortcut modulation. Both release code oriented toward reproducing specific large-scale experiments rather than supporting modular experimentation; neither offers Hugging Face-native loading or a lightweight inference path.

RWKV [22] and Mamba [23] represent a different point in the design space: they use implicit state-space recurrence to achieve linear-time inference and constant memory, making them attractive for long contexts. They do not, however, expose an explicit multi-step latent refinement loop, so they are not well suited for studying or leveraging the kind of controllable test-time compute depth that is central to LatentRecurrentDepthLM.

Standard GPT-style models in the Hugging Face Transformers library [14] provide excellent general-purpose support but do not include recurrent-depth mechanisms. Users wishing to explore latent iteration must implement custom loops themselves—precisely the gap this framework addresses.

In summary, LatentRecurrentDepthLM stands out for its combination of full Hugging Face ecosystem readiness, explicit and user-configurable iteration depth, complete single-GPU training and inference pipelines, and a 93-million-parameter footprint that runs on consumer hardware. It does not aim to replicate the benchmark performance of frontier-scale looped models, but provides an ideal research platform and starting point for extensions toward that goal.

## IX. DISCUSSION AND CONCLUSION

### A. Discussion

LatentRecurrentDepthLM provides an accessible, well-engineered entry point into the recurrent-depth language modeling paradigm. Its core contribution is not a new architectural formula but a production-ready open-source realization of ideas established in the literature [8], [9], [11] in a form that is immediately usable by researchers and practitioners who lack large-scale compute infrastructure.

The weight-shared recurrent block achieves true parameter efficiency: model size is fixed at initialization, while effective reasoning depth is a runtime variable. This directly realizes the promise of test-time compute scaling without emitting intermediate tokens, offering a clean alternative to chain-of-thought methods [6] for tasks where latent refinement is sufficient. The three-stage Prelude-Recurrent-Coda decomposition, the one-class-per-file module organization, and the randomized depth curriculum during training collectively make the framework both easy to understand and easy to extend.

Several limitations must be acknowledged honestly. The repository currently demonstrates training only on a small WikiText-2 subset with no perplexity or downstream benchmark scores reported, leaving questions about real-world scaling

behavior and optimal iteration counts open. There are no automated unit tests or continuous integration workflows, which are standard expectations for production open-source software. The fixed 1 024-token context limit restricts applicability to document-level tasks. Distributed training is not yet implemented, limiting scalability on large corpora. Finally, the iteration count is either fixed or randomly sampled; a learned adaptive halting mechanism [7] would enable more principled allocation of compute per input. None of these are fundamental architectural flaws; each maps onto a concrete, bounded development task suitable for community contribution.

### B. Conclusion

LatentRecurrentDepthLM delivers a robust, extensible, and immediately usable open-source framework for recurrent latent depth language modeling. Its three-stage architecture, explicit test-time depth control, complete training and inference pipelines, and seamless Hugging Face integration lower the barrier to exploring one of the most promising directions in efficient sequence modeling. The framework is particularly well suited for researchers conducting ablation studies on latent iteration strategies, educators teaching alternatives to stacked transformers, developers prototyping parameter-efficient reasoning systems, and contributors seeking a clean foundation for extensions such as adaptive halting, FlashAttention integration, longer contexts, or multimodal variants.

We recommend the following directions for future development: large-scale pre-training with systematic evaluation on standard reasoning and language-modeling benchmarks; adaptive computation mechanisms such as learned halting or confidence-based early exiting; support for longer contexts, distributed training, and optimized inference backends such as vLLM; addition of unit tests, a CI pipeline, and a pinned `requirements.txt`; and community-driven multimodal or vision-language extensions.

The repository (`codewithdark-git/LatentRecurrentDepthLM`) and the associated Hugging Face model checkpoint at `codewithdark/latent-recurrent-depth-lm` are released under the MIT license. We invite the community to adopt, extend, and improve this foundation as recurrent-depth and latent-reasoning techniques continue to mature.

## References


[1] A. Vaswani et al., "Attention is all you need," in *Advances in Neural Information Processing Systems*, 2017.

[2] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[3] J. Kaplan et al., "Scaling laws for neural language models," *arXiv preprint arXiv:2001.08361*, 2020.

[4] J. Hoffmann et al., "Training compute-optimal large language models," *arXiv preprint arXiv:2203.15556*, 2022.

[5] T. Brown et al., "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, 2020.

[6] J. Wei et al., "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems*, 2022.

[7] M. Dehghani, S. Gouws, O. Vinyals, J. Uszkoreit, and L. Kaiser, "Universal transformers," *arXiv preprint arXiv:1807.03819*, 2018.

[8] J. Geiping et al., "Scaling up test-time compute with latent reasoning: A recurrent depth approach," *arXiv preprint arXiv:2502.05171*, 2025.

[9] Y. Zhu et al., "Scaling latent reasoning via looped language models," *arXiv preprint arXiv:2510.25741*, 2025.

[10] A. Pappone et al., "Two-scale training for recurrent-depth language models," *arXiv preprint*, 2025.

[11] A. Jedd et al., "Loopformer: Elastic-depth looped transformers for latent reasoning," *arXiv preprint arXiv:2602.11451*, 2026.

[12] "Depth-recurrent attention mixtures (dreamer): Sparse expert routing for looped transformers," *arXiv preprint arXiv:2601.04512*, 2026.

[13] "Recurrent-depth vision-language-action models for robotics," *arXiv preprint arXiv:2602.07845*, 2026.

[14] T. Wolf et al., "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2020, pp. 38–45.

[15] A. Paszke et al., "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, 2019.

[16] K. Choromanski et al., "Rethinking attention with performers," in *International Conference on Learning Representations*, 2021.

[17] W. Merrill et al., "On the expressivity of transformers with shared parameters," *arXiv preprint*, 2023.

[18] N. Saunshi et al., "Looped transformers for efficient depth scaling," *arXiv preprint*, 2024.

[19] "Efficient parallel samplers for recurrent-depth models," *arXiv preprint arXiv:2510.14961*, 2025.

[20] "Teaching pretrained language models to think deeper with recurrent depth," *arXiv preprint arXiv:2511.07384*, 2025.

[21] "Emergent latent chain-of-thought in depth-recurrent transformers," *arXiv preprint arXiv:2512.18934*, 2025.

[22] B. Peng et al., "Rwkv: Reinventing rnns for the transformer era," *arXiv preprint arXiv:2305.13048*, 2023.

[23] A. Gu and T. Dao, "Mamba: Linear-time sequence modeling with selective state spaces," *arXiv preprint arXiv:2312.00752*, 2023.

[24] S. Arora and A. Risteski, "On the opportunities and risks of foundation models," *arXiv preprint arXiv:2108.07258*, 2023.