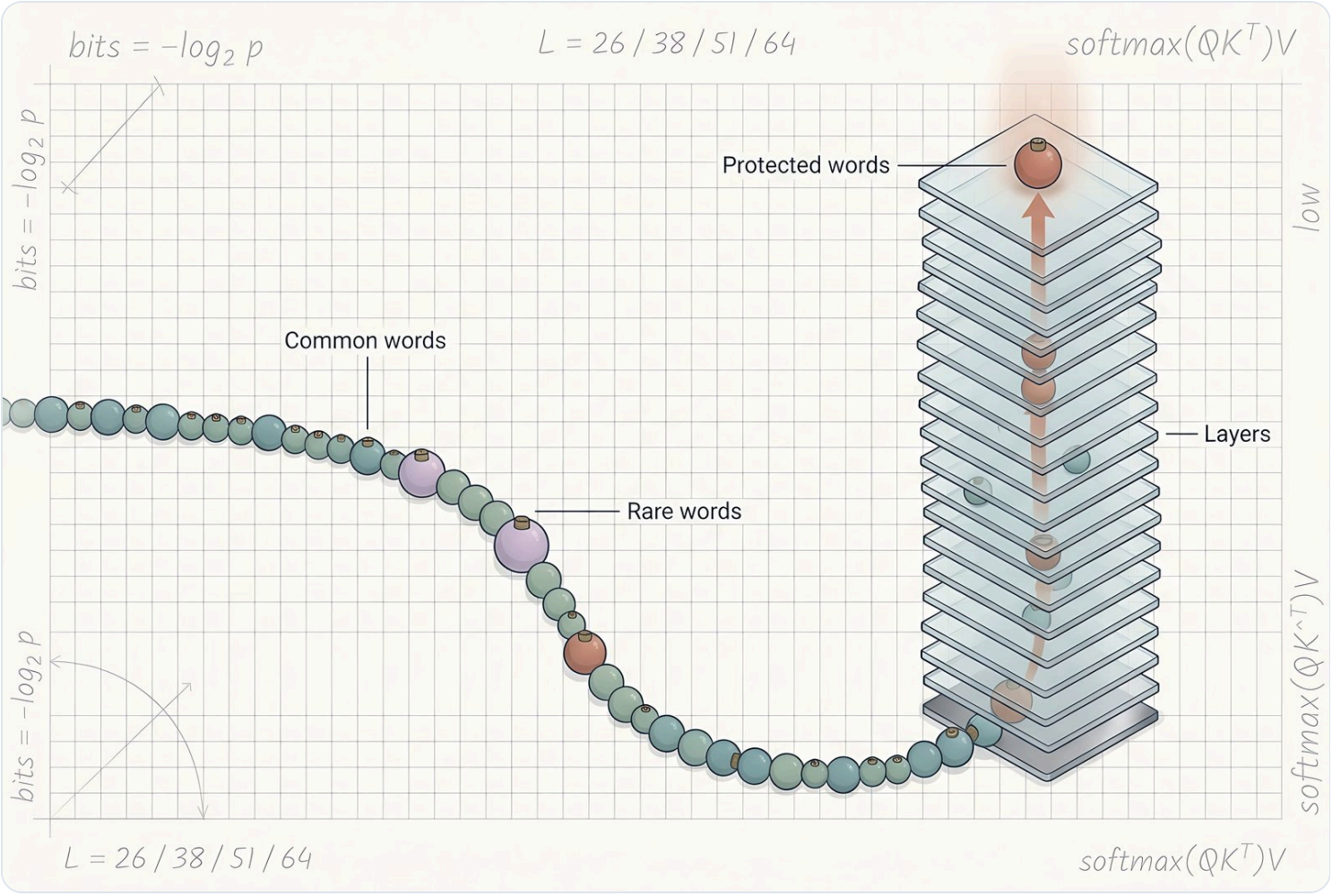

WORKING PAPER · LANGUAGE MODELS AND EFFICIENT INFERENCE

Dynamic Beads

One short tag on each token sets its code length, its embedding size and its layer depth, with a depth guard for the words that must not be skimmed

Eytan Yair Goldstein and Benny Goldstein · Benny Labs Foundation, New York · b@bennylabs.org
Working paper, November 29, 2026

ABSTRACT

A language model treats every token the same way: each gets an identifier of the same width, an embedding of the same size and a trip through every layer. Dynamic Beads gives each vocabulary unit, called a bead, a tiered binary code. Common beads get short codes and rare ones longer codes, and a short tag at the front names the tier. The central idea is that the one tag can set three things at once: how many bits the bead takes to send and store, how large its embedding is, and how many

layers it passes through. A depth guard sends negations, numbers, logical words and low-confidence positions deeper, whatever their tier. Around that core the design has tiers that promote and relegate beads as usage changes, a personal tier whose codes mean different things for each user over one shared model, a ladder of tiers up to concepts with stored answers, knowledge taught through a temporary shadow copy of a few weights, and growth capacity held in reserve.

We report measured results. In small models built from scratch, the depth guard lifted a negation test from 53.7% to 99.9% at 55.8% of the layer work, and tiers placed by use needed 55.8% of the layer work against 83.0% for the same tiers filled at random. Put onto Qwen3.8-27B with a short tune-up, per-token depths read a prompt 1.41 times as fast as the original model at 16,384 tokens and 1.47 times at 65,536, gave the same next token 93.5% of the time at 63.9% of the prompt's layer work, found 18 of 18 hidden needles, answered 60 of 60 negation questions and scored 95.0% on GSM8K against the original's 93.9%. The tiered codes averaged 12.1 bits per token against 18, and narrower embeddings for rare tokens removed 73.2% of the input embedding table.

Not everything held. On Qwen3.5-9B a shuffled tag tied the real tag, and the tier depths added nothing measurable beyond the depth guard. Section 9 says so plainly, and the paper ends with the questions we would most like others to test.

Keywords: language models, tokenization, variable-length codes, early exit, layer skipping, adaptive computation, adaptive embeddings, prompt processing

1. Introduction
2. Beads, tiers and the tag
3. One tag, three jobs
4. The depth guard
5. Tiers that move
6. A personal tier
7. Concepts, spoon-fed knowledge and growth in reserve
8. What we measured
9. What did not hold up
10. Related work
11. Open questions
References

1. Introduction

Why does a language model spend the same work on the word "the" as on a rare technical term? A tokenizer cuts text into tokens, and from then on every token is handled alike. Each has an identifier of the same width, an embedding of the same size, and a pass through every layer of the model.

Language is not spread evenly. A small set of tokens makes up most of any text, and most of the vocabulary is rare. Dynamic Beads is a design that lets the model's treatment of a token follow that fact, using one small piece of information carried by the token itself.

The pieces are known. Variable-length codes, early exit, embeddings of different sizes and answer caches all exist (Section 10). What this design adds is tying them to one tag on the token, known before the first layer runs, and letting tokens move between tiers while the model is in use. This paper describes the ideas, then reports what the tests showed, including a test that went against us.

---

2. Beads, tiers and the tag

We call a vocabulary unit a *bead*. A bead may stand for a word, a piece of a word, a phrase, a sentence or a concept. The vocabulary is split into tiers, and each tier is named by how many bits its codes use. Common beads sit in short-code tiers. Rare beads sit in long-code tiers.

A bead's code has two parts: a short *tag* that names its tier, and a *payload* of that tier's length. Because the tag comes first, a stream of beads can be read with no separators: the reader reads the tag, knows how many payload bits follow, and reads them.

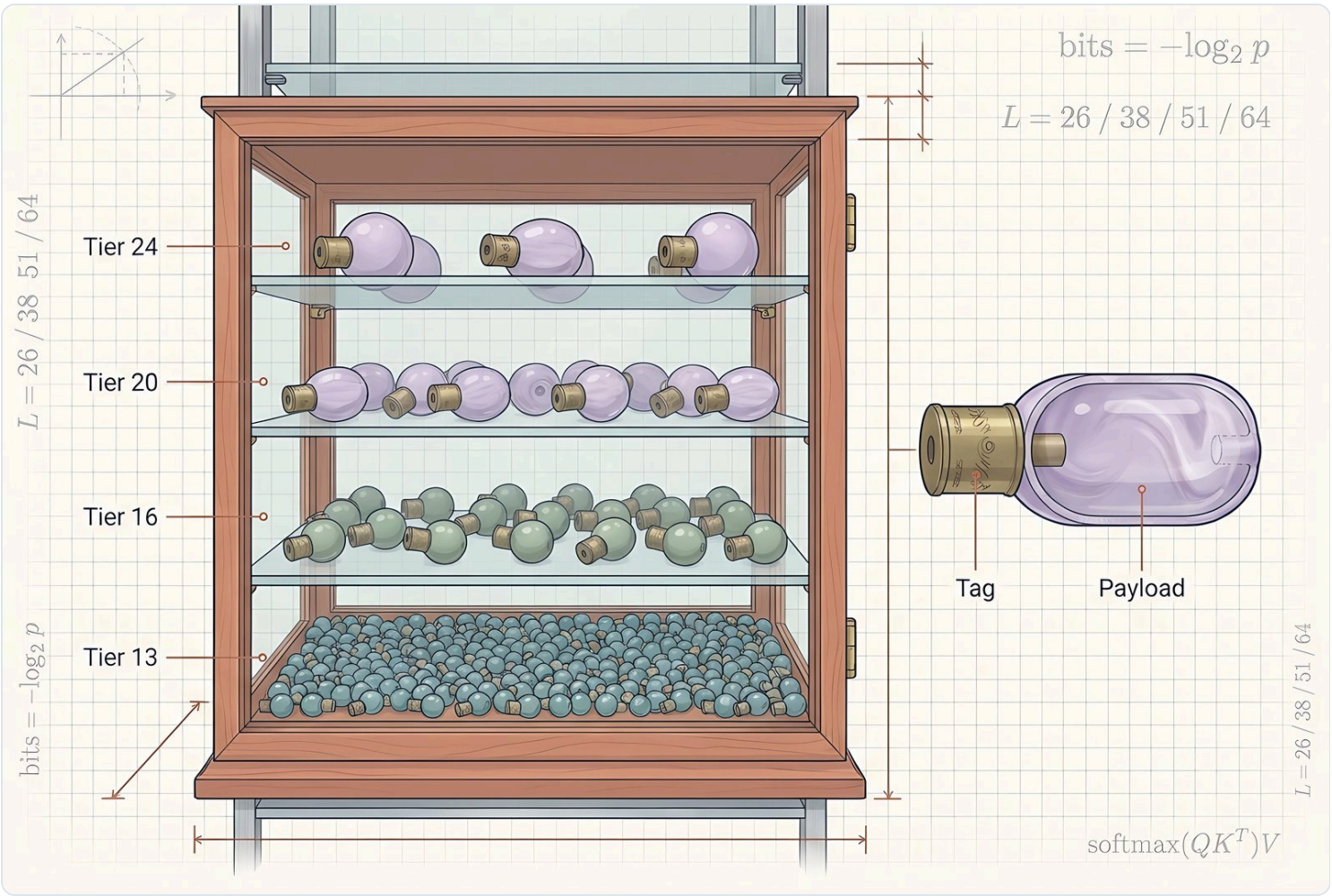

Figure 1. Tiers as shelves. Common beads are small and many; rare beads are large and few. Every bead is a tag plus a payload.

3. One tag, three jobs

The model reads the tier from the tag. That one read sets three things:

- **Code length.** How many bits the bead takes to send and to store.
- **Embedding size.** Each tier has its own embedding table. Rare beads get a narrower embedding, which a small projection maps up to the model's working width.
- **Layer depth.** Each tier has a default number of transformer layers. Past that depth the bead skips the feed-forward computation, while light attention continues so that other tokens can still see it.

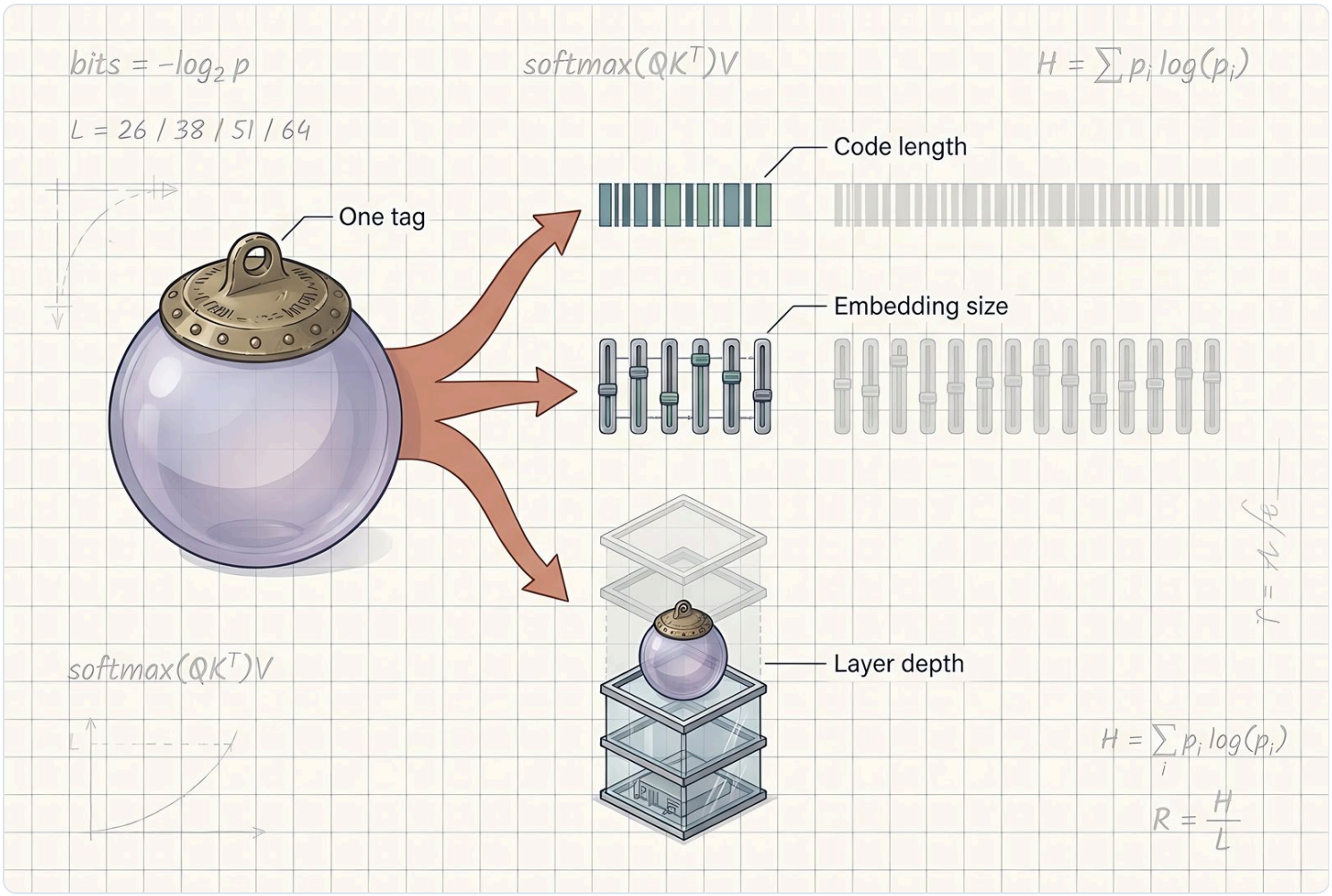

Figure 2. One tag sets three things: the bead's code length, its embedding size and how many layers it passes through.

4. The depth guard

Frequency is a poor guide for some words. "Not" is very common and very important. A model that gave it shallow treatment because it is common would get sentences backwards.

So a *depth guard* raises the depth for beads marked as negations, numbers, logical operators or pronouns, and for any position where a confidence measure taken partway through the model is low. The picture at the top of this page shows it: common beads leave the stack early, and a protected bead goes all the way up.

---

5. Tiers that move

Tier membership is not fixed. Usage of each bead is counted, and beads are promoted to shorter-code tiers or relegated to longer ones, in the manner of a sports league. A newcomer displaces the least-used member of a tier only after beating it by a set margin for a set time, so beads do not bounce back and forth. Each tier is updated at its own pace: slowly for proven common tiers, quickly for a trend tier.

When a bead moves, its embedding row is moved to the new tier's table and resized. The new dictionary, the moved rows, the projections and the output weights are checked and switched on together as one *version bundle*, and every request runs from start to finish under one bundle. Dictionaries on the two ends of a connection stay in step by applying the same rule to the same ordered messages.

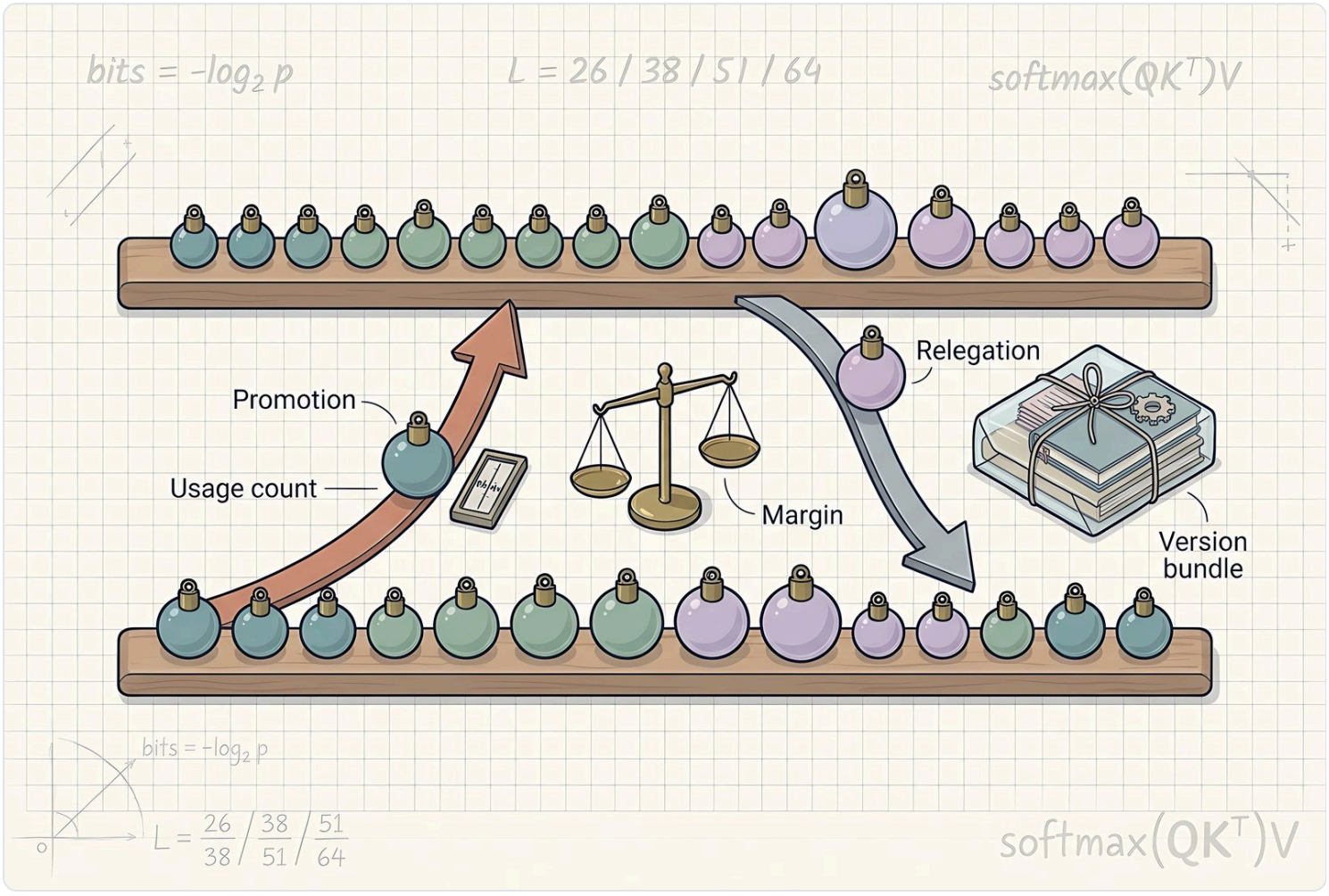

Figure 3. Promotion and relegation. A bead moves up only after it beats the weakest member by a margin, and every move ships inside one version bundle.

6. A personal tier

One range of codes is reserved as a personal tier. The same code means something different for each user. A person's table maps a personal code to a phrase that person repeats often, so the whole phrase takes a single input position. The single bead is put to use only after a test shows that the model treats it like the full phrase. The shared model weights are unchanged and serve every user.

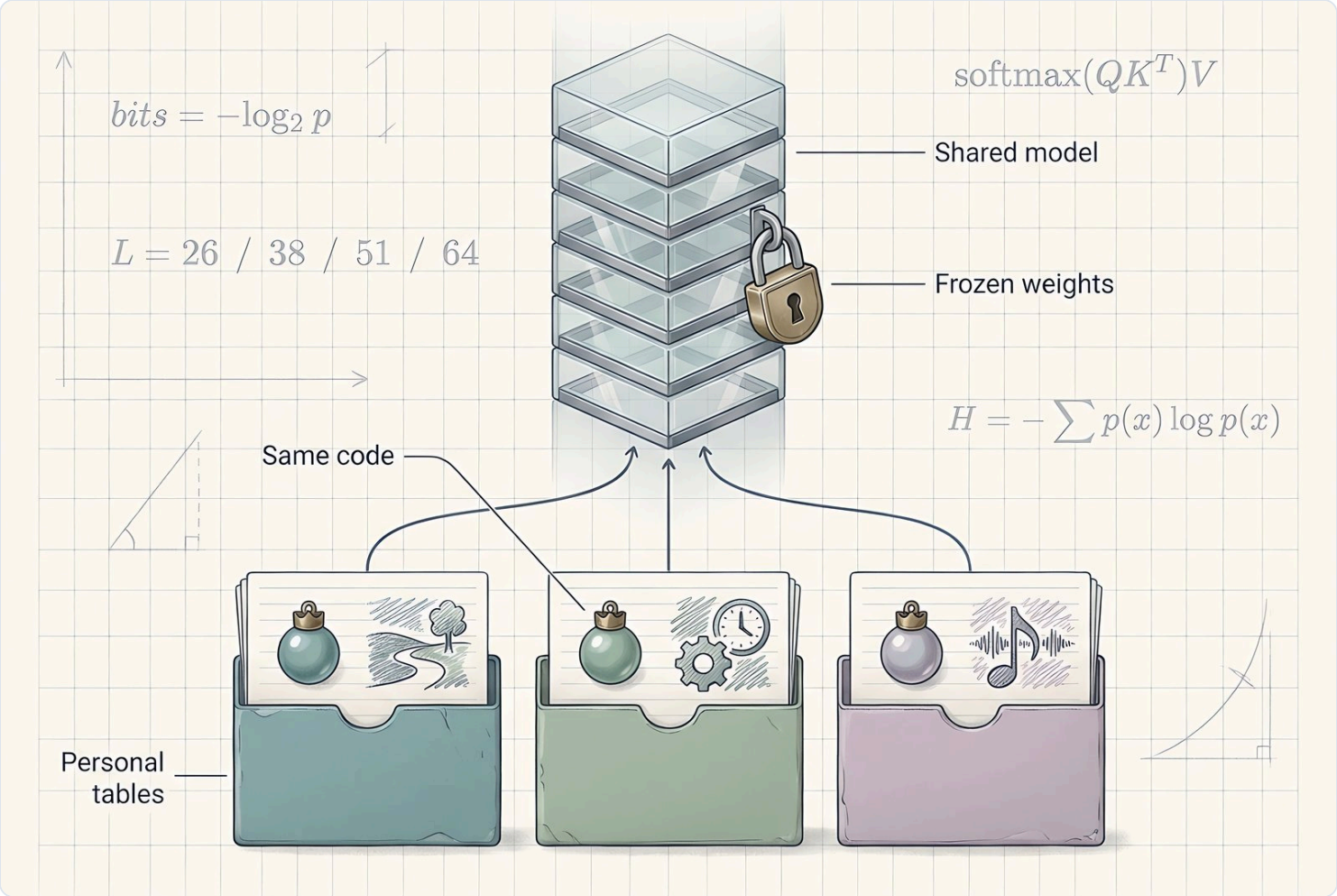

Figure 4. A personal tier. The same code points to a different phrase for each person, over one shared model whose weights do not change.

7. Concepts, spoon-fed knowledge and growth in reserve

A ladder up to concepts. The tiers run from words through phrases and sentences to concepts and trends, all in one tagged code space. Differently worded questions with the same meaning map to one concept bead. An answer stored under that bead can be returned without running the model, but only after checks on names, numbers, negation, dates, permissions and versions.

Spoon-fed knowledge. A fact supplied together with its answer is taught to a temporary shadow copy of a few of the model's weights. The shadow copy is nudged quickly while the original weights stay as they were. It is used through a gate for matching inputs, folded into the main weights over time, and retired once the main weights give the answer by themselves.

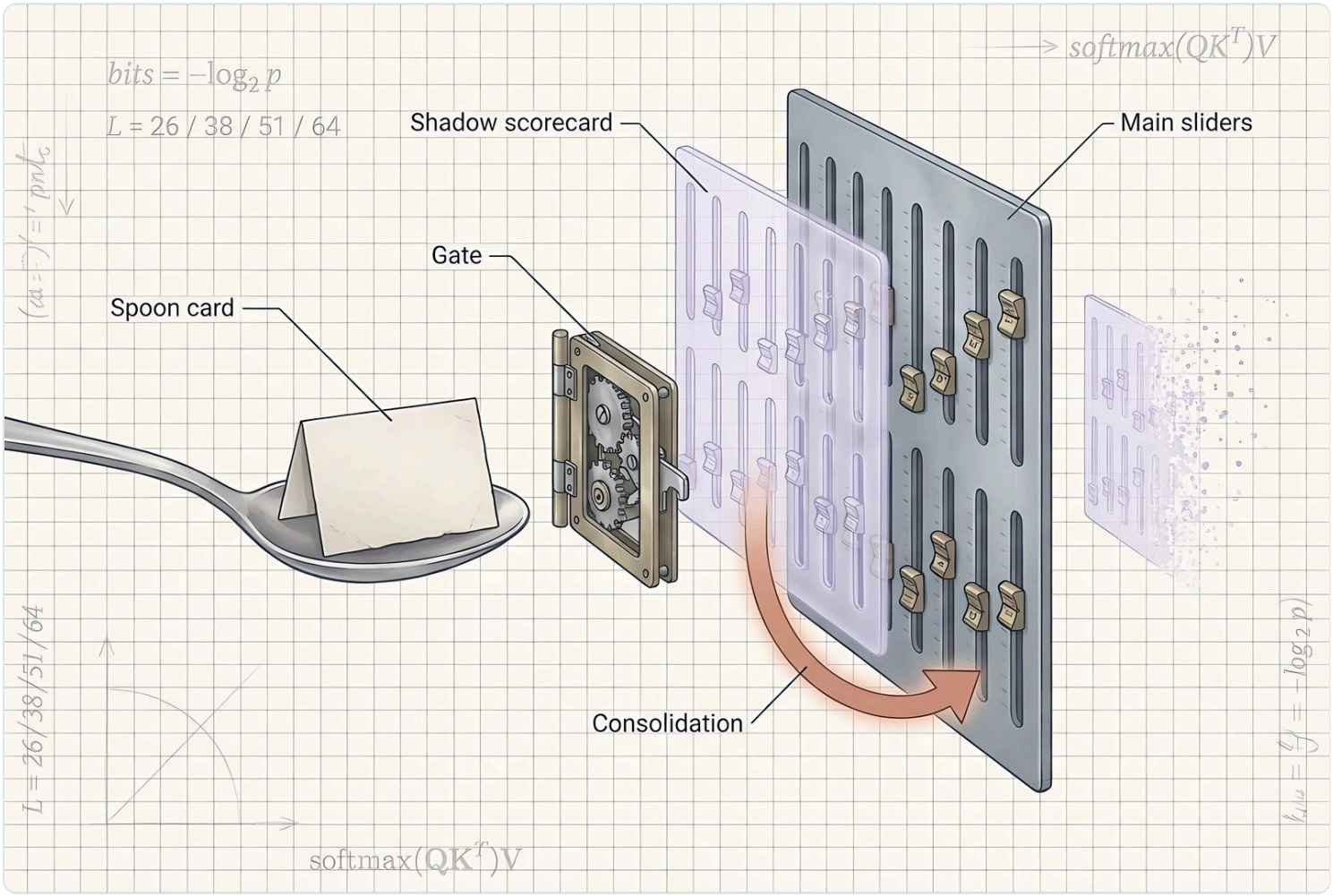

Figure 5. Spoon-fed knowledge. A new fact is taught to a shadow copy of a few weights, used through a gate, then consolidated into the main weights.

Growth in reserve. Spare blocks of parameters are built in advance and left dormant. They are set to contribute zero and flagged to be skipped, so they cost storage but no arithmetic. When a tier's performance calls for it, a block is switched on, trained, and kept only if it passes a test on held-out data.

8. What we measured

8.1 Small models built from scratch

The first tests used small models and a toy language built for the purpose, so that each idea could be tested by itself and could fail.

Table 1. Results in small models built from scratch, and on one small open model.

Test	Result
Depth guard on a negation test	53.7% right without the guard, 99.9% with it, at 55.8% of the layer work
Tiers placed by use, against the same tiers filled at random	55.8% of the layer work, against 83.0%
Code length in a toy language of 128 beads	4.66 bits per bead, against 7.00 for a fixed code. The best possible for that distribution is 4.52.
The margin rule for moving between tiers	Tier moves over 12 periods fell from 186 to 12
Dictionaries on two ends, one message lost	899 messages decoded right and 0 wrong with check values; 595 right and 304 wrong without
Tier depths put onto SmolLM2-135M	Read 1.31 times as fast at 50.5% of the layer work, with 100% on the negation test

Two lessons came out of these. The guard is necessary: without it the tagged model did no better on negation than ignoring the word. And placement must follow use: the same tiers filled at random kept only a small part of the saving.

8.2 Qwen3.8-27B

We then put the tier depths onto an existing open model, Qwen3.8-27B, with a short tune-up, and compared it with the original model on the same inputs.

Table 2. Results on Qwen3.8-27B against the original model.

Test	Result
Reading a long prompt	1.41 times as fast at 16,384 tokens, 1.47 times at 65,536
Same next token as the original model	93.5% of the time, at 63.9% of the prompt's layer work
Hidden needles found in long documents	18 of 18
Negation questions	60 of 60
GSM8K	95.0%, against 93.9% for the original
Code length on 729,930 held-out tokens	12.1 bits per token, against 18 for fixed-length identifiers (33% shorter)
Narrower embeddings for rare tokens	931 million numbers removed, 73.2% of the input embedding table, at a divergence (KL) of 0.009 at full depth before any training

Shorter codes alone do not make a model faster. The speed comes from fewer layers per bead. These tests measure reading the prompt, which is where a long document spends its time; they do not measure writing the answer.

9. What did not hold up

The tests were designed so that they could fail, and one important one did.

On Qwen3.5-9B we tested the tag against a shuffled tag, one that hands tokens to tiers with no regard to how common they are. The shuffled tag scored the same as the real one: 85.3% against 85.7%. Random depths that kept the guard scored about the same too. A guard placed at the same number of random positions was not clearly worse than the real guard (83.2% against 85.5%). In that test the tier depths added nothing we could measure beyond the guard.

So we hold two things apart. That a model can give many tokens fewer layers and keep its answers, with a guard on the words that matter, is well supported. That *frequency* is the right way to choose which tokens get fewer layers is supported in small models and is still open on mid-size ones.

The models also stayed weak at adding up numbers taken from a long document.

10. Related work

Adaptive depth. Confident Adaptive Language Modeling exits when a confidence measure inside the network is high enough [1]. Mixture-of-Depths trains a router that lets only some tokens through each block [2]. LayerSkip trains a model to exit early and checks its drafts with the remaining layers [3]. Closest to the tag idea, HashEE fixes each token's exit layer by hashing its identifier, with no learned decision [4].

Embeddings of different sizes. Adaptive input representations give frequency bands input embeddings of different sizes projected to one width [5].

Codes and vocabularies. Huffman coding gives frequent symbols short codewords [6]. SuperBPE merges across spaces into multi-word tokens [7].

Stored answers and edited facts. GPTCache stores whole answers for repeated questions [8]. ROME locates a fact in a model's weights and edits it directly [9].

In these approaches depth is decided inside the network or by a trained router, and code length, embedding size and depth are separate matters. In Dynamic Beads one tag on the token sets all three, it is known before the first layer runs, and it changes with use.

11. Open questions

We would be glad to see others test these, and results that go against ours are just as welcome.

1. Why would a shuffled tag tie the real one on a mid-size model? What test would settle whether frequency should set depth?
2. Does the reading speed-up hold on other open model families and at other sizes?
3. What does the design save when the model is writing an answer, not reading a prompt?
4. The moving tiers, the personal tier, the concept ladder and spoon-fed knowledge have been tested only in small models so far. How do they behave at scale?

References

1. T. Schuster, A. Fisch, J. Gupta, M. Dehghani, D. Bahri, V. Q. Tran, Y. Tay and D. Metzler, "Confident adaptive language modeling," Advances in Neural Information Processing Systems 35, 2022. arXiv:2207.07061
2. D. Raposo, S. Ritter, B. Richards, T. Lillicrap, P. C. Humphreys and A. Santoro, "Mixture-of-Depths: dynamically allocating compute in transformer-based language models," arXiv:2404.02258, 2024. arxiv.org/abs/2404.02258
3. M. Elhoushi and colleagues, "LayerSkip: enabling early exit inference and self-speculative decoding," Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics, 2024.

arXiv:2404.16710

4. T. Sun, X. Liu, W. Zhu, Z. Geng, L. Wu, Y. He, Y. Ni, G. Xie, X. Huang and X. Qiu, "A simple hash-based early exiting approach for language understanding and generation," Findings of the Association for Computational Linguistics: ACL 2022. arXiv:2203.01670
5. A. Baevski and M. Auli, "Adaptive input representations for neural language modeling," International Conference on Learning Representations, 2019. arXiv:1809.10853
6. D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952. doi:10.1109/JRPROC.1952.273898
7. A. Liu, J. Hayase, V. Hofmann, S. Oh, N. A. Smith and Y. Choi, "SuperBPE: space travel for language models," arXiv:2503.13423, 2025. arxiv.org/abs/2503.13423
8. Zilliz, "GPTCache: semantic cache for LLMs," open-source software repository, 2023. github.com/zilliztech/GPTCache
9. K. Meng, D. Bau, A. Andonian and Y. Belinkov, "Locating and editing factual associations in GPT," Advances in Neural Information Processing Systems 35, 2022. arXiv:2202.05262

How to cite

E. Y. Goldstein and B. Goldstein, "Dynamic Beads: One Tag per Token Sets Code Length, Embedding Size and Layer Depth," Benny Labs Foundation working paper, November 2026. https://bennylabs.org/dynamic-beads/paper/

> **Disclosure.** The authors have a financial interest in this work. The tests used open models under their publishers' licences, with no affiliation with or support from their publishers, and ran on a home computer and rented cloud hardware paid for by the authors.

© 2026 Eytan Yair Goldstein and Benny Goldstein. Published by Benny Labs Foundation Inc., a 501(c)(3) nonprofit public charity. · bennylabs.org