purebyte

PureByte: SLMs Are the Future

Tiny byte-level AI Specialists, trained on your own data
(by you, or by an AI agent), running on ordinary CPUs

Pablo Sirvent Jiménez

PureByte · purebyte.ai · pablo@purebyte.ai

Preprint, version 1 · September 2026

Code: github.com/purebyte-ai/purebyte (runtime) and github.com/purebyte-ai/purebyte-train (training), Apache-2.0. Model weights: PureByte Model License 1.0 (free use, commercial use included; no redistribution, derived models or hosted service whose main value is the model). This paper: CC BY 4.0.
The evaluations of `secrets-code` and `pii` can be rerun from those repositories and public data, except those on CredData's original values; the limits of reproduction are stated in Appendix E.

Abstract

We hold that most automated decisions over data (is there a credential in this file, which bytes of this ticket are personal data, what is this blob) are best made by small models trained for each job, and we present PureByte, an architecture, runtime and training stack for them. We use "small language model" in a broad sense: an AI Specialist is a small learned model of byte sequences that answers with decisions, scores, labels and byte ranges instead of generated text. It is a stack of ternary state-space blocks of the Mamba-2 family over a byte embedding and optional hashed *n*-gram tables, with heads decoded under constraints, stored as one file of 1 to 29 MiB that a dependency-free C++ runtime executes on ordinary CPUs, with bit-identical results across the kernels and thread counts of a platform. Specialists are trained from scratch in minutes to an hour on one consumer GPU by a factory that blocks leaks between training data and examinations and verifies that the runtime reproduces the trained model (its leak checks came after the three released examples, whose overlaps with their examinations we measure); an AI coding agent can drive it from a description of the job, which we call Vibe Training. We formulate the approach (the answer types, an argument for when a specialist needs no general knowledge, and a cost model that includes the cost of errors) and measure three released examples, which all answer with byte ranges gated by a decision (the other answer types have no trained model yet). On 164 of the 168 repositories of CredData's test half, a 1.9-million-parameter model finds 3.5 times as many labeled credential lines as gitleaks with its default rules, at a close precision on labeled lines (3.1 times on all 168), after learning partly from labeled lines of the benchmark's other half; with the benchmark's obfuscated values replaced by the original ones, 2.9 times. An 8.3-million-parameter model scores F2 0.769 on the PII Masking Benchmark, 13th of 36 and above OpenAI's Privacy Filter (1.4 billion parameters, about 50 million active, with a narrower policy), after training on the sources of four of its six tasks (on the other two tasks the two models are level); the best small pretrained encoders score higher at a few times its cost per byte. A 64-byte decision takes 0.8 to 1.5 ms on eight threads of a desktop CPU, 3 to 6 ms on one core. Each released model missed a pre-registered criterion as first written, and we report every one. We also report research on an attachable exact document memory, withdrawn from one application after a real-data test. The code, the model format and the training stack are open under Apache-2.0; the weights of the examples are free to use under a separate model license.

Contents

1 Introduction **4**

2 Background and related work **6**
2.1 Small language models and the case for them . . . 6
2.2 Specialists versus generalists . . . 7
2.3 Byte-level models . . . 8
2.4 State-space models . . . 8
2.5 Low-bit networks . . . 9
2.6 TinyML and edge AI . . . 10
2.7 The cost and energy of inference . . . 10
2.8 Agents that engineer and make models . . . 11
2.9 Detection of secrets and personal data . . . 12
2.10 Arm: the facts behind the analogy . . . 12
2.11 Positioning . . . 13

3 A theory of AI Specialists **13**
3.1 Jobs, specialists and generalists . . . 13
3.2 Any byte is trainable, and so is the output . . . 13
3.3 Why a specialist does not need general knowledge . . . 15
3.4 Making AI more efficient: the cost of an answer . . . 17

4 The architecture **18**
4.1 Overview . . . 19
4.2 Bytes and the byte embedding . . . 20
4.3 The hashed *n*-gram memory . . . 20
4.4 The `ssm_v2` block . . . 21
4.5 Ternary weights . . . 24
4.6 The optional attention block . . . 25
4.7 Heads, gating and decoding . . . 25
4.8 Windows, lookahead and the stream context . . . 27
4.9 The model file . . . 28
4.10 Parameters and bytes of the released specialists . . . 29
4.11 Design choices and their evidence . . . 30

5 The runtime **33**
5.1 Layers, registries and profiles . . . 33
5.2 The numerics contract . . . 33
5.3 Kernels . . . 35
5.4 Threads . . . 35
5.5 Memory . . . 36
5.6 Performance . . . 37
5.7 Interfaces . . . 40
5.8 Portability . . . 41
5.8.1 From servers to routers, robots and microcontrollers . . . 41

6 The training stack **43**
6.1 Two repositories, and what never enters git . . . 43
6.2 The task file and the plugin contract . . . 43
6.3 Data . . . 44
6.3.1 Exam fingerprints and the leak checks . . . 45
6.4 Recipes: the knobs that matter . . . 46
6.5 The training loop . . . 47
6.6 The factory: what each stage enforces . . . 49
6.7 Selection and calibration on held-out data . . . 50
6.8 Evaluation: control, rival and gate . . . 50
6.9 Export and verification . . . 51
6.10 Releases . . . 51
6.11 Hardware and cost . . . 52
6.12 Reproducibility: guarantees and gaps . . . 53

7 Vibe Training: an AI that trains small AIs **54**
7.1 Ways an agent can do a byte-level job . . . 54
7.2 The hypothesis and a cost model . . . 55
7.3 Why a trained model, and not a program the agent writes . . . 57
7.4 Why not reason at run time . . . 58
7.5 This project as a case, and what we did not measure . . . 59
7.6 Train-yourself, and small models as the tools of agents . . . 59

8 Case studies: three AI Specialists **60**
8.1 How we measure . . . 60
8.2 `secrets-code`: credentials in source code . . . 61
8.3 `pii`: personal data in text . . . 65
8.4 `secrets-bin`: credentials inside compiled binaries . . . 68

8.5 What the three examples teach . . . 70

9 Research direction: an attachable document memory 72
9.1 Design . . . 72
9.2 Instruments . . . 73
9.3 Results . . . 74
9.4 Attached to a released specialist, and a lesson from real data . . . 75

10 Discussion 76
10.1 SLMs are the future, in a precise sense . . . 76
10.2 The competing future . . . 77
10.3 The ARM of AI: an analogy and its limits . . . 77
10.4 Origin: a laboratory for efficient training . . . 78
10.5 Your data, your device . . . 78
10.6 Privacy, energy and cost . . . 78

11 Limitations and threats to validity 79

12 Conclusion 80

Acknowledgements and disclosure 81

A Notation 103

B Configurations of the released specialists 103

C Parameters of the cost model 104

D Data and licenses 105

E Reproducing the numbers 106

F Glossary 107

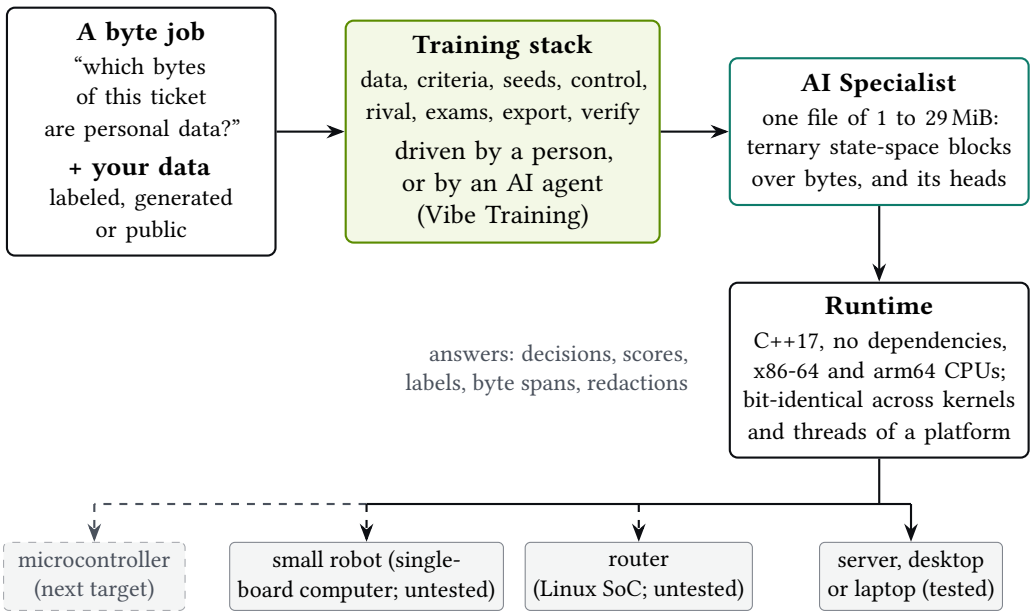


Figure 1: PureByte in one picture. A job and its data (left) become an AI Specialist through the training stack, driven by a person or by an AI agent (Vibe Training); the specialist is one small file that the runtime runs on ordinary x86-64 and arm64 CPUs, from servers, desktops and laptops (tested) to small boards (not yet tested), and, as a next target, microcontrollers (Section 5.8.1).

1 Introduction

Much of the AI that runs in the world does not write essays. It decides whether a file holds a leaked credential, which bytes of a support ticket are personal data, whether a log line is anomalous, what a blob of bytes is. These decisions are narrow, repetitive and numerous, and today they are answered in one of two ways: by rules that a person or an agent wrote, which are exact where the pattern has a fixed form and blind where it does not; or by a large general model, which understands the question but pays for its generality on every input. In a benchmark of 88 models on data-center GPUs, multi-purpose generative models used orders of magnitude more energy than task-specific ones on the same tasks, though with higher accuracy in text classification; across tasks, a thousand classifications cost about 0.002 kWh and a thousand short generations about 0.047 kWh [1]. And in one reported game-control scenario, a 1.3-million-parameter specialist trained on demonstrations outperformed language models up to 92,000 times larger that played without training [2].

This paper is about the small end. It presents PureByte, an open architecture, runtime and training stack for AI Specialists: byte-level models of two to fifty-four million parameters, each trained for one job, that read raw bytes (text, code, logs, binaries, network payloads) and answer with decisions, scores, labels and byte ranges instead of generated text. A specialist is a stack of ternary state-space blocks of the Mamba-2 family over a byte embedding and optional hashed n-gram tables, with heads that are decoded under constraints (Section 4). It is a single model file, from one to twenty-nine mebibytes, that a dependency-free C++ runtime executes on ordinary CPUs, offline, with bit-identical results on every kernel and at every thread count of a platform (Section 5). It is trained from scratch, with no pretrained body, on data that describe its job, in minutes to an hour on one consumer GPU (Section 6). And an AI coding agent can train it from a description of the job, as one did for the three examples under the author's direction (Section 7.5), through a factory that blocks leaks, silent options and unverified exports and records failed criteria, while some of its rules are written rather than enforced: Vibe Training (Section 7).

Theses. The paper argues four positions, each stated in a box where it is argued, with the evidence that bears on it; two of them (Vibe Training and the future of small models) are positions without a

measurement of their own:

1. *Specialist tasks do not need generalist knowledge* (Section 3.3): when the answer depends on local evidence in the bytes and the training data cover the inputs' forms, a model sized for that evidence and trained from scratch on the job's data can reach the quality of the tools in use. *Train with your data, not a generalist*: a generalist is not required, although small pretrained encoders remain strong competitors.
2. *Making AI more efficient* (Section 3.4): a model the size of the job does each decision with hundreds of times fewer operations and bytes of weights than a small generative model and several times fewer operations than a small pretrained encoder, on the processor already in the machine; the saving grows with every decision.
3. *Vibe Training* (Section 7): for byte-level jobs that meet the conditions of Hypothesis 11, an AI agent is better used training a small model once than writing rules for the job or reasoning about every input at run time.
4. *SLMs are the future, and PureByte aims to be the ARM of AI* (Section 10): most of the decisions over data now made by rules or by calls to general models will be made by small models made for their jobs, and the right principles for that world are the ones that made ARM ubiquitous: efficiency first, an architecture that others build on (ARM's by license; ours, like RISC-V's, open), and every device.

The title borrows from, and departs from, a recent argument that small language models suffice for many of the calls inside agentic systems [3]: those authors fine-tune pretrained generative models, whereas we train non-generative models of bytes from scratch, which we call small language models in the broad sense of small learned models of sequences. The title states a position (Section 10), not a result.

Contributions.

(i) A formulation of AI Specialists: byte jobs, the answer types that a small set of heads covers ("any byte is trainable, and so is the output"; today only byte ranges with a gating decision are trained), an argument for when a specialist needs no general knowledge, and the cost of an answer (Section 3).

(ii) The architecture in full detail, with the equations and the order of operations of every component, and the design choices with the evidence behind them (Section 4).

(iii) A deterministic CPU runtime: a numerics contract under which the scalar kernel, the vector kernel of each platform (AVX2 on x86-64, NEON on arm64) and every thread count give bit-identical results, measured latency, throughput and memory, and an analysis of what routers, robots and microcontrollers would require (Section 5).

(iv) A training stack whose factory blocks leaks between training data and examinations, trains a control and a classic rival, and verifies that the runtime reproduces the trained model; the released examples were trained on an AMD GPU, and NVIDIA and Apple silicon GPUs are supported through the same stock PyTorch code but not yet validated (Section 6).

(v) Vibe Training: a definition, a hypothesis with its conditions, a cost model that includes the cost of errors, the evidence for and against, and this project as a case (Section 7).

(vi) Three released specialists, measured with their conditions (Section 8). On 164 of the 168 repositories of CredData's test half, `secrets-code` (1.9 M parameters) finds 3.5 times as many labeled credential lines as gitleaks with its default rules, at a close precision on labeled lines (3.1 times on all 168), after learning partly from the benchmark's other half; on the original values, which the benchmark obfuscates, 2.9 times. `pii` (8.3 M) scores F2 0.769 on the PII Masking Benchmark, 13th of 36, above OpenAI's Privacy Filter (0.662; 1.4 B parameters, about 50 M active, with a narrower policy), after

training on the sources of four of its six tasks (on the other two tasks the two models are level). `secrets-bin` reports 1 to 12 false alarms per set of clean binaries (five sets, 193 files), and finds 360 of 360 decidable credentials on a synthetic probe made by its own training generator, since we know of no public, labeled benchmark for that job. None of the three met every pre-registered criterion as first written, and we report each.

(vii) Research on an attachable, exact document memory that, on one constructed 16 MiB stream, recalls reuses at 15 MiB of distance (12 of 12) with 3 MB of state, and a real-data test that led us to withdraw one of its applications (Section 9).

Limits of the claim

We do not claim that small models replace large ones: questions that change every time, need knowledge of the world or ask for free text belong to generalists. We do not claim results beyond the three examples and one desktop machine; we did not measure agents building specialists, generative models on our benchmarks or classical learned baselines; and we present microcontrollers as a target, not a result. Section 11 collects the limits.

Organization. Section 2 reviews related work. Section 3 states the theory of AI Specialists, Section 4 the architecture, Section 5 the runtime and Section 6 the training stack. Section 7 develops Vibe Training, Section 8 the three case studies and Section 9 the research on document memory. Section 10 discusses the theses, Section 11 the limitations, and Section 12 concludes. The appendices give the notation, the configurations, the cost model's parameters, the data and their licenses, and the commands to reproduce the public-benchmark numbers.

2 Background and related work

This section places PureByte among the work it builds on and competes with: what each work shows, under which conditions, and the evidence against our theses next to the evidence for them. Figures that vendors publish about their own products are marked as vendor claims.

2.1 Small language models and the case for them

The position paper. Our title echoes Belcak et al. [3], who argue that small language models (SLMs) are "sufficiently powerful, inherently more suitable, and necessarily more economical for many invocations in agentic systems". An SLM, in their working definition, "is a LM that can fit onto a common consumer electronic device and perform inference with latency sufficiently low to be practical when serving the agentic requests of one user"; as of 2025 that covers most models below 10 B parameters. They argue that agents use a narrow, heavily prompted slice of a model's abilities; that agent calls must follow strict formats, which a model fine-tuned on one format follows more reliably; that any call can go to a different model, so agentic systems are naturally heterogeneous; and that logged calls supply the data to train a specialist for each. They cite estimates that serving a 7 B model is 10 to 30 times cheaper in latency, energy and FLOPs than serving a 70 to 175 B one, and concede that whether centralized serving stays cheaper through economies of scale is open ("the jury is still out"). Their LLM-to-SLM conversion logs an agent's calls, removes personal data from them (itself a detection job), clusters them into tasks and fine-tunes candidate SLMs for each; for three open-source agents they estimate that 40 to 70 % of the LLM calls could go to specialized SLMs, an assessment rather than a measurement. As of September 2026 we found no published rebuttal on arXiv or on the authors' page for correspondence (blogs not searched); the closest challenges are empirical (Section 2.2).

Surveys and model families. Surveys measure 70 open decoder-only SLMs of 100 M to 5 B parameters on devices [4], cover their architectures and compression [5], define SLMs by specialized tasks under

resource constraints, where LLMs underperform and cloud APIs raise privacy concerns [6], find SLMs of 1 to 8 B matching or beating larger models in about 160 papers [7], and review how small models collaborate and compete with LLMs [8]. The families show what data and distillation buy: phi-1 (1.3 B, "textbook quality" data) reaches 50.6 % pass@1 on HumanEval [9]; phi-3-mini (3.8 B) reaches 69 % on MMLU and runs on a phone [10]; Phi-4 (14 B) surpasses its GPT-4 teacher on STEM questions [11]; Gemma 2's 2 and 9 B models, distilled, compete with models two to three times larger [12]; SmolLM2 (1.7 B, about 11 T tokens) beats Qwen2.5-1.5B and Llama-3.2-1B [13]; Qwen3 (0.6 to 235 B) builds its small models from its flagships [14]; Llama 3.2's 1 and 3 B models "fit onto edge and mobile devices" [15]. The smallest are SmolLM's 135 M model [16] and Gemma 3 270M, whose INT4 version used 0.75 % of a Pixel 9 Pro's battery for 25 conversations in Google's tests [17] (a vendor claim). Among generative models, the closest precedent in scale is TinyStories: models of less than 10 M parameters write fluent, consistent stories when the domain is narrow [18].

How AI Specialists differ. AI Specialists share this motivation and differ in five respects. *Output*: a generative SLM answers by generating tokens, a forward pass per token; an AI Specialist answers with the output of a head (a class, labels, a score, a grade, typed byte spans or a label per byte) after one pass over its input windows, and cannot write free text (Section 3). *Input*: raw bytes rather than subword tokens, so every file, protocol or binary is valid input and every answer maps to byte offsets. *Size*: 2 to 54 M parameters (networks of 2 to 4 M plus, for some jobs, hashed n-gram tables), below the smallest model above. *Knowledge*: no pretraining; an AI Specialist is trained from scratch on data for its job, in minutes to an hour on one consumer GPU (Section 6). *Execution*: a dependency-free C++ runtime on the CPU, bit-identical across the kernels and thread counts of a platform (Section 5). AI Specialists meet the size and latency clauses of the definition of Belcak et al. [3], but not its first clause ("is a LM"): they are not language models, since they neither model the probability of text nor generate it. We call them SLMs in the broad sense of small learned models of sequences (Section 10), and the evidence most relevant to them concerns task-specific models.

2.2 Specialists versus generalists

Evidence for specialists. The paper that introduced knowledge distillation also proposed "specialist models" for the fine-grained classes that a full model confuses [19]. Specialization keeps trading generality for size: a 770 M T5 trained with LLM-written rationales outperforms few-shot 540 B PaLM on one benchmark [20]; distilling math reasoning into T5 models lifts their scaling curve "by paying the price of decreased generic ability" [21]; and 4-bit LoRA fine-tunes beat GPT-4 by 10 points on average over 31 tasks [22]. On tasks like those of AI Specialists, fine-tuned small models beat prompted LLMs: in information extraction, LLMs are worse, slower and costlier, and help only by reranking the hard samples that small models pass on [23]; zero-shot ChatGPT "still faces challenges" in sequence tagging [24]; and fine-tuned BERT-style models beat zero-shot GPT-3.5, GPT-4 and Claude Opus in every text-classification case tested [25]. Classical baselines can suffice: with labels, Complement Naive Bayes matches a zero-shot 27 B LLM on AG News (89.1 against 89.0 %), and runs on a CPU 40 to 486 times faster than small LLMs batched on a GPU [26]; a specialist must beat rules and classical baselines too; we compared it with rules only (Sections 8 and 11). The closest published result at PureByte's scale that we know of is a 1.3-million-parameter DOOM player [2]: a ModernBERT encoder with hash embeddings and a classification head, trained on 31,000 human demonstrations, that picks actions from ASCII frames in 31 ms on consumer hardware. With the same inputs, it scored 178 frags in 10 episodes of one scenario, more than all tested LLMs combined (13), which include models up to 92,000 times larger. The evidence is one scenario, but the recipe is close to ours.

Specialists inside agents, and routing. TabAgent replaces the LLM calls that make an agent's closed-set decisions (routing, shortlisting, gating) with a compact classifier trained on execution traces; on AppWorld it keeps task-level success and cuts latency by about 95 % and cost by 85 to 91 % [27], the

pattern that Vibe Training generalizes. Routers pick the cheapest adequate model per query: FrugalGPT's cascade matches GPT-4 at up to 98 % lower cost [28]; Hybrid LLM sends easy queries to a small, edge-deployable model and calls the large one up to 40 % less without quality loss [29]; learned routers cut costs by more than half in some settings without losing quality [30].

Counter-evidence. Where the task needs knowledge, a well-prompted generalist can win: with general-purpose prompting (Medprompt), GPT-4 outperforms domain-tuned specialists such as Med-PaLM 2 on all nine MultiMedQA benchmarks [31]. Imitation copies style, not capability: models of 1.5 to 13 B that imitate ChatGPT close "little to none of the gap" on tasks not covered by the imitation data [32]. Larger models are more sample-efficient [33], some abilities appear only at scale [34] (many, though, only under certain metrics [35]), and "general methods that leverage computation are ultimately the most effective" [36]. Small generative models also fail as drop-in agents: with Gemma-3 4B and Qwen-3 1.7B, four issue-resolution frameworks resolved almost no task of SWE-bench Verified Mini, their energy differing by up to 9.4 times and going largely to unproductive reasoning loops [37]; across 16 open models of 0.27 to 32 B, small ones suffice for short, structured tool use while frontier models keep an edge in long-horizon planning [38]. Part of the failure lies in the harness: a failed call left in the transcript raises the probability of repeating it from 0.06 to 0.54 in models of 135 M to 1.7 B, and a harness change removes 76 % of the effect [39]; an adapted harness lets the best SLM agent recover 89.7 % of an LLM's performance at 4 % of its cost [40]. We read these results as limits of our claim. They concern knowledge, breadth and open-ended agency, whereas AI Specialists are proposed only for jobs whose answer depends on local evidence in the bytes they read, with training data that cover the inputs' forms (Section 3.3), as tools that programs and agents call. And the bitter lesson cuts both ways: it favors learning over hand-written rules, but also general methods and scale over hand-sized models, and our training generators encode hand-written knowledge (Section 7.3).

2.3 Byte-level models

Byte-level models remove the tokenizer. ByT5 shows that a standard Transformer on bytes is competitive with token models, "significantly more robust to noise" and better on spelling-sensitive tasks [41]; CANINE, a character encoder with downsampling, beats a comparable mBERT by 2.8 F1 on TyDi QA with 28 % fewer parameters [42]. Later models reshape the architecture for long byte sequences: MEGABYTE with local and global patch models, over more than a million bytes [43]; MambaByte with a selective SSM, competitive with and sometimes better than subword Transformers [44]; SpaceByte with larger blocks after word boundaries, roughly matching tokenized Transformers at fixed compute [45]; the Byte Latent Transformer with patches sized by next-byte entropy, matching token models in a FLOP-controlled study up to 8 B parameters and 4 T bytes, with better inference efficiency and robustness [46]; and H-Net with learned chunking, beating a BPE Transformer at matched compute and data [47]. bGPT predicts bytes of any modality [48]; MalConv detects malware from whole executables [49]; ByteFormer classifies images and audio from file bytes [50]. Bytes are not free: byte and character encodings still differ more than 4-fold in length between some languages [51], and large generalists are strong byte predictors, at a large compute cost [52].

Nearly all of this work is generative modeling at scale. PureByte uses bytes as the one input shared by text, code, logs, binaries and packets, with exact offsets for the answers; its models are small non-generative networks with task heads, closer to CANINE, MalConv and ByteFormer. MambaByte is the closest in ingredients (bytes and a selective SSM), but it generates, with 353 M to 1.6 B parameters, and its speed is reported on an A100 GPU [44].

2.4 State-space models

State-space models (SSMs) carry a fixed-size state through a recurrence, so their cost is linear in length and their memory constant. S4 reached the state of the art on every Long Range Arena task [53]; Mamba made the recurrence selective (input-dependent), with five times the inference throughput of

Transformers and a 3 B model that outperforms Transformers of its size [54]; Mamba-2's structured state-space duality relates SSMs to attention and makes the core layer 2 to 8 times faster [55], and PureByte's blocks belong to this family (Section 4); Mamba-3 adds a complex-valued state and matches Mamba-2's perplexity with half the state size [56]. The fixed state is also the known weakness. Jelassi et al. [57] prove that fixed-state models are fundamentally limited at copying and find pretrained Transformers far better at copying and retrieval; in Zoology, a 70 M attention model beats a 1.4 B gated-convolution model at associative recall, and associative recall explains 82 % of the perplexity gap between gated convolutions and attention [58]. At 8 B parameters on identical data, pure SSMs lag on copying and in-context learning, while a hybrid with 7 % attention layers beats the Transformer on all 12 standard tasks [59]; such hybrids are now common [60–63]. And a hardware-in-the-loop search under edge constraints chose gated short convolutions and a few attention blocks for LFM2, not SSM layers, giving up to twice faster CPU prefill and decoding than similar-size models [64]. For PureByte the weakness is bounded: an AI Specialist reads 512-byte windows and its jobs need mostly local evidence (Section 3), so the fixed state costs little within a window and much across a document, as Section 9 measures; Section 4 compares state-space, attention and other blocks at PureByte's scale.

2.5 Low-bit networks

Binary weights turn multiply-accumulates into accumulations [65]; XNOR-Net saves 32 times the memory and runs convolutions 58 times faster on CPUs [66]; ternary weights in $\{-1, 0, +1\}$ with a scale compress up to 16 times and beat binary ones [67]; learned ternary values match or beat full-precision ResNets on CIFAR-10 [68]; training relies on the straight-through estimator [69]. For language, BitNet trains 1-bit Transformers from scratch with a scaling law like that of full precision [70], and BitNet b1.58 is reported to match FP16 models of equal size and training tokens with ternary weights, at lower latency, memory and energy [71]. The open BitNet b1.58 2B4T (2 B parameters, 4 T tokens) is on par with full-precision peers [72]; its CPU runtime, bitnet.cpp [73], reports speedups of 2.37 to 6.17 times on x86 and 1.37 to 5.07 times on ARM CPUs across model sizes [74]; ternary layers also combine with a recurrent token mixer [75]. Ternary SSMs exist: Slender-Mamba trains BitNet-style ternary Mamba models from scratch [76], the direct precedent for PureByte's blocks; binary (1-bit) Mamba models of 780 M to 2.7 B trained from scratch with distillation are comparable to full precision [77]; grouped ternary quantization-aware training with 16-bit activations, PureByte's format, has been applied to a pretrained Mamba-2, where learnable scales caused a "zero-ratio collapse" and corrections that work for Transformers failed because errors accumulate through the recurrence [78] (PureByte's scales are computed, not learned); a 1.58-bit Mamba-2 has been run on ARM CPUs [79]; and SSM activations have large outliers [80]. Hashed n-gram tables with several hash heads, a context gate and an injection layer are the design of Engram [81], which PureByte uses over bytes (Section 4.3); hashed character n-grams go back to fastText [82].

The ternary penalty at tiny scale. The evidence that ternary weights cost little comes from large models: ternary models beat quantized and float models of equal size in bits only above one billion parameters [83], and a ternary 600 M model beats the previous best ternary 3 B one [84]. The one controlled study at tiny scale that we know points the other way. In 98 byte-level runs at about 60,000 parameters, three seeds each, the depth and width of the baseline moved validation loss more than any architecture effect at a 16 M-byte budget, and at 130 M bytes the ternary penalty was +28.1 % for a gated SSM against +5.3 % for the best Transformer, a gap the authors do not attribute to architecture alone because their Transformers kept 11 to 22 % of their parameters in full precision [85]. At a few million parameters, ternary weights are thus a deployment choice with a price in quality, to be measured against tuned baselines over several seeds; PureByte makes only the two large projections of each block ternary and reports the price in Section 4.

2.6 TinyML and edge AI

Microcontrollers have 100 to 1,000 times less compute, memory and power than mobile processors; TensorFlow Lite Micro, an open-source interpreter, runs models in a few kilobytes on hardware that often lacks dynamic memory allocation [86], and CMSIS-NN's Cortex-M kernels raise throughput 4.6 times and energy efficiency 4.9 times [87]. MCUNet co-designs the network (by architecture search) and the engine for memories "2-3 orders of magnitude smaller even than mobile phones", and was the first to exceed 70 % ImageNet top-1 on an off-the-shelf microcontroller [88]; its successor exceeds 90 % on Visual Wake Words within 32 kB of SRAM [89]. MLPerf Tiny measures accuracy, latency and energy per inference on keyword spotting, visual wake words, image classification and anomaly detection [90]; in its v1.2 keyword-spotting results, three STMicroelectronics boards used a median of 656 to 1,820 µJ per inference, at 23 to 89 inferences per second and about 23 to 122 mW [91]. A small language model trained on TinyStories runs on a RISC-V microcontroller with an NPU at 490 µJ per token [92]. Vendor pages give typical budgets: 256 KB of RAM and 1 MB of flash (nRF52840, a 64 MHz Cortex-M4) [93]; 264 and 520 kB of SRAM (RP2040, RP2350) [94]; 512 KB of SRAM with vector instructions for neural networks (ESP32-S3) [95]; 1 MB of SRAM (i.MX RT1060, a 600 MHz Cortex-M7) [96]; and up to 13.5 MB of SRAM with two Cortex-M55 cores and two Ethos-U55 NPUs (Ensemble E3) [97]. Many routers "do not have the memory or processing power to train and sometimes even execute such models" [98], yet binary networks in SmartNICs classify traffic with up to 100 times lower latency at 40 Gbps [99], and a switch pipeline runs per-packet inference for 3.8 % more area [100]. On robots, a DNN navigates a 27 g nano-drone at 64 mW [101], a reinforcement-learning policy runs on a Crazyflie's Cortex-M4 [102], and the field calls for "automated end-to-end design tools" [103]. These results concern images, audio, sensor signals and network traffic. PureByte applies sequence models to byte streams, runs today on x86-64 and arm64 CPUs (tested on servers, desktops and laptops, not yet on Linux-class devices such as routers or small boards), and treats microcontrollers as its next target, not as a result (Sections 5 and 10).

2.7 The cost and energy of inference

Training costs were measured first: their financial and carbon cost [104], the call to report a model's "price tag" [105], and footprints that the choice of model, data center and processor can change by 100 to 1,000 times [106]. For inference, the anchor of this paper is Luccioni et al. [1], who measured 88 models on 10 tasks and 30 datasets (1,000 inferences per dataset, A100 GPUs, CodeCarbon). Mean energy per 1,000 inferences was 0.002 kWh for text classification and 0.004 kWh for token classification, against 0.047 kWh for generating 10 tokens per input and 2.9 kWh for image generation. Against eight zero-shot multi-purpose models (BLOOMz, Flan-T5) on the same datasets, they find that "multi-purpose, generative architectures are orders of magnitude more expensive than task-specific systems for a variety of tasks, even when controlling for the number of model parameters". In text classification, however, the multi-purpose models had the higher average accuracy: the saving is large, but it is not free.

Per query, generative inference costs little in absolute terms. Google measured its median Gemini Apps text prompt in production at 0.24 Wh, after a 33-fold fall in one year [107]; OpenAI's chief executive gave "about 0.34 watt-hours" per average ChatGPT query, without a published method [108]; Epoch AI estimated about 0.3 Wh for a GPT-4o query, assuming 500 output tokens and H100 GPUs at 10 % utilization [109]. For scale (our conversion), 0.24 to 0.34 Wh is 860 to 1,220 J, the energy of roughly 5×10^5 to 2×10^6 keyword-spotting inferences on the boards of Section 2.6; the tasks are not comparable, and the ratio shows only orders of magnitude. In aggregate, data centers used about 415 TWh in 2024, 1.5 % of the world's electricity, and are projected to reach about 945 TWh by 2030, with AI as the main driver [110]. Price trends cut against the cost argument: the price of a fixed level of performance fell 9 to 900 times per year depending on the benchmark [111], and the cost at fixed quality about ten times a year [112]. If these trends continue, the gap in marginal cost narrows, and the case for specialists must rest also on latency, privacy, determinism and where they can run (Section 7); but cheaper AI drives disproportionately more use [113], so total energy can rise while the cost per query falls.

2.8 Agents that engineer and make models

Agents that do machine-learning engineering. On MLAgentBench, the best agent (Claude v3 Opus) averages 37.5 % success over 13 tasks, from 100 % on well-established datasets to 0 % on recent Kaggle challenges [114]; on MLE-bench, o1-preview with the AIDE scaffold, which frames ML engineering as tree search over code [115], earns at least a bronze medal in 16.9 % of 75 Kaggle competitions [116]. AutoKaggle reaches a validation submission rate of 0.85 over eight competitions [117], and AutoML-Agent goes from a task description to a deployment-ready model [118]. The AI Scientist writes ML papers for less than $15 each, some above a top conference's acceptance threshold as judged by its authors' own automated reviewer [119]. In autoresearch (a repository, not a paper), an agent edits the training script of a small language model, trains for five minutes on one GPU, and keeps or discards each change by validation bits per byte [120]. Agents beat human experts on RE-Bench within two hours but not within 32 [121], improve baselines "usually by finding better hyperparameters" [122], and reach an average replication score of 21.0 % on ICML papers, below the human baseline [123]: they do well in bounded loops with a clear metric, which is what Vibe Training asks of them.

Agents that make models. The closest precedent of Vibe Training is Program-as-Weights [124]: a 4 B "compiler", trained on a released dataset of 10 million examples, turns a natural-language specification of a "fuzzy function" (alerting on important log lines, repairing malformed JSON) into adapters for a frozen 0.6 B Qwen3 interpreter, which then matches direct prompting of Qwen3-32B with about one fiftieth of the inference memory, at 30 tokens per second on a MacBook M3. The foundation model becomes "a tool builder: invoked once per function definition", which is the stance of Vibe Training. PAW turns a specification, with one call of the compiler, into an adapter for a shared generative interpreter; in Vibe Training an agent builds data, criteria fixed in advance and a held-out examination, and trains a separate byte-level model from scratch (Section 7). Pioneer Agent, "given only a natural-language task description", acquires data, builds evaluation sets and trains small language models, gaining 1.6 to 83.8 points over base models on eight benchmarks and raising intent classification from 84.9 to 99.3 % in a production-style deployment built from public tasks [125]. In agentic knowledge distillation, a teacher LLM generates data and fine-tunes Qwen2.5-0.5B or SmolLM2-135M for SMS threat detection with no human in the loop, reaching 94.31 % accuracy at best, with results that depend strongly on the teacher [126]. LLM agents also design microcontroller networks in 1 to 2 hours instead of hundreds of GPU hours of hardware-aware search [127]. None of these trains byte-level models from scratch for byte jobs. Older roots include AutoML [128–130], data programming [131], LM-generated data for a "tiny task model" [132] and Software 2.0 [133]. The name comes from vibe coding, which validates AI-generated code "through outcome observation rather than line-by-line code comprehension" and for which empirical studies report "unexpected productivity losses" [134]; Vibe Training adds criteria fixed before training and an examination the model is not trained on; Definition 10 says which of these rules the factory enforces.

Small models as tools of agents. Language models call tools because they "struggle with basic functionality, such as arithmetic or factual lookup, where much simpler and smaller models excel" [135]; a fine-tuned LLaMA writes API calls better than GPT-4 [136]; an LLM can dispatch subtasks to specialized models [137]; and the Model Context Protocol gives agents a standard channel to such tools [138]. Function calling itself runs on small models: Octopus v2 (2 B, on the device) surpasses GPT-4 in accuracy and latency [139], and TinyAgent's 1.1 and 7 B models match or surpass GPT-4-Turbo on a MacBook [140]. The nearest product is Cactus Needle [141], a single 8 to 29 MB, 2-bit binary for phones, wearables, robots, cars and microcontrollers that "trade[s] general chat capacity to beat models 10x its size on mobile tool calls". Most of its parameters sit in an n-gram "engram" memory read by gather, so that its 121 M model "does the arithmetic of a 50M one"; after fine-tuning on DroidCall, subnetworks from 29 M parameters up are reported to pass DeepSeek V4 Flash; and fine-tuning can be delegated to Claude Code or Codex (self-reported figures; telemetry is on by default). Needle shares PureByte's bet

on tiny specialized models and on tables that add capacity without arithmetic, but it generates tokens, under a byte-level grammar, and is specialized by fine-tuning a pretrained base; an AI Specialist reads bytes, answers without generating and is trained from scratch for one job.

2.9 Detection of secrets and personal data

The example specialists of Section 8 detect credentials in code and binaries and personal data in text. *Secrets* leak widely: in "over 100,000 repositories" on GitHub [142], 28,649,024 new ones in public GitHub commits in 2025 by a vendor's count [143], and in 8.5 % of the container images studied [144]. The standard scanners are rule-based: gitleaks [145]; TruffleHog, which also verifies credentials live [146]; detect-secrets [147]; and CredSweeper, which can filter candidates with a machine-learning model [148]. On a benchmark of nine tools, the best precision was 75 % (GitHub's scanner; gitleaks 46 %) and the best recall 88 % (gitleaks), with errors traced to generic or faulty regular expressions, poor entropy measures and skipped file types [149]. CredData's README describes 19,459,282 lines from 297 repositories, of which 73,842 are labeled and 4,583 are true credentials; the commit we measure (`c09c0c5`, September 2026) has 337 repositories, split by us into halves of 169 and 168 (Section 8.1). The README reports F1 of 0.859 for CredSweeper, 0.334 for gitleaks and 0.235 for truffleHog3, from a benchmark run by CredSweeper's maintainers on old tool versions "by combining training and test data" [150]. Learned detectors improve on rules: a voting classifier cuts the false positives of regular expressions [151]; a fine-tuned 8 B LLaMA-3.1 that classifies regex candidates reaches F1 0.9852 [152]; in GitHub issue reports, fine-tuned RoBERTa and CodeBERT reach F1 92.70 % against 80.13 % for few-shot GPT-4o [153]; and a Transformer over string-literal context reaches F1 98.74 % [154] on SecretBench [155]. Most of these classify candidates that rules or a parser extracted; PureByte's secret specialists tag byte spans directly, in source files and in binaries.

For *personal data*, Presidio combines named-entity recognition, patterns and custom pipelines [156]; GLiNER, a compact bidirectional encoder, beats ChatGPT and fine-tuned LLMs at zero-shot entity extraction [157]; GLiNER2 stays efficient on CPU [158]; and GLiNER2-PII (0.3 B) covers 42 PII types at character-span resolution [159]. OpenAI's Privacy Filter is the closest in form to PureByte's `pii`: a bidirectional token classifier converted from an autoregressive checkpoint "with similar architecture to gpt-oss, albeit of a smaller size", with 1.5 B total and 50 M active parameters according to its model card (1.40 B counted by the PIIMB leaderboard), eight categories tagged with BIOES labels under a constrained Viterbi decoder, and tunable operating points; it runs in a browser or on a laptop, and reads tokens rather than bytes [160]. The PII Masking Benchmark scores zero-shot masking at the character level, independently of label names, on six tasks with about 25,000 documents and 152,000 entities (from AI4Privacy, Nemotron-PII, Gretel, Privy and MAPA EUR-LEX), ranked by masking F2; contamination is left to an honor system, and its card warns that "for specialised use cases, this benchmark does not provide the full picture" [161]. Language models trained on personal data leak it, about 3 % of PII sequences even under sentence-level differential privacy [162]; a distilled de-identifier trails its cloud teacher on per-category recall (0.81 against 0.90), and because institution-specific entities transfer poorly, its authors pair broad models with specialized ones [163].

2.10 Arm: the facts behind the analogy

Section 10.3 compares PureByte's aims with Arm's history and relies on these facts. Arm "began as a joint venture between Acorn Computers, Apple Computer, and VLSI Technology" in 1990, to build a processor that was "high performance, power efficient, easy to program, and readily scalable", and its CPUs spread in mobile phones in the mid-1990s "because our energy-efficient processors provided an appropriate level of performance while consuming little power" [164]; the first ARM silicon (1985) ran at up to 6 MHz on "about 0.1 W" [165]. Customers pay a license fee and then "a per-unit royalty on substantially all chips shipped" [164]; under Arm Flexible Access, a "nominal fee per year" opens more than 80 components [166]. More than 350 billion Arm-based chips had been reported shipped by 31 March 2026; in March 2026 Arm began selling its own production silicon, a departure from pure

licensing [164]. Arm's architecture is licensed, not open: the case for free, open instruction sets was made for RISC-V [167], and open instruction sets are expected to help lead computer architecture [168]. PureByte's openness (format, runtime and training stack under Apache-2.0) is thus closer to RISC-V's; what it takes from Arm is efficiency as the design goal.

2.11 Positioning

Byte- and character-level classifiers trained from scratch for one security job are a decade old: MalConv reads whole executables [49], eXpose reads URLs, file paths and registry keys [169], and character-level convolutional networks classify text [170]. What PureByte adds to that lineage is a ternary state-space body with typed heads, a declared model format, a deterministic CPU runtime and a factory that trains a specialist in minutes to an hour. Among the neighbors in Table 1, none combines PureByte's three defining choices: byte input, answers without generation, and models trained from scratch for one job in minutes to an hour. Byte-level language models share the input but generate; PII detectors share the output but read tokens and start from pretrained models; Needle shares the size and the specialization but generates and fine-tunes a pretrained base; Program-as-Weights and agent-trained SLMs share the maker, an AI working from a description, but adapt generative models of 135 M to 0.6 B parameters; rule scanners share plain CPU execution but do not learn. The table compares designs, not quality; Section 8 measures quality against the tools that share PureByte's jobs.

3 A theory of AI Specialists

This section states what an AI Specialist is, which questions it can answer, when it can do without general knowledge, and what that saves. It is the conceptual core of the paper, and a formulation rather than a theory with theorems: definitions, one observation about output types, an argument about knowledge and an estimate of cost. The architecture that implements it follows in Section 4, and the evidence that bears on each claim in Section 8.

3.1 Jobs, specialists and generalists

Let $\mathcal{B} = \{0, 1, \ldots, 255\}$ be the set of byte values and $\mathcal{B}^*$ the set of finite byte strings.

Definition 1 (Byte job)**.** A *byte job* is a tuple $J = (\mathcal{Y}, P_J, \ell)$: an output space $\mathcal{Y}$, a distribution P_J over pairs $(x, y) \in \mathcal{B}^* \times \mathcal{Y}$ that describes the inputs the job will meet and their correct answers, and a loss $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_{\geq 0}$ that says what a wrong answer costs.

Definition 2 (AI Specialist)**.** An *AI Specialist* for a job J is a model $f_\theta : \mathcal{B}^* \to \mathcal{Y}$ whose output space is a product of the types of Observation 3, trained for that job alone, on examples drawn from (or built to resemble) P_J, with a surrogate of the expected loss $\mathbb{E}_{(x,y)\sim P_J}\, \ell(f_\theta(x), y)$, and with a size $|\theta|$ chosen for that job. It is trained *from scratch* when none of its parameters comes from a model trained for other jobs, as the three released specialists are, and *fine-tuned* otherwise. A *generalist* is a model trained to serve many jobs at once, typically through natural language, whose parameters are shared by all of them.

The definitions are deliberately plain. The claims of this paper are about specialists trained from scratch, whose size $|\theta|$ is chosen for one job, whose input is raw bytes and whose output is one of a small set of structured types. Small pretrained encoders fine-tuned for one job are specialists too in this sense, and they are the strongest alternative to ours (Section 3.3).

3.2 Any byte is trainable, and so is the output

Every digital input is already a byte string: text in any encoding, source code and minified code, logs, JSON and CSV, network payloads, archives, executables and firmware, sensor frames. A model that

Table 1: PureByte and the closest systems or families. Each entry is what the cited sources state, vendor claims included; "not stated": the sources we use do not say. M, B: millions, billions of parameters. KD: knowledge distillation.

System or family	Input unit	Output	Parameters	Runs on	Trained by	Open
PureByte AI Specialists (this paper)	raw bytes	decisions, labels, scores, byte spans; no generation	2–54 M (networks 2–4 M; the rest *n*-gram tables)	x86-64 and arm64 CPUs (C++17 runtime)	a person or an agent (Vibe Training), from scratch, minutes to an hour on one GPU	runtime, format, training stack (Apache-2.0); example weights: use only (PureByte Model License)
Generative SLMs: Phi, Gemma, SmolLM, Qwen, Llama 3.2 [10–16]	subword tokens	generated text	135 M–14 B	phones, edge devices (phi-3-mini, Llama 3.2)	vendors: pretraining on trillions of tokens, distillation	weights released (Gemma, SmolLM, Qwen, Llama)
BitNet b1.58 2B4T, bitnet.cpp [72, 74]	tokens	generated text	2 B, ternary	x86 and ARM CPUs; GPUs	vendor; natively ternary, 4 T tokens	weights, inference code
Byte-level LMs: MambaByte, BLT [44, 46]	bytes (BLT: entropy patches)	generated bytes	353 M–1.6 B; up to 8 B (BLT)	A100 GPU (MambaByte); not stated (BLT)	authors, from scratch (BLT: up to 4 T bytes)	not stated
TinyML: TFLM, MCUNet [86, 88]	images, audio	classes (ImageNet, wake words)	not stated; kB to MB of memory	microcontrollers	engineers; MCUNet by architecture search	TFLM open source; MCUNet code, models
Program-as-Weights [124]	text; a spec in natural language	generated text	4 B compiler; 0.6 B interpreter + adapters	a laptop (MacBook M3)	a 4 B compiler, once per function	training data; rest not stated
Agent-trained SLMs: Pioneer Agent, agentic KD [125, 126]	tokens	generated text or labels	135 M, 0.5 B (KD); not stated (Pioneer)	on-device (KD); not stated (Pioneer)	an LLM agent, from a pretrained SLM	not stated
Cactus Needle [141]	tokens (byte-level output grammar)	tool calls, fields, embeddings	29–121 M; 8–29 MB file	phones, wearables, robots, cars, MCUs (vendor)	vendor; fine-tuned by the user or a coding agent	Apache-2.0; telemetry on by default
Secret scanners: gitleaks, TruffleHog, detect-secrets [145–147]	lines and files of text	matched secrets; live verification (TruffleHog)	none (rules)	CPU	rules written by maintainers	MIT; AGPL-3.0; Apache-2.0
PII detectors: Presidio, GLiNER2-PII, Privacy Filter [156, 159, 160]	tokens	typed spans; redaction (Presidio)	0.3 B (GLiNER2-PII); 1.5 B total by its card (1.4 B by the PIIMB leaderboard), 50 M active (Privacy Filter)	CPU (GLiNER2); browser, laptop (Privacy Filter)	vendors, from pretrained encoders or LMs; Presidio adds patterns	MIT; Apache-2.0; GLiNER2-PII released

reads bytes has one input interface for all of them and needs no tokenizer, whose vocabulary is learned from a corpus and splits strings unlike that corpus (a credential, a base64 blob, a machine instruction) into pieces that depend on the corpus, not on the job [41]; every position of its answer maps back to an exact byte offset. By "any byte is trainable" we mean exactly this: any byte string is a valid input, for training and at run time. It does not mean that every byte string carries a pattern a model can learn: the content of a compressed or encrypted stream does not, until something decodes it.

The output side is structured the same way. PureByte's model format declares *heads*, each of which turns the hidden states of a window into one kind of answer (Section 4.7):

- `choice`: one of K classes (a decision, including "nothing here");
- `multilabel`: a subset of K properties;
- `score`: a real number;
- `ordinal`: a grade in an ordered scale of K levels;
- `tag`: typed byte spans, decoded from BIOES labels with a constrained Viterbi search;
- `byte_map`: one label per byte.

A seventh head, `exit`, is not an answer: it is a probe that can stop the reading of a window early on an easy input (Section 4.7). A post-processing *profile* may then act on the spans, for example to write a redacted copy of the input in which every tagged span is replaced by a placeholder.

Observation 3 (What the output types cover)**.** *Let $g : \mathcal{B}^* \to \mathcal{Y}$ be the answer function of a job. If $\mathcal{Y}$ is (a) a finite set, (b) the set of subsets of a finite set, (c) $\mathbb{R}$ (a* `score` *head is an affine map and does not enforce a range), (d) a finite ordered set, (e) a set of non-overlapping typed byte ranges of the input with types from a finite set, or (f) a sequence of labels from a set of 2 to 256 labels, one per input byte, then g has the output type of one PureByte head, and a product of at most 64 such types has the type of a set of heads (the value of a gated head may be absent). A transformation that copies the input and replaces the byte ranges of (e) by fixed strings is the composition of a* `tag` *head and a redaction profile.*

Each case matches one head's declared output; a model file may declare several heads over one body, whose outputs form the product. BIOES sequences over typed labels represent exactly the sets of non-overlapping typed spans, and the constrained decoder emits only valid sequences; the redaction profile copies every byte outside the tagged spans and writes the placeholder of each span's type in its place. The observation concerns the types of answers a model file can express, not which functions a model can compute or learn. In the window context a head sees one window, and a document's answer is an aggregate of window answers (Section 4.8), so a job whose answer depends on bytes farther apart than a window is outside the released configuration whatever its type; whether a small model can learn g from the available data is an empirical question, answered job by job in Section 8. It also marks the boundary of the approach. Free text (new bytes that are not a copy of the input) and unbounded structures are outside it: an AI Specialist decides, scores, labels, locates and redacts; it does not write.

Training follows the same abstraction: a job is written as a task file that declares the output schema (the labels and their types), and the training stack builds the corresponding heads and losses. Today the training stack trains the span type (e) with a gating decision (a), which is what the three released specialists answer; the other types are implemented in the runtime and in the reference implementation, and no trained model of them exists yet (Section 6).

3.3 Why a specialist does not need general knowledge

For a fixed job, the best possible answer to an input x depends only on the conditional distribution $P_J(y \mid x)$: under loss ℓ, the Bayes decision is $f^*(x) = \arg\min_{\hat{y}} \mathbb{E}_{y \sim P_J(\cdot \mid x)}\, \ell(\hat{y}, y)$. Everything a model knows that does not change $P_J(y \mid x)$ on the inputs the job meets is, for that decision, dead weight. Formally, if a statistic $T(x)$ is sufficient for y under P_J, that is $P_J(y \mid x) = P_J(y \mid T(x))$, then a model that computes T and maps it to the answer loses nothing.

Sufficiency concerns the decision, not how it is learned. Knowledge that leaves $P_J(y \mid x)$ unchanged can still reduce the labeled data needed to estimate it [171] and the damage done by a shift in the inputs [172]; and a specialist is trained on data generated or borrowed to resemble P_J, not drawn from it. Whether a small specialist trained from scratch reaches the quality a job needs is therefore an empirical question, job by job.

For byte jobs, the sufficient statistics are often local. Whether a quoted string is a credential rather than a placeholder depends on its characters, its length, the name of the variable or key it is assigned to, the file type and a few lines of context; whether a token is a person's surname depends on its shape, its neighbors, the document's form and a vocabulary of names, which the training data must supply. A specialist has to carry only these features. The released examples test how far that goes (Section 8):

- `secrets-code`, a network of 1.9 million parameters trained from scratch, finds three and a half times as many labeled credential lines as gitleaks with its default rules, at a close precision on labeled lines, on 164 of the 168 repositories of CredData's test half (the other four overlap its training corpus; on all 168 the ratio is 3.1). It learned partly from labeled lines of CredData's other half, whose credential values the benchmark randomizes in the same way: with the original values restored, its F1 falls from 0.797 to 0.694 and its lead to 2.9 times, while gitleaks barely moves (Section 8.2).
- `pii`, with 8.3 million parameters, 6.3 million of them in lookup tables, scores F2 0.769 on the PII Masking Benchmark, 13th of 36 models, above OpenAI's Privacy Filter (0.662; 1.4 billion parameters, about 50 million active per token). It was trained on documents of the datasets behind four of the benchmark's six tasks (the filter's model card names no training data), and it masks with the benchmark's broad policy, which the filter does not; on the two tasks from sources `pii` did not learn from, the two are level (Section 8.3).
- The best small encoders fine-tuned from general pretraining score higher on the same benchmark, at a few times `pii`'s operations per byte (Table 2): a fine-tune of DeBERTa-v3-small [173, 174] scores 0.816, and one of a 68-million-parameter Ettin encoder [175, 176] 0.778, each trained on one of `pii`'s four sources; the 32-million-parameter Ettin fine-tune scores 0.755, below `pii`.

Our evidence therefore shows that general pretraining is not *necessary* to reach useful quality on these jobs, not that it is useless: a network trained from scratch beats a mature rule set on recall, and the best fine-tuned small encoders, which are specialists in our sense, stay ahead on the PII benchmark at several times the cost per byte.

Thesis. Specialist tasks do not need generalist knowledge

When the answer to a job depends on local evidence in the bytes, and the training data cover the forms the inputs take, a model sized for that evidence and trained from scratch on the job's data can reach the quality of the tools in use, with orders of magnitude fewer parameters than a generalist and a fraction of the cost per byte of a pretrained encoder. *Train with your data, not a generalist*: a generalist is not required. The thesis is refuted for a job if a specialist trained on adequate data stays clearly below a pretrained model of similar cost.

The same argument names its exceptions. When the answer depends on facts outside the bytes (who a company is, whether a person is famous, what a law says), no amount of byte patterns replaces knowledge, and a generalist or a database is the right tool. And a specialist knows only the distribution it was trained on: on the PII benchmark, most of `pii`'s gap to the twelve models above it lies in two kinds of text that resemble nothing in its training data (protocol traces and European legal texts). We attribute that gap to data coverage rather than to knowledge, but we have not tested it, for instance by adding such texts to the training data and measuring again.

Table 2: Order-of-magnitude cost per byte read. Specialists: their measured sizes (Section 4.10), with operations estimated as $2|\theta_{\text{net}}| \times 1.33$ for the overlap of the windows. Transformers: $2|\theta|$ operations per token [33] at an assumed four bytes per token, counting the parameters used per token (the backbone of the DeBERTa encoder, the whole Ettin encoder as an upper bound, the active parameters of the Privacy Filter), with weights at 16 bits. The encoders are the two fine-tunes of Section 3.3; the dense generative model is illustrative, and its row stops before it generates any answer. Weights: the size of the model files, or of the transformers' weights at 16 bits; runtime 1.0.0 also expands each ternary weight to a byte in memory, so the specialists occupy 2.7, 6.3 and 32.3 MiB when loaded (Section 5.5). Estimates of work, not measurements of time or energy.

Model	Parameters used per byte	Operations per byte	Weights
`secrets-code`	1.93×10^6	$\approx 5 \times 10^6$	0.98 MiB
`pii`	1.93×10^6 + 3 table rows	$\approx 5 \times 10^6$	4.5 MiB
`secrets-bin`	3.79×10^6 + 3 table rows	$\approx 1 \times 10^7$	28.8 MiB
DeBERTa-v3-small fine-tune	4.4×10^7	$\approx 2.2 \times 10^7$	270 MiB
Ettin encoder fine-tune, 68 M	$\leq 6.8 \times 10^7$	$\leq 3.4 \times 10^7$	131 MiB
Privacy Filter	5×10^7 active of 1.4×10^9	$\approx 2.5 \times 10^7$	2.6 GiB
Dense generative model, 8 B	8×10^9	$\approx 4 \times 10^9$	14.9 GiB

3.4 Making AI more efficient: the cost of an answer

The work a model does per byte is roughly proportional to the number of parameters it touches per byte. A dense transformer performs about $2|\theta|$ floating-point operations per token it reads [33], once for an encoder that labels tokens, and once per token read and again per token generated for a generative model; a token of English prose is about four bytes. A specialist with a network of $|\theta_{\text{net}}|$ parameters reads each byte once per window; with the examples' windows of 512 bytes every 384, each byte is read $512/384 \approx 1.33$ times, at about $2|\theta_{\text{net}}|$ operations per byte. Its n-gram tables add three row reads per byte, whatever their size. Its large projections are ternary, which shrinks the file to 2.25 bits per weight (Section 4.5); runtime 1.0.0 still multiplies every weight as a float (Section 5.3), because an integer or additions-only kernel would change the bits that the numerics contract fixes (Section 5.2), so the ternary weights save bytes on disk, not operations; such a kernel, with its own contract, is future work.

Table 2 and Figure 2 put the orders of magnitude side by side. Against a small generative model, a specialist does 400 to 800 times fewer operations per byte and has 500 to 16,000 times fewer bytes of weights on disk, before the generative model has produced a single token of its answer. Against the small pretrained encoders that compete with `pii` on the PII benchmark, the margin is 4 to 7 times in operations and 30 to 60 times in bytes of weights, and against the Privacy Filter 5 times in operations. Against rules the comparison goes the other way: on our machine gitleaks reads bytes roughly 80 times faster than `secrets-code` (a busy-machine timing, so approximate); the specialist spends that extra work on the credentials that have no fixed form.

Time and energy follow operations and memory traffic only on the same hardware. We did not measure energy, and the comparison with a generative model crosses hardware: the specialist runs on a desktop CPU, a generative model on a data-center GPU, whose efficiency per operation is one to two orders of magnitude higher, and estimates from operations misjudge real energy [177]. A rough bound, under stated assumptions: at the measured 131 kB/s of `secrets-code` on the whole desktop and the processor's rated package power of 105 to 142 W, the specialist spends about 1 mJ per byte; the illustrative 8-billion-parameter model on an NVIDIA H100 (about 990 dense BF16 teraFLOPS and up to 700 W [178]) running at 10 to 40 % of that peak spends about 7 to 28 mJ per byte. By that estimate the energy per byte differs by 7 to 28 times, one order of magnitude rather than the two to three of the operations. The larger saving is elsewhere: the specialist needs no accelerator at all, so it can run where the data already are.

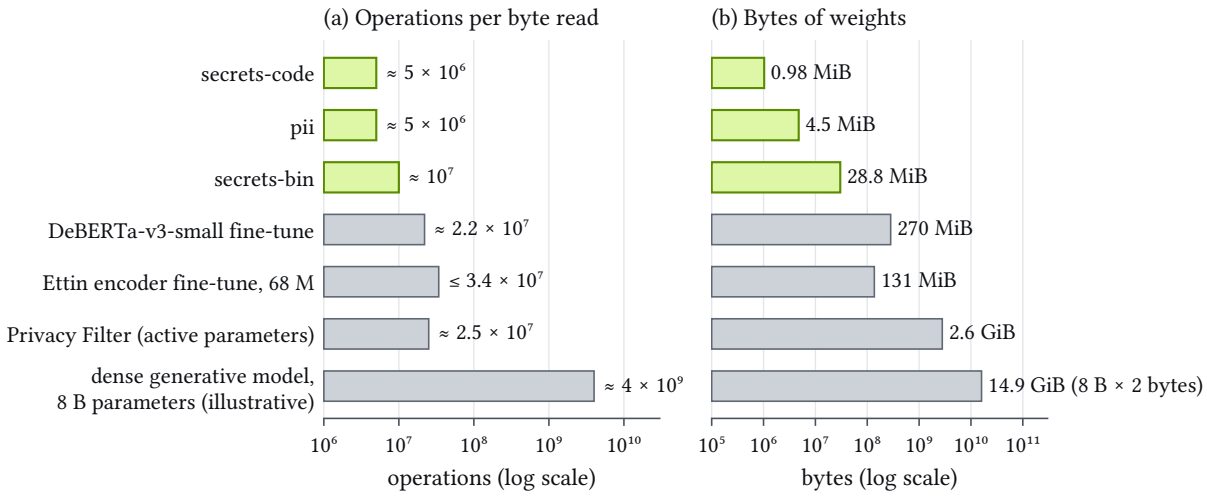


Figure 2: Table 2 on logarithmic axes. (a) Operations per byte read by the three released specialists, by two small pretrained encoders fine-tuned for personal data, by the Privacy Filter (its active parameters) and by an illustrative dense generative model of 8 B parameters before it generates any answer. (b) Bytes of weights, at 16 bits for the transformers. Estimates of work under the assumptions of the table, not measurements of time or energy.

Thesis. Making AI more efficient

Most automated decisions over data do not need a model that can talk. Answering them with a model the size of the job makes each decision with hundreds of times fewer operations and bytes of weights than a small generative model, and 4 to 7 times fewer operations and tens of times fewer bytes than a small pretrained encoder; it runs on the processor already in the machine, keeps the data there, and gives the same bits every time on a platform. Against rules it costs more work per byte and buys recall where the form is not fixed. The saving is multiplied by the number of decisions.

Limits of the claim

Table 2 estimates operations; the measured quantities of this paper are latencies, throughput and memory on one desktop CPU (Section 5), and we did not measure energy. The comparison with generative models is about cost, not quality: we did not evaluate generative models on the examples' benchmarks. The encoder and Privacy Filter rows use published parameter counts and an assumed four bytes per token, not measurements of those models.

4 The architecture

An AI Specialist is a function from a byte string to a small set of declared answers. This section defines that function: the input modules that turn bytes into vectors, the state-space block that processes them, its ternary weights, the optional attention block, the heads that produce decisions, scores, labels and byte spans, the policy that runs a model over inputs of any length, and the file that declares all of it. The normative definition is the model-format specification that ships with the runtime (`spec/FORMAT.md`, format version 3), which fixes the order of every floating-point operation; the equations below follow it, and a NumPy reference implementation of about 1,000 lines computes the same forward pass [179], within the tolerances of its tests rather than bit for bit, because it evaluates the C library functions in binary64 and rounds once. Section 5 describes the runtime that executes it and Section 6 how the weights are trained. The three specialists released with runtime 1.0.0, `secrets-code`, `pii` and `secrets-bin`, are the running example (Table 3).

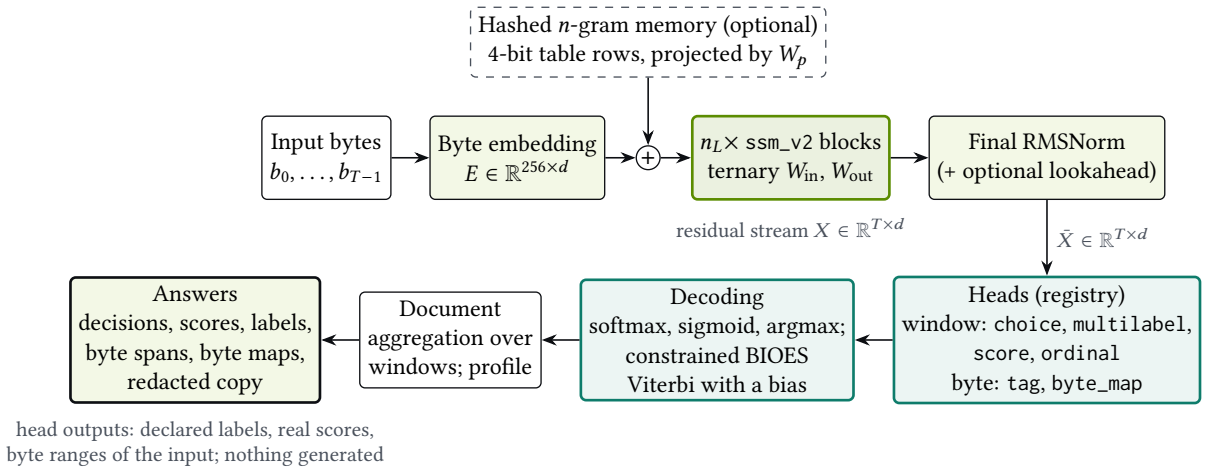


Figure 3: The architecture of an AI Specialist. Green: the blocks, whose large projections are ternary. Dashed: optional parts. Teal: heads and decoding. The released `pii` and `secrets-bin` use the n-gram memory; `secrets-code` does not. Every part is declared in the model file (Section 4.9).

Notation. A window is a byte string $b_0, \ldots, b_{T-1}$ with $b_t \in \{0, \ldots, 255\}$. The model width is d and the number of blocks n_L. The residual stream is $X \in \mathbb{R}^{T\times d}$ with rows X_t; after the final normalization the hidden states are $\bar{X} \in \mathbb{R}^{T\times d}$, with rows $\bar{X}_t$. For a vector v, $v[a, b)$ denotes its elements a to $b-1$. Values are IEEE 754 binary32 ("float") unless marked binary64 ("double") [180]. Sizes of files are given in bytes or in MiB (2^{20} bytes); MB and kB denote 10^6 and 10^3 bytes. Every matrix-vector product of the blocks and of the n-gram module, $y_o = \langle W_o, x\rangle$, is the canonical dot product of Equation (22); the heads use a sequential sum, $s \leftarrow \beta$, then $s \leftarrow s + w_i x_i$ for ascending i, each product rounded before its sum.

4.1 Overview

Figure 3 shows the pipeline. Each byte selects a row of a $256 \times d$ embedding, to which a model may add the output of a hashed n-gram memory, large lookup tables indexed by the bytes that end at each position. A stack of n_L blocks, each of a type from a registry (`ssm_v2`, a Mamba-2-style selective state-space block, or `attention`), transforms the residual stream. A final RMSNorm, optionally followed by a small anticausal convolution (the lookahead), gives the hidden states $\bar{X}$, which heads of types from a second registry turn into window-level decisions, labels, scores and levels, and byte-level spans and byte maps, the spans decoded by a Viterbi pass restricted to consistent label sequences. Inputs longer than a window are cut into windows of a declared size and stride, or read as one stream with carried state (Section 4.8); a post-processing profile, declared by the file or chosen by the caller, combines the windows into document-level results (Section 5.1).

Four properties follow. *Bytes in*: the alphabet is the 256 byte values, with no tokenizer, vocabulary, special symbol or input normalization, so a model reads any encoding, code, logs or machine code as stored and reports byte offsets. *Linear cost*: the released models use only causal `ssm_v2` blocks, whose work per byte does not depend on the window length (Proposition 4). *Closed outputs*: heads return only probabilities over declared classes, real scores, levels, labels from a declared set or byte ranges of the input, never generated text (Proposition 5). *A specialist is a file*: sizes, types, labels, operating points and the window policy are declared in one GGUF file (Section 4.9); the forward pass and the heads contain no task-specific code (merging rules, masking and redaction live in named profiles), so a model that finds credentials and one that labels file regions differ only in their files.

Table 3: The declared configuration of the three released specialists. Minimum: the shortest window evaluated (24 is the format's default; `pii` declares 1). Profile: the post-processing each model is paired with in the model catalog.

	`secrets-code`	`pii`	`secrets-bin`
Blocks n_L (all `ssm_v2`, causal)	4	4	8
Width d; block H, P, N, G, K; ternary group size	256; 8, 64, 64, 2, 4; 128		
n-gram memory (orders 2–4, 64 columns, 4-bit)	none	2^{15} rows per order	2^{18} rows per order
Window size / stride / minimum (bytes)	512 / 384 / 24	512 / 384 / 1	512 / 384 / 24
Head 0: `tag`, one entity type ($k = 1$)	`secret`	`PII`	`secret`
Head 1: `choice`, 2 classes, gates head 0	max pooling, MLP $256 \to 128 \to 2$		
Stored operating bias of head 0	0.0	−0.5	0.0
Profile (catalog)	`secrets-code`	`redact`	`secrets-binary`
File size (bytes)	1,028,704	4,764,864	30,163,488

4.2 Bytes and the byte embedding

The embedding is a float tensor E (`embed.weight`) of shape $[256, d]$ whose row b is the vector of byte value b. The forward pass starts with a row copy,

$$X_t = E[b_t], \qquad t = 0, \ldots, T-1, \tag{1}$$

with no scaling and no positional term: `ssm_v2` models have no position encoding, and order enters through the causal convolution and the scan of each block and, when present, the n-gram memory; the attention block uses rotary position embeddings. With $d = 256$ the embedding holds 65,536 parameters, stored as 262,144 bytes, about a quarter of the `secrets-code` file (Section 4.10). The alphabet is fixed, so the embedding does not grow with the task and no input is out of vocabulary.

4.3 The hashed n-gram memory

A model may add knowledge about short byte patterns through hashed tables of n-grams [46, 82, 181, 182], looked up with the bytes that end at each position and projected into the residual stream. The memory is present when the file contains tables; its orders are the orders of the tables present, each between 1 and 64.

Hashing. For the order n_i (orders indexed by $i = 0, 1, \ldots$ in ascending order) and the hash head h (`purebyte.ngram.heads`, 1 to 64, default 1), the n-gram that ends at position t selects row $r_{i,h}(t)$ of a table with n_r rows (buckets) through a polynomial hash modulo the Mersenne prime $p = 2^{31} - 1$:

$$\begin{aligned} &v \leftarrow (1000\, i + 7 + 7919\, h) \bmod p, \qquad v \leftarrow (257\, v + c_{t-j} + 1) \bmod p \quad \text{for } j = n_i - 1, \ldots, 0, \\ &r_{i,h}(t) = v \bmod n_r, \end{aligned} \tag{2}$$

where $c_\tau = b_\tau$ when position τ is readable and $c_\tau = 0$ otherwise, evaluated in 64-bit unsigned integers. An unreadable position therefore hashes like a zero byte, so an n-gram at the start of a window hashes like one that follows zero bytes, which are common in binaries. In a window, positions before its first byte are not readable, even when the input has bytes there, so a window is computed from its own bytes; in the stream context (Section 4.8) earlier bytes of the stream are readable. The selected rows are concatenated, order-major and then head, into $\phi_t \in \mathbb{R}^w$, where w is the number of orders times the number of heads times the width n_c of one table; the format admits at most 2^{24} rows per table and 65,536 concatenated columns.

4-bit rows. A table is stored as floats or, when `purebyte.ngram.bits` is 4, with 4 bits per value and one float scale per row. A 4-bit row holds $n_c/2$ bytes; byte j carries column $2j$ in its low nibble and column $2j+1$ in its high nibble, each a two's-complement integer $q \in \{-8, \dots, 7\}$, and a column's value is

$$M[r, j] = \sigma_r \cdot q_{r,j} \qquad \text{(one float product)}. \tag{3}$$

The exporter of the training stack quantizes each row symmetrically, $\sigma_r = \max_j |M[r, j]|/7$ and $q = \mathrm{clip}(\mathrm{rint}(M/\sigma_r), -7, 7)$ (a row of zeros gets $\sigma_r = 1$), so the released files use the codes -7 to 7. The runtime reads the tables in place from the file and decodes a row when it is looked up; a lookup reads one row per order and head, whatever the number of rows.

Projection, gate and injection. The concatenated rows are projected to the model width by a float matrix W_p (`ngram.proj.weight`, $[d, w]$), $e_t = W_p\, \phi_t$, and added to the residual stream before block ℓ_{ng} (`purebyte.ngram.layer`; the default 0 means right after the embedding):

$$X_t \leftarrow X_t + e_t. \tag{4}$$

With the optional context gate (`purebyte.ngram.gate`), the vector is first scaled by a scalar that compares it with the stream,

$$a_t = \mathrm{sigmoid}\Big(\frac{\langle X_t\, \rho(X_t),\ e_t\, \rho(e_t)\rangle}{\sqrt{d}} + \beta_{\mathrm{g}}\Big), \qquad X_t \leftarrow X_t + a_t\, e_t, \tag{5}$$

where ρ is the inverse RMS of Equation (14) and β_{g} the scalar `ngram.gate_bias`. Hash heads, a context gate of this form and a choice of injection layer are the design of Engram [81], here over bytes and without its key and value projections.

Released configuration. `pii` and `secrets-bin` use orders 2, 3 and 4, one hash head, $n_c = 64$ columns per table and 4-bit storage, with $n_r = 2^{15}$ rows per table in `pii` and $n_r = 2^{18}$ in `secrets-bin`; $w = 192$, W_p is 256×192, and the vector is added right after the embedding without a gate (the presets of their recipes set neither the gate nor another injection layer). Per position the memory costs three hash evaluations (nine multiply-and-modulo steps), three row reads of 32 bytes plus a scale, and 49,152 multiply-adds for the projection, under 10 % of one block (Proposition 4), whatever n_r. The tables hold 6,291,456 of the parameters of `pii` and 50,331,648 of those of `secrets-bin` (Section 4.10); they are trained jointly with the rest of the model from a random initialization (Section 6). A quarter of the table entries of `secrets-bin` can never be read: for the order index $i = 0$ and the single hash head $h = 0$, Equation (2) gives $v = 462{,}601 + 257\, c_{t-1} + c_t$, always below p, so the 65,536 bigrams reach 65,536 distinct rows of the 262,144 of the order-2 table, and the format requires every order's table to have the same number of rows. The other 196,608 rows of that table, 12,582,912 parameters, are stored but receive no gradient in training and are never looked up (Section 4.10); the order-2 table of `pii`, with 2^{15} rows, is fully used. Section 4.11 gives the measurements behind the memory and its 4-bit storage.

4.4 The `ssm_v2` block

The main block is a pre-norm residual block in the style of Mamba-2 [54, 55]: a selective state-space model whose step size and input and output projections of the state depend on the input, with a short depthwise causal convolution before the scan and a gated, grouped RMS normalization [183] after it. Its hyperparameters are read from `purebyte.block.<i>.<key>` when present, else from the model-wide `purebyte.<key>`: the number of heads H, the head width P, the state size N, the number of groups G of the state's input and output projections (H mod $G = 0$), the convolution width K (0 to 64; 0 for none) and the direction (`causal`, `bimamba` or `hydra`). They define the inner width $d_i = HP$, the $c_w = d_i + 2GN$ convolved channels and the $p_o = 2d_i + 2GN$ rows of the input projection; in the released models $d_i = 512 = 2d$, $c_w = 768$ and $p_o = 1{,}280$. Table 4 lists the tensors of a block.

Table 4: The tensors of an `ssm_v2` block (`blocks.<i>.*`) and their sizes in the released models. The two projections are ternary (Section 4.5); everything else is float.

Tensor	Shape	Released shape	Parameters
`norm.weight` (w_{n})	$[d]$	$[256]$	256
`in_proj.weight` (W_{in}, ternary)	$[p_o, d]$	$[1280, 256]$	327,680
`dt_proj.weight` (W_{Δ})	$[H, d]$	$[8, 256]$	2,048
`dt_bias`, `A_log`, `D`	$[H]$ each	$[8]$ each	24
`conv1d.weight`, `conv1d.bias`	$[c_w, 1, K]$, $[c_w]$	$[768, 1, 4]$, $[768]$	3,840
`gated_norm.weight` (w_{gn})	$[d_i]$	$[512]$	512
`out_proj.weight` (W_{out}, ternary)	$[d, d_i]$	$[256, 512]$	131,072
Block, of which 458,752 ternary			465,432

Computation. Over a window of T positions the block computes, each step for every position before the next (Figure 4):

$$\hat{x}_t = \mathrm{norm}(X_t,\ w_{\mathrm{n}}), \tag{6}$$

$$p_t = W_{\mathrm{in}}\,\hat{x}_t \in \mathbb{R}^{p_o}, \qquad z_t = p_t[0, d_i), \qquad \xi_t = p_t[d_i,\ d_i + c_w), \tag{7}$$

$$\Delta_{t,h} = \mathrm{softplus}\left(b_{\Delta,h} + \langle W_{\Delta,h}, \hat{x}_t\rangle\right), \qquad \bar{a}_{t,h} = \exp(a_h\,\Delta_{t,h}), \qquad a_h = -\exp(A_{\log,h}), \tag{8}$$

$$u_{t,c} = \mathrm{SiLU}\left(\beta_c + \sum_{k=0}^{K-1} \kappa_{c,k}\,\xi_{t-K+1+k,\,c}\right), \qquad c = 0, \ldots, c_w - 1, \tag{9}$$

$$s_t^{(c)} = \bar{a}_{t,h}\,s_{t-1}^{(c)} + \left(\Delta_{t,h}\,x_{t,c}\right) B_{t,g}, \qquad y_{t,c} = \left\langle s_t^{(c)},\ C_{t,g}\right\rangle + D_h\,x_{t,c}, \qquad c = 0, \ldots, d_i - 1, \tag{10}$$

$$\tilde{y}_{t,c} = y_{t,c} \cdot \mathrm{SiLU}(z_{t,c}), \tag{11}$$

$$\bar{y}_t^{[j]} = \mathrm{norm}\left(\tilde{y}_t^{[j]},\ w_{\mathrm{gn}}^{[j]}\right), \qquad j = 0, \ldots, G - 1, \tag{12}$$

$$X_t \leftarrow X_t + W_{\mathrm{out}}\,\bar{y}_t, \tag{13}$$

where every normalization is the RMSNorm of the numerics contract,

$$\begin{aligned} \rho(x) &= \mathrm{fl}\Big(1/\mathrm{sqrtf}\left(\mathrm{fl}\left(\mathrm{fl}(q/n) + \mathrm{fl}(10^{-5})\right)\right)\Big), \qquad q = \sum_{i=0}^{n-1} x_i^2 \ \text{(binary64, ascending } i\text{)}, \\ \mathrm{norm}(x, w)_i &= (x_i\,\rho(x))\,w_i, \end{aligned} \tag{14}$$

where fl rounds to float: q/n is computed in binary64 and rounded, 10^{-5} is the nearest float, and the sum, the correctly rounded `sqrtf` and the reciprocal are float operations; the last product has two float roundings. The ternary projection gives the gate z_t and the convolution input ξ_t; the step size comes from its own float matrix W_{Δ} applied to the normalized input, with $\mathrm{softplus}(v) = v$ for $v > 20$ and $\log(1 + e^v)$ otherwise (the C library's `log1pf` and `expf`), and a_h is computed once, at load. The depthwise causal convolution runs over the c_w channels of ξ (not over z), accumulated with fused multiply-adds for ascending k from the bias β_c; taps before the first readable row (in a window, row 0) are skipped. The SiLU of the blocks, $x/(1 + \exp(-x))$, uses the polynomial exponential of the numerics contract (Section 5.2). The convolution output splits into $x_t = u_t[0, d_i)$ and, for each group g, $B_{t,g} = u_t[d_i + gN,\ d_i + (g+1)N)$ and $C_{t,g} = u_t[d_i + GN + gN,\ d_i + GN + (g+1)N)$. In the scan, channel c belongs to head $h = \lfloor c/P \rfloor$, which reads group $g = \lfloor h/(H/G) \rfloor$, and its state $s^{(c)} \in \mathbb{R}^N$ starts at zero at the beginning of each window; stacking the P channels of a head gives the matrix form of Mamba-2, $S_{t,h} = \bar{a}_{t,h} S_{t-1,h} + \Delta_{t,h}\,x_{t,h} \otimes B_{t,g}$ and $y_{t,h} = S_{t,h} C_{t,g} + D_h x_{t,h}$, with $S_{t,h} \in \mathbb{R}^{P \times N}$ and one scalar decay rate a_h per head. Algorithm 1 gives its order of operations. The output is gated by $\mathrm{SiLU}(z)$, normalized separately in each of the G groups of d_i/G consecutive channels, and projected back by the ternary W_{out} into the stream.

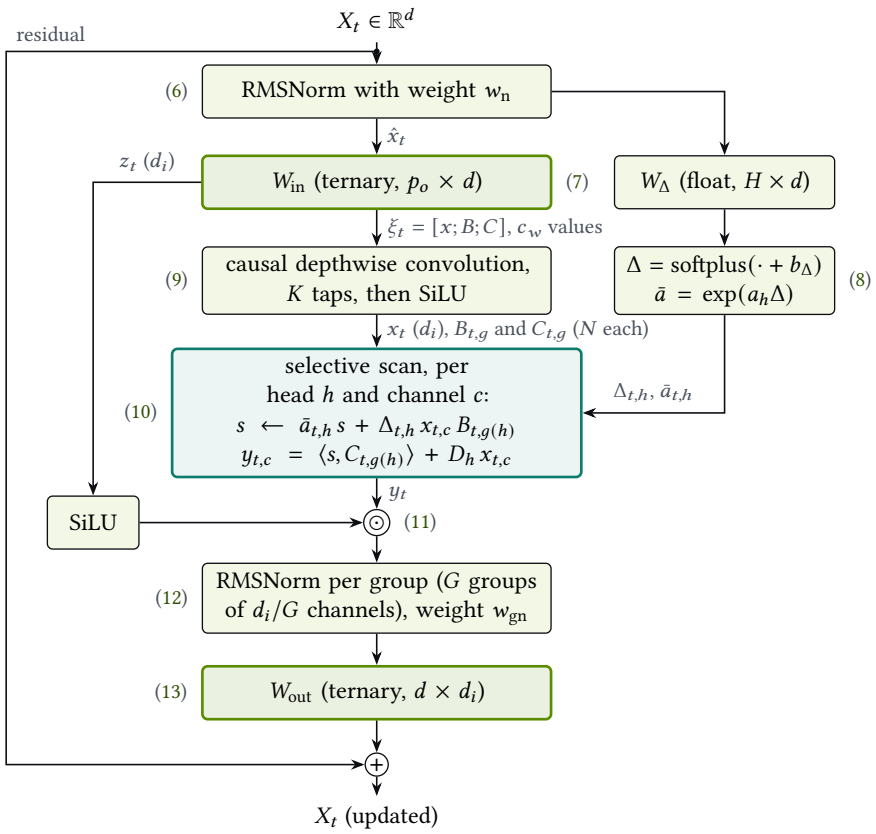


Figure 4: Dataflow of one `ssm_v2` block at position t, with the equation of each step. Green: ternary projections. Teal: the selective scan, the only step that carries state from one position to the next. In the released models $d = 256$, $d_i = 512$, $p_o = 1{,}280$, $c_w = 768$, $H = 8$, $P = N = 64$, $G = 2$ and $K = 4$.

Algorithm 1 The scan of one channel c of head h (group g) over a window, in the order the runtime executes it. $N_8 = N - (N \bmod 8)$; fma is the correctly rounded fused multiply-add (Section 5.2); every other operation is one rounded float operation.

$s[0..N-1] \leftarrow 0$ ▷ fresh state at the start of a window
for $t = 0, \ldots, T-1$ **do**
  $\delta \leftarrow \Delta_{t,h} \cdot x_{t,c}$; $\ell[0..7] \leftarrow 0$
  for $n = 0, \ldots, N_8 - 1$ **do**
    $s[n] \leftarrow \text{fma}(\delta,\ B_{t,g}[n],\ \bar{a}_{t,h} \cdot s[n])$ ▷ the product $\bar{a}\, s$ rounded first
    $\ell[n \bmod 8] \leftarrow \text{fma}(s[n],\ C_{t,g}[n],\ \ell[n \bmod 8])$
  $o \leftarrow ((\ell_0 + \ell_4) + (\ell_1 + \ell_5)) + ((\ell_2 + \ell_6) + (\ell_3 + \ell_7))$
  for $n = N_8, \ldots, N-1$ **do** ▷ each product rounded before its sum
    $s[n] \leftarrow \bar{a}_{t,h} \cdot s[n] + \delta \cdot B_{t,g}[n]$; $o \leftarrow o + s[n] \cdot C_{t,g}[n]$
  $y_{t,c} \leftarrow o + D_h \cdot x_{t,c}$ ▷ a separate product and sum, never fused

State and directions. The only values a causal block carries from one position to the next are the HPN values of the scan state and the last $K - 1$ rows of ξ: 35,072 floats ($32{,}768 + 3 \times 768$), 140,288 bytes, for the released block. This is all a causal block needs to continue a sequence, and what the stream context carries (Section 4.8). The directions `bimamba` and `hydra` are weight-tied variants of the bidirectional blocks of Vision Mamba [184] and Hydra [185]: the reversed pass reuses the forward pass's convolution, B, C and Δ, whereas Vision Mamba gives the backward direction its own convolution, projections, A and D, and Hydra its own B, C and Δ; the format has no tensors for a separate backward direction. Both convolve and scan the reversed sequence, giving y^{b}, whose row r belongs to position

$T-1-r$; `bimamba` adds it to the forward result, $y_t = y^{\mathrm{f}}_t + y^{\mathrm{b}}_{T-1-t}$, both with the D term, and `hydra` adds shifted scans computed without D and then D once, $y_{t,c} = (y^{\mathrm{f}}_{t-1,c} + y^{\mathrm{b}}_{T-2-t,c}) + D_h x_{t,c}$, out-of-range terms being 0. All released blocks are causal (Section 4.11).

The scan in training. The runtime and the NumPy reference compute Equation (10) position by position. Training computes the same function in the chunked form of the state-space duality of Mamba-2 [55]: within chunks of $Q = 64$ positions a masked matrix product whose accumulated decay exponents are clamped at -30, and across chunks a recurrence over chunk states; padding positions get $\Delta = 0$, an identity step. The two forms agree up to the clamp ($e^{-30} \approx 10^{-13}$) and rounding, not bit for bit; each export is checked by running the file in the runtime and requiring the same spans as the trained model on test windows (Section 6). At initialization $|a_h|$ and Δ_h are log-spaced over $[1, 16]$ and $[10^{-3}, 10^{-1}]$ and paired in ascending order, so the initial half-lives $\ln 2/(|a_h|\Delta_h)$ of the eight heads of a released block span 0.43 to 693 bytes. This departs from Mamba-2, which draws $|a_h|$ uniformly from $[1, 16]$ and Δ_h log-uniformly, independently: with eight heads, such a draw gave a longest initial half-life of 38 bytes, every head local. We did not compare the two initializations on a task.

Proposition 4 (Work per byte). *In a window of T positions, a causal* `ssm_v2` *block performs, per position,*

$$m = d\,p_o + d_i\,d + 2\,d_i\,N + c_w K + Hd \tag{15}$$

multiply-adds in its two projections, its scan, its convolution and its step sizes, independently of T, plus $d_i N$ multiplications of the decay in the scan and $O(p_o)$ other operations. For the released block $m = 327{,}680 + 131{,}072 + 65{,}536 + 3{,}072 + 2{,}048 = 529{,}408$: about 271 M multiply-adds per block for a window of 512 bytes, and 2.12 M per window position for four blocks (2.82 M per input byte with the released windows of 512 bytes every 384).

Proof. Equations (6) to (8) and (11) to (13) read only position t and Equation (9) reads K rows. Counting one multiply-add per entry of W_{in}, W_{out}, W_Δ and κ, and two per state value and position in the scan (the update's fused multiply-add and the read-out), gives m; the update's product $\bar{a}\,s$ is the separate multiplication counted apart (Algorithm 1). No term involves T. □

The count includes the zeros of the ternary matrices, which the runtime multiplies (Section 5.3). An attention block performs $O(t\,d)$ multiply-adds at position t, so its work per window grows with T^2.

4.5 Ternary weights

The two large projections of every `ssm_v2` block, W_{in} and W_{out}, are ternary: each weight is -1, 0 or $+1$ times a float scale shared by a group of g consecutive inputs of a row (`purebyte.tern.group`; $g = 128$ in the released models). They hold 458,752 of the 465,432 parameters of a released block (98.6 %); the projections of an attention block may also be ternary. Everything else is float: the embedding, W_Δ, the convolution, the normalizations, a, b_Δ, D, the n-gram projection and the heads (the n-gram tables have their own 4-bit format).

Storage (`pb_tern2`). A ternary projection of shape $[out, in]$, where in is a multiple of g, is stored as two tensors: `<name>.packed`, bytes of shape $[out, \lceil in/4 \rceil]$, and `<name>.scales`, floats of shape $[out, in/g]$. Code j of row o occupies bits $2(j \bmod 4)$ and $2(j \bmod 4) + 1$ of byte $\lfloor j/4 \rfloor$ of the row: `00` is 0, `01` is $+1$, `10` is -1; `11` is invalid and the padding codes of a row must be `00`, and a file that breaks either rule is refused. The weight is

$$W_{o,j} = q_{o,j} \cdot s_{o,\lfloor j/g \rfloor}, \qquad q_{o,j} \in \{-1, 0, +1\}, \tag{16}$$

a product that is exact in binary32. With one 32-bit scale per group, a weight takes $2 + 32/g$ bits: 2.25 for $g = 128$. The runtime unpacks the codes once, at load, to one signed byte per weight, and its kernels expand a few rows at a time to floats (exact) before the canonical dot product (Section 5.3). There is no

integer arithmetic and no quantization of activations: every activation of a specialist is a float. What the ternary constraint buys in this release is size: the file of `secrets-code` is 7.6 times smaller than in float32 (Section 4.10), and its weights in memory, the file plus one byte per ternary weight, about 2.7 times smaller (Section 5.5). It does not reduce the operations of Proposition 4: the kernels multiply every weight, zeros included. Kernels that exploit ternary weights, as bitnet.cpp does [74], are future work.

Training, at a high level. The ternary constraint is present from the first training step. Each ternary matrix keeps a float latent weight, and the forward pass uses

$$s_{o,j} = \max\left(10^{-8},\ \frac{1}{g}\sum_{i\in\mathcal{G}_j}|W_{o,i}|\right), \qquad \mathrm{Tern}(W)_{o,i} = s_{o,j}\cdot \mathrm{clamp}\left(\mathrm{round}(W_{o,i}/s_{o,j}), -1, 1\right), \tag{17}$$

where $\mathcal{G}_j$ is the j-th group of g inputs of row o: an absmean scale as in BitNet b1.58 [71], applied per group. A weight becomes 0 when $|W_{o,i}| < s_{o,j}/2$, which zeroes about 31 % of Gaussian weights. The scale is recomputed at every step and receives no gradient; the backward pass is the identity straight-through estimator [69]. There is no float warm-up and no conversion after training: the exported codes and scales are those of Equation (17), so the runtime's weights are, value for value, the ternary weights of the trained model's forward pass (Section 6).

4.6 The optional attention block

The block registry also holds `attention`, causal softmax self-attention [186] with rotary position embeddings [187]. With head width h_d (`purebyte.block.<i>.head_dim`, even, default 64) and d/h_d heads, the block normalizes the stream, projects it with W_{qkv} ($[3d, d]$: rows for q, then k, then v), rotates each head of q and k in rotate-half pairs $(j,\ j + h_d/2)$ by the angle $t\,f_j$, with $f_j = \exp(j\cdot \mathrm{fl}(-\ln 10{,}000/(h_d/2)))$, and computes per head

$$e_i = \langle k_i, q_t\rangle \cdot \frac{1}{\sqrt{h_d}}, \qquad w_i = \exp\left(e_i - \max_{i'\le t} e_{i'}\right), \qquad y_t = \sum_{i=0}^{t}\frac{w_i}{\sum_{i'\le t} w_{i'}}\,v_i, \qquad X_t \leftarrow X_t + W_o\,y_t, \tag{18}$$

with sequential sums in ascending order. It needs the whole window, so a model with an attention block cannot use the stream context. No released model has one (Section 4.11).

4.7 Heads, gating and decoding

A model declares up to 64 heads (`purebyte.head.count`), numbered from 0, each of a type from the head registry, with optional names for its classes, labels or entity types. The seven types are `choice`, `multilabel`, `score` and `ordinal`, which answer once per window, `tag` and `byte_map`, which label every byte, and `exit`, a probe that answers nothing; Figure 5 shows how they read the hidden states.

Window-level heads. `choice`, `multilabel`, `score` and `ordinal` pool the T rows of $\bar{X}$ into one vector v (`purebyte.head.<i>.pooling`): `last` takes $\bar{X}_{T-1}$, `mean` accumulates $\bar{X}_{t,j}/T$ for ascending t, `max` takes $\max_t \bar{X}_{t,j}$ with the C function `fmax` from -10^{30}, and `last_mean` (the default) and `max_mean` concatenate two of them. The first three then apply a two-layer perceptron whose widths come from its tensors,

$$out = W_2\ \mathrm{SiLU}(W_1 v + b_1) + b_2, \tag{19}$$

with sequential sums and the C library's `expf`. A `choice` head returns $p = \mathrm{softmax}(out/\tau)$ (temperature $\tau > 0$, default 1; the maximum subtracted first) and the first index of the largest probability; class 0 is the negative class, the head is positive when its label is not 0, and the window's p_{positive} is $1 - p_0$. A `multilabel` head returns $p_k = \mathrm{sigmoid}(out_k/\tau)$, label k being on when p_k reaches its threshold (default 0.5), and is positive when a label is on. A `score` head returns $out_k \cdot scale_k + offset_k$. An `ordinal` head, in the style of CORAL [188], computes $z = \langle w, v\rangle$ without a perceptron and returns $p_k = \mathrm{sigmoid}(z + b_k)$ for $k < L - 1$, with the thresholds sorted in decreasing order at load, and the level $\#\{k : p_k > 0.5\}$; it is positive when the level is above 0.

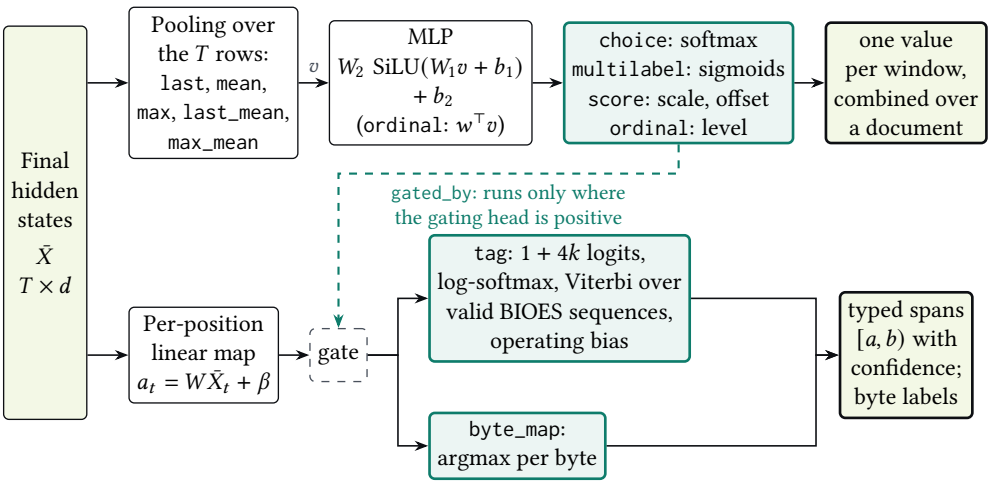


Figure 5: Window-level heads pool the T hidden states and answer once per window; byte-level heads read every position. A head with `gated_by` runs only in windows where its gating head is positive. In the released models head 1 (`choice`, max pooling, MLP $256 \to 128 \to 2$) gates head 0 (`tag`, $k = 1$). The optional `exit` probe, which reads the residual stream after ℓ blocks, is not drawn; no released model has one.

The `tag` head. A `tag` head labels every byte with the BIOES scheme for k entity types [189]: label 0 is O (outside), and entity e has B $= 1 + 4e$ (begin), I $= 2 + 4e$ (inside), E $= 3 + 4e$ (end) and S $= 4 + 4e$ (single byte), $L = 1 + 4k$ labels in all. A linear map with a sequential sum gives $\alpha_{t,l} = \beta_l + \sum_i W_{l,i}\bar{X}_{t,i}$ and a log-softmax gives $\lambda_{t,l} = \alpha_{t,l} - (\mu_t + \log \sum_{l'} e^{\alpha_{t,l'} - \mu_t})$ with $\mu_t = \max_l \alpha_{t,l}$. The labels are decoded, as in OpenAI's Privacy Filter [160], by a Viterbi pass [190] in binary64 over the consistent sequences only,

$$\hat{y} = \arg\max_{y \in \mathcal{Y}_T} \sum_{t=0}^{T-1} \Big(\lambda_{t,y_t} + \gamma_{e(y_t)} \, [\, y_t \neq \mathrm{O} \,] \Big), \tag{20}$$

where $\mathcal{Y}_T$ holds the sequences that start with O, B or S, end with O, E or S, and in which O, E_x and S_x are followed only by O, B_y or S_y, and B_x and I_x only by I_x or E_x of the same entity; among equal scores the lowest label index wins, for predecessors and for the final label. γ_e is the operating bias of entity type e, added to its four labels and never to O, once per byte: a positive bias yields more and longer spans (recall), a negative one fewer and shorter (precision). It is the caller's per-type vector, else the caller's single value, else the file's `operating_bias_vec`, else its `operating_bias`; the released files store 0.0, −0.5 and 0.0. Walking $\hat{y}$, B opens a span, E of the same entity closes it (inclusive), S is a one-byte span and O discards an open span. A span's confidence is the mean over its bytes of $\exp(\lambda_{t,\hat{y}_t})$, without the bias, in binary64 rounded to float; it ranks spans and is not calibrated.

`byte_map` and `exit`. A `byte_map` head labels every byte with the first index of the largest of C logits ($2 \le C \le 256$); over a document, a byte takes the label of the earliest window that covers it, the one with the most left context. An `exit` head is an early-exit probe after the first ℓ blocks: with $v_t = X_t \, \rho(X_t)$ it forms $f_j = \max_t v_{t,j}$ and $f_{d+j} = \sum_t v_{t,j}/T$ (in binary64) and a score $\beta + \sum_j w_j f_j$ in binary64. When the caller enables early exit, an approximation that is off by default, a window whose score is below the declared threshold stops after those ℓ blocks and counts as negative.

Gating and documents. A head with `gated_by` $= u$ runs only in windows where head u is positive; u must be an ungated `choice`, `multilabel` or `ordinal` head other than the head itself. Ungated heads run first, in index order, then the gated ones. Gating is part of the declared model, not an approximation: it decides which heads run, not what they compute, and it saves only the gated heads' work, since the body runs in every window; a caller may run gated heads regardless (`PB_SCAN_UNGATED`). In the released models the `tag` head is gated by the `choice` head, so spans are produced only in windows that the choice

head classifies as positive; the tag head is trained on every window and used only where the gate is positive. The gate's max pooling suits the question it answers, whether anything in the window is a span (Section 4.11). Over a document, the declared aggregate of a window-level head combines the windows in which it was computed: max (the default of choice, multilabel and ordinal) keeps the highest value, $1 - p_0$ for choice; mean (the default of score) averages in binary64; choice also accepts vote and any, multilabel any, and score min; ties go to the earliest window or the lowest index. The spans of tag heads are collected in input coordinates and combined by the profile (Section 5.1).

Proposition 5 (Closed outputs)**.** *For any input of n bytes and any model file the runtime accepts, as long as no value overflows and the decoder's path scores stay between -10^{29} and 10^{29} before its penalties, every value a head reports for a window is one of the following: a vector of probabilities over the classes, labels or levels the head declares, possibly with an index into them; a vector of real scores of the declared length; a set of byte ranges $[a, b)$ with $0 \le a < b \le n$, each with an entity type declared by the head and a confidence in $[0, 1]$; one label per byte from a declared set of at most 256 classes. An* exit *head reports nothing.*

Proof. By the definitions above. The runtime's reader refuses a float tensor that holds a NaN or an infinity, so the weights are finite. Softmax and sigmoid outputs are probabilities, and score outputs are affine maps of the perceptron's output. The decoder marks a forbidden transition with a penalty of -10^{30}, so under the second condition every path that contains one scores below every consistent path; decoded tag labels therefore form consistent sequences over the window, whose spans lie between positions of the window and are shifted by the window's offset into input coordinates (spans that end inside a query prefix are dropped, and those that start inside it are cut); a confidence is a mean of values $\exp(\lambda)$ with $\lambda \le 0$. byte_map labels are indices of logits. Class and entity names are strings stored in the file. □

The profiles transform these values (the redact profile, for example, replaces each resolved span by a typed marker and checks that every other byte is copied unchanged), but the network cannot produce a value of another form.

4.8 Windows, lookahead and the stream context

Windows. In the window context (the default), an input of n bytes is cut into windows of window.size bytes (keys purebyte.window.*; default 512) starting at $0, s, 2s, \ldots$, where the stride s is window.stride (1 to the size; default the size minus a quarter of it, 384). Each window has $\min(\mathit{size}, n - \mathit{start})$ bytes, and the walk stops after the first window that reaches the end of the input (Figure 6). When the next window would be shorter than window.min (default 24), it is not evaluated and one last window of window.size bytes aligned to the end of the input is evaluated instead, so that every byte is in some window; the released models' size and stride never need it. An input shorter than window.min has no window (redaction evaluates it all the same). Each window is an independent forward pass: nothing is carried between windows and the n-gram hashes read nothing before a window's first byte, so a window's result depends only on its bytes, and windows can run in any order, in parallel. Callers may override the size and the stride, and then results differ from the declared behavior, or run an input as one window of up to 1 GiB (PB_SCAN_WHOLE).

For the released policy an input of $n > 512$ bytes gives $\lceil (n-512)/384 \rceil + 1$ windows (2,731 for 1 MiB); consecutive windows overlap by 128 bytes, so a long input is read about 4/3 times, and every byte at position p lies in at least one window in which at least $\min(p, 128)$ bytes precede it. For each window the runtime reports every head's outputs, the label and p_{positive} of the first choice head, whether the window was gated, stopped by the exit head or skipped by an optional prefilter, and on request a 64-bit FNV-1a digest of its final hidden states and head values, which fingerprints its numerics. No window owns a part of the input: overlapping windows can report the same span, and the profiles merge spans in input coordinates.

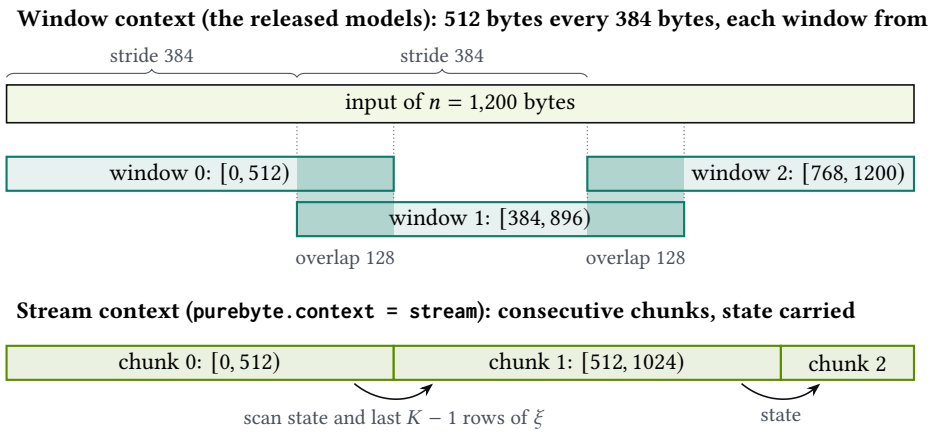


Figure 6: Top: the window context of the released models on an input of 1,200 bytes; consecutive windows overlap by 512 − 384 = 128 bytes and the last window has 432 bytes. Bottom: the stream context, in which the input is one sequence and each block's state is carried from chunk to chunk.

Query template and lookahead. A model may declare a query region (`purebyte.query.*`): each window then starts with up to `query.max` questions, each written as a prefix, the question and a suffix, padded to the region and followed by *size* − *region* document bytes, and spans and byte labels are reported for document bytes only; this extracts a value named at run time. A model may also declare a lookahead, a tensor ω (`look.conv_w`, $[d, k_{\text{la}}]$, $1 \le k_{\text{la}} \le 1024$) that gives every head the k_{la} bytes to the right of each position,

$$\bar{X}'_{t,c} = \bar{X}_{t,c} + \sum_{j=1}^{k_{\text{la}}} \omega_{c,j-1}\, \bar{X}_{t+j,c} \qquad (\bar{X}_{t+j} = 0 \text{ past the window's end}), \tag{21}$$

accumulated for ascending j with each product rounded before its sum. The released models use neither.

Stream context. With `purebyte.context =stream`, each input is one sequence. The runtime runs it in consecutive chunks of `window.size` bytes and carries each block's state (Section 4.4) from chunk to chunk, so that the hidden states are exactly those of one forward pass over the whole input; the n-gram hashes read the bytes before a chunk. The heads run on every chunk, reported as a window: window-level heads pool over the chunk, and spans never cross a chunk boundary. A stream model may not have attention blocks, non-causal `ssm_v2` blocks, a lookahead, a query template or an exit head, and a caller cannot ask for a stride, a prefilter or early exit. The carried state does not grow with the input: 140,288 bytes per block of the released shape. The released models are trained on windows and declare the window context; a research model of the code task, run as a stream without retraining, flagged thousands of new positions (Section 4.11). Models trained for the stream context are future work.

4.9 The model file

Container. A model is one GGUF file, version 3 [191], little-endian: the magic number `0x46554747`, the version, the counts of tensors and of metadata entries (each at most 2^{20}), typed key–value metadata, the tensor descriptions (name, rank up to 4, dimensions, type, offset), padding to the alignment (`general.alignment`, a power of two from 8 to 65,536, default 32) and the tensor data. Two tensor types are used: F32 for every number and I8 for packed data (ternary codes and 4-bit tables), read as raw bytes. Shapes are written row-major, while GGUF stores dimensions fastest first. A key or tensor name may not appear twice, every tensor must lie inside the file and no two tensors may share a byte; the reader treats the file as untrusted and checks every length, count, offset and dimension before using it.

Table 5: The metadata keys of a model file, with the prefix `purebyte.` unless written in full. The values of the released files are in Table 3; they leave `ngram.layer` and `ngram.gate` at their defaults (0, no gate) and declare no query template and no lookahead.

Part	Keys
Model	`general.architecture` (`purebyte`), `format_version`, `required_features`, `d_model`, `n_layers`, `context`, `profile`; informative: `general.name`, `general.alignment`, `general.version`, `general.license`, `general.license.name`
Ternary weights	`tern.type` (`pb_tern2`), `tern.encoding` (`00=0,01=+1,10=-1`), `tern.group`
Blocks	`block.<i>.type`; `n_heads`, `d_head`, `d_state`, `n_groups`, `conv_width`, `direction`, model-wide or as `block.<i>.<key>`; `block.<i>.head_dim` and `.n_heads` (attention)
n-gram memory	`ngram.orders`, `.heads`, `.bits`, `.hash`, `.buckets`, `.dim`, `.layer`, `.gate`
Heads	`head.count`; `head.<i>.type`, `.name`, `.labels`, `.gated_by`, `.aggregate`, `.pooling`, `.temperature`, `.thresholds`, `.scale`, `.offset`, `.n_entities`, `.n_labels`, `.scheme`, `.operating_bias`, `.operating_bias_vec`, `.features`, `.layer`, `.threshold`
Windows, queries, lookahead	`window.size`, `.stride`, `.min`; `query.region`, `.max`, `.prefix`, `.suffix`, `.pad`; `lookahead`

Declaration. Table 5 lists the metadata keys. They declare everything that determines what the model computes: the sizes, the ordered blocks with their hyperparameters, the input modules, the heads with their labels, pooling, gating, aggregation and operating points, the context and window policy, and the profile the model is meant for. The runtime assembles the model from registries of block types, head types and profiles (Section 5.1); the file carries no code.

Versions and refusals. `purebyte.format_version` is required: a reader of version v reads versions 2 to v, refuses a newer file as unsupported and a file without a version, or below 2, as malformed. The released files use keys that version 3 introduced (window policy, pooling, gating). A file may also list, in `purebyte.required_features`, the optional capabilities it needs: `attention`, `lookahead`, `ngram_gate`, `ngram_layer`, `ngram_heads`, `multilabel`, `score`, `ordinal`, `byte_map`, `query` and `stream`; a reader refuses a file that requires one it does not know. Unknown keys are ignored, so a key that changes what a model computes must come with a new version or a required feature, and every tensor must be used: an unused tensor means that the file computes something the reader would not. A reader also refuses, naming the problem, truncated or malformed files, unknown types or encodings, missing or misshapen tensors, values out of range, contradictory declarations (label counts, dimensions, gating targets, a stride larger than the window, a stream context with a part that cannot stream) and invalid ternary codes. The specification's rule is that a reader must not guess; the test suite checks it on 34 malformed or contradictory files. The container has no checksum: each released file is identified by its size and SHA-256 in the runtime's model catalog, and the command-line tool refuses to load an installed catalog model whose file does not match its published hash.

4.10 Parameters and bytes of the released specialists

Table 6 counts the parameters of the released files from their tensor shapes, and Figure 7 draws them. The network is the embedding, the blocks and the final normalization; the *n*-gram memory is the tables and their projection; the heads are a `choice` perceptron of 33,154 parameters ($256 \cdot 128 + 128 + 128 \cdot 2 + 2$) and a `tag` projection of 1,285 ($5 \cdot 256 + 5$). The per-group scales of the ternary weights and the per-row scales of the 4-bit tables are derived from the weights and not counted as parameters; the table gives their number. Earlier documents of the project quote the totals without the heads: 1,927,520, 8,268,128 and 54,170,048; so do the release's model cards, which count the *n*-gram tables as part of the network, and the rounded figures of this paper (1.9, 8.3 and 54.2 million).

The same shapes and the storage formats of Sections 4.3 and 4.5 reconstruct the files: 2 bits per ternary weight plus a float scale per group, 4 bits per table value plus a float scale per row, and 4 bytes per float parameter give 1,023,900, 4,759,452 and 30,155,036 bytes of tensor data, all but 4,804, 5,412 and

Table 6: Parameters of the released specialists, counted from the tensor shapes of their files. Never read: the rows of the order-2 table of `secrets-bin` that no bigram reaches (Section 4.3).

	`secrets-code`	`pii`	`secrets-bin`
Byte embedding (256 × 256)	65,536	65,536	65,536
Blocks (465,432 each)	1,861,728	1,861,728	3,723,456
of which ternary	1,835,008	1,835,008	3,670,016
Final normalization	256	256	256
Network	1,927,520	1,927,520	3,789,248
n-gram tables ($3 \times n_r \times 64$, 4-bit)	—	6,291,456	50,331,648
of which never read	—	0	12,582,912
n-gram projection (256 × 192)	—	49,152	49,152
Heads (`choice` and `tag`)	34,439	34,439	34,439
Total	1,961,959	8,302,567	54,204,487
Not counted: ternary group scales	14,336	14,336	28,672
Not counted: 4-bit row scales	—	98,304	786,432

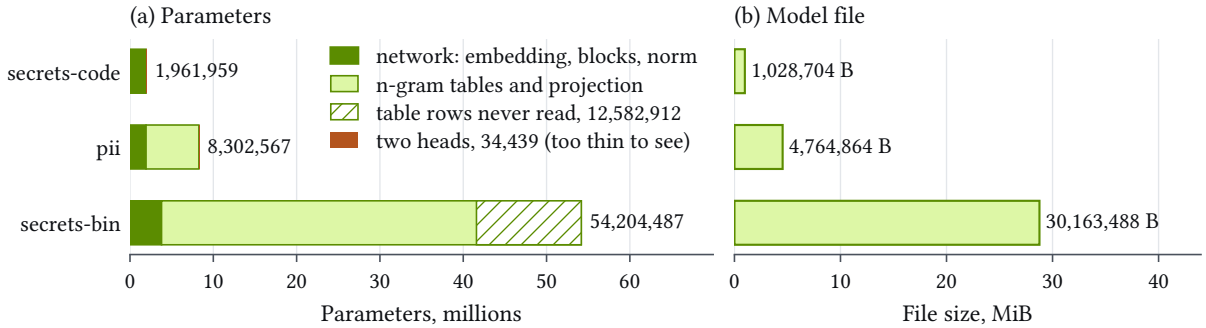


Figure 7: Where the parameters and the bytes of the released specialists are. (a) Parameters by component (Table 6): `secrets-code` and `pii` have the same network, and most of `pii` and `secrets-bin` is n-gram tables, which cost three row reads per byte whatever their size; hatched: the table parameters of `secrets-bin` that are never read. (b) File sizes in MiB, with the exact number of bytes.

8,452 bytes of the files, which hold the header, the metadata, the tensor descriptions and the alignment padding. In `secrets-code` the ternary projections take 50 % of the file (458,752 bytes of codes and 57,344 of scales), the float embedding 25 % (262,144 bytes) and the float heads 13 % (137,756 bytes); in `secrets-bin` the n-gram memory takes 94.5 % (25,165,824 bytes of 4-bit values, 3,145,728 of row scales and 196,608 of projection). The rows of `secrets-bin` that are never read (Section 4.3) hold 12,582,912 parameters, 23 % of the model, and take 7,077,888 bytes, 23 % of the file; without them the model has 41,621,575 addressable parameters (41.6 million against 54.2). A format version that allowed a row count per order would remove them: an order-2 table of 65,791 rows, the span of its hash values, would hold every bigram without collision in a file of about 22.0 MiB. As 32-bit floats, the 1,961,959 parameters of `secrets-code` would take 7,847,836 bytes, 7.6 times its file. The memory these files need at run time is a different account (Section 5.5).

4.11 Design choices and their evidence

The configuration of the released specialists survived ablations run with the research code that trained them. The box lists each choice, the alternatives measured and the results, with the conditions the sources give; the other sections cite it for these numbers. FA: false alarms, findings on inputs without credentials; bpb: bits per byte of held-out data in a language-modeling comparison. Unless noted, a

comparison uses three seeds and the same data, and false alarms are heavy-tailed: about one binary run in six ended with 50 to 230 FA. We therefore give the value of each seed and state each difference in standard errors of the difference of the means (Welch's statistic; for FA, on the logarithm of one plus the count), and we call a difference under two standard errors *not resolved*. Two standard errors are weak evidence with three runs against three: the Welch p-values of the resolved differences below range from 0.02 to 0.11, and no rank test on three runs against three can give a two-sided p below 0.1. The exams are those of the case studies (`bin-fp-a` and `bin-fp-b`, clean binaries; `code-fp-b`, clean code repositories; the PII Masking Benchmark), which also report the released models (Table 19): the architecture was chosen on them, so the case-study figures on the same sets are not independent of these choices, and an unbiased estimate needs an exam that played no part in them.

Measured

- **Hashed n-gram memory** (`secrets-bin`; against no tables, more entries, four hash heads, longer orders). A `tiny` model trained on the task's generator alone, FA on `bin-fp-a` by seed: without tables 43, 188 and 172 (mean 134.3); with 12.6 M table entries 37, 24 and 4 (21.7; 2.4 standard errors); with 50 M entries 12, 5 and 13 (10.0; 4.4 standard errors), at full recall on the decidable planted-credential probe. Every seed with tables is below every seed without. Not resolved: 200 M entries, 3, 13 and 9 (8.3, against 10.0); four hash heads in 12.6 M entries, 61, 21 and 44 (42.0, against 21.7 with one); orders 2, 4, 6 and 8 in 67 M entries, 4, 6 and 34 (14.7, against 10.0). A quarter of the 50 M entries and nearly a third of the 200 M are order-2 rows that no bigram reaches (Section 4.3), so this axis overstates capacity. On `bin-fp-b`, system ELF files and Windows PE files from outside the training corpus (except eight ELF files later found in it, still counted here), FA went from 21, 52 and 96 to 14, 19 and 39 on the ELF files and from 4,605, 3,583 and 259 to 375, 62 and 1,057 on the PE files (neither resolved).
- **4-bit tables** (against 32-bit): the `secrets-bin` file went from 193.8 to 28.8 MiB and its load time from 170 to 61 ms; spans changed in 22 of 2,000 test windows.
- **Ternary projections** (against float): on `bin-fp-a` without tables, a float control had FA 114, 39 and 17 (mean 56.7) against 43, 188 and 172 for the ternary model (134.3), a factor of 2.4 in the means that three seeds do not resolve (1.3 standard errors; one ternary seed is below the float mean), while the tables cut the ternary model's mean by a factor of 13. A float model with tables was not trained. In language modeling at 3.8 M parameters (two seeds, paired), ternary weights cost 4.6 % in bpb and nothing on the downstream task, which saturated. In this release they save size, not operations (Section 4.5).
- **Zeroing threshold** $s/2$, the rounding of BitNet b1.58 (against $s/4$, fewer zeros): an earlier binary recipe, six seeds, FA on `bin-fp-a` of 12, 5, 13, 103, 9 and 1 at $s/2$ (mean 23.8) against 9, 0, 9, 232, 1 and 10 at $s/4$ (43.5), not resolved (0.3 standard errors). Each mean is set by one tail seed; without it the other five average 8.0 and 5.8. We kept $s/2$, and the trainer's recipes do not expose the threshold. A threshold with more zeros, such as the $0.7\,\mathbb{E}|W|$ of Ternary Weight Networks [67], was not tried.
- **Causal body** (against `hydra`, the weight-tied bidirectional variant that the trainer's bidirectional mode builds, Section 4.4; `bimamba` and untied bidirectional blocks were not tried on the tasks). For `pii`, in an earlier recipe with a smaller public pool and at an operating point chosen for F2 on held-out selection sentences, mean F2 on the PII Masking Benchmark was 0.7543, 0.7560 and 0.7597 causal against 0.7490, 0.7543 and 0.7511 bidirectional (-0.0052; 2.3 standard errors), a tie by the pre-registered rule of that comparison (a change under 0.01), although the bidirectional validation F1 was higher (0.913 against 0.84–0.85). On the code task it also tied on generated test windows. The released models run in the window context, so the ability to stream did not decide the choice. The cost per training step was not measured under controlled load.

- **Only `ssm_v2` blocks** (against late attention, attention first, gated delta-rule mixers [192], an MLP block). In a language-modeling probe (one seed, 2,000 steps, sequences of 1,024 bytes), three state-space blocks and one late attention block beat four state-space blocks by 15 % in bpb on unseen code, the bytes whose 32-byte string never occurs in the training data (1.58 to 1.34), with fewer weight multiply-adds per byte (1.71 M against 1.91 M; attention's own products, which grow with the context, not counted), and by 23–25 % with n-gram tables added. Attention first was worse than none; gated delta-rule mixers gained 0.5–2.5 % in bpb for 2 to 3 times the training time; an MLP block in place of a state-space block was worse. On the real tasks, with the recipes otherwise unchanged (for code, at the learning rate of 3×10^{-4}), late attention raised the findings of a `nano` model on `code-fp-b` from 93, 70 and 74 (mean 79.0) to 104, 88 and 129 (107.0; 2.1 standard errors), and FA of the `tiny` model on `bin-fp-a` from 43, 188 and 172 (134.3) to 65, 886 and 30 (327; not resolved).
- **A small body** (against 3.8, 12.6 and 18.7 M parameters). On code, larger bodies raised the findings on `code-fp-b` (150, 180 and 343 from 3.8 to 18.7 M, one model each, too few to resolve; the 1.9 M body was not measured in that setting); shrinking 12.6 M to 1.9 M kept the F1 on generated test windows (0.976 → 0.979) and cut the cost per window from 209 to 46 ms on the research engine of the time. For `pii`, in the earlier recipe of the causal-body item, the larger preset `tiny_n18` gave mean F2 0.7369, 0.7469 and 0.7477 against 0.7543, 0.7560 and 0.7597 (−0.013; 3.4 standard errors, every seed below every seed of the smaller preset).
- **Max pooling in the gating head** (against `last_mean`): with `last_mean`, the mean half dilutes a 40-byte credential in a 512-byte window 1:13, and the last state sees only about the last 256 bytes (next item); in one early run the window accuracy stayed at 0.53 against a chance level of 0.51.
- **The code task is local**, the basis of 512-byte windows with 128 bytes of overlap: cutting a window's left context to 32 bytes did not hurt the span F1 on generated windows (0.914 → 0.930, standard error 0.012), although the network uses 192 to 256 bytes internally: the positions whose output changes fall from 5,320 with no left context to 69 with 192 bytes and 0 with 256. The overlap of 128 bytes thus guarantees each byte less left context than the network uses. We did not compare other window sizes or strides.
- **Float activations** (against int8): an int8-activation path of the research engine ran 2.2 times faster, changed spans in 23 to 29 of 14,100 windows and lost one planted credential; it was not released.
- **Window context** (against running a window-trained model as a stream): a research model of the code task, run over each file as one stream without retraining, disagreed with its windowed run at 23,211 of 79,488 positions compared on 28 files of 600 bytes to 16 KiB from a public repository of planted credentials made to test scanners, where its windows had flagged 1,232: 23,202 were new flags, and 1 of 14 spans was lost. On 59 development files it lost both of their 2 spans.

Limits of the claim

Seed variance is larger than many of these effects: across the three seeds of one recipe, FA on `bin-fp-a` ranged from 2 to 27, and from 43 to 188 in another; most comparisons use three seeds, and several above are not resolved. Most ablations were run with the research code and on earlier recipes, not with runtime 1.0.0 and the released recipes (the runtime reproduces the research engine bit for bit on the paths the released models use, Section 5.2). The exams that chose the architecture also report the released models (Section 8). Gains in bits per byte did not predict the real tasks: late attention improved language modeling and worsened false alarms on code. The configuration is the best one found for three tasks, not a claim about others.

5 The runtime

The runtime is `libpurebyte`, a C++17 library with no third-party dependencies (it links only the platform's thread library), a C API with a stable ABI, and three front ends on that API: the `purebyte` command-line tool, a local HTTP server and Python bindings. It executes any model declared by a file that follows Section 4 and knows nothing about tasks: the only task-specific code is in the post-processing profiles, which a model names and a caller can override. This section describes runtime 1.0.0 (model format version 3); its performance figures come from one desktop and are given with their conditions (Section 5.6).

5.1 Layers, registries and profiles

Layers. Under `engine/src/`, a bounds-checked GGUF reader (`format/`) and the kernels (`kernels/`), one table of functions per instruction set, sit at the bottom. Model assembly (`model/`, `blocks/`, `heads/`) turns a file into a model; the runtime layer (`runtime/`) walks windows or streams and runs them on threads; the detect layer (`detect/`) applies profiles, ensemble votes, masking, redaction and archive expansion. The C API (`purebyte/pb.h`) is the only stable interface. Loading reads the whole file, checks the container, the format version, the required features and the ternary encoding, builds every part in order through the registries, refuses the file if a tensor was left unused and checks the restrictions of the stream context; a loaded model is immutable and can be shared by any number of threads and sessions.

Registries. Four tables name the parts that can be extended: block types (`ssm_v2`, `attention`), head types (Section 4.7), profiles (`none`, `secrets-code`, `secrets-binary`, `redact`) and kernels (`scalar`, `avx2`, `neon`). A new block, head or profile type is one module plus one line in its registry; a new model component also needs its text in the format specification, a function in the NumPy reference and random models in the test suite, and a new kernel must be bit-identical to the scalar kernel.

Profiles. The profile is the caller's, else the file's `purebyte.profile`, else `none`, which reports the heads' outputs in a generic schema and resolves overlapping spans of one type into their union. `secrets-code` and `secrets-binary` add rules for credentials in text and in binaries; `redact` replaces every resolved span by a typed marker and checks, before returning, that every byte outside the spans is copied unchanged and in order. With an ensemble, a span is kept by default when a majority of the members find it. Two accelerations are not exact and must be requested: a string prefilter, which skips the windows of a binary that no printable run touches (by default, runs of 16 ASCII or 16 UTF-16LE characters), and early exit, which needs an `exit` head.

5.2 The numerics contract

The specification fixes the order of every floating-point operation, and the portable scalar kernel is its executable form; the AVX2 and NEON kernels compute the same IEEE operations in the same order and differ only in scheduling. The contract has five rules.

1. **Operations as written**: binary32 operations rounded to nearest-even, in the written order. Compilers must not contract $a \cdot b + c$ into a fused multiply-add, reassociate sums or keep intermediates in a wider format (C's `FLT_EVAL_METHOD` must be 0, which excludes x87 arithmetic); the build uses `-ffp-contract=off -fno-fast-math -O2` with GCC and Clang, and `/fp:precise` with MSVC 19.14 or newer, with `#pragma fp_contract(off)` in the sources that compute numerics (versions of MSVC before 2022 may contract under `/fp:precise`).
2. **A correctly rounded fused multiply-add**, $\mathrm{fma}(a, b, c) = \mathrm{fl}(a \cdot b + c)$ with one rounding. The AVX2 and NEON kernels use the processor's fused multiply-add instructions in their vector lanes. The scalar code uses the hardware instruction only when GCC or Clang compile for AArch64, or for x86-64 with FMA enabled; every other build (MSVC on any architecture, 32-bit ARM, RISC-V,

and x86-64 without FMA, which is how the scalar kernel is compiled on x86-64) uses an exact emulation [193]: the product $p = a \cdot b$ is exact in binary64 (24 + 24 significant bits fit in 53), the sum $s = p + c$ is computed in binary64 and its error e recovered with TwoSum [194], s is moved to its neighbor on the side of e when $e \neq 0$ and its last significand bit is 0 (rounding to odd), and one final rounding to binary32 gives the correctly rounded result. The C library's `fmaf` is not used, because some implementations round twice. The emulation costs about ten binary64 operations per multiply-add, and on a floating-point unit without binary64 arithmetic (Cortex-M4F, Cortex-M33, RISC-V with the F extension alone) those operations are themselves emulated in software.

3. **A canonical dot product** for every matrix-vector product of the blocks and of the n-gram module, with $n_8 = n - (n \bmod 8)$ and eight lanes starting at 0:

$$\begin{aligned} &\ell_{i \bmod 8} \leftarrow \mathrm{fma}(a_i, x_i, \ell_{i \bmod 8}) \quad (i = 0, \ldots, n_8 - 1), \\ &r \leftarrow \big((\ell_0 + \ell_4) + (\ell_1 + \ell_5)\big) + \big((\ell_2 + \ell_6) + (\ell_3 + \ell_7)\big), \\ &r \leftarrow r + a_i\, x_i \quad (i = n_8, \ldots, n - 1, \text{ each product rounded before its sum}), \qquad \langle a, x \rangle = r. \end{aligned} \tag{22}$$

   An 8-wide AVX2 register holds exactly these lanes; on NEON they are a pair of 4-wide registers.

4. **Fixed-order normalization**: Equation (14), the squares summed in binary64 in element order.

5. **A polynomial exponential inside the blocks**: the SiLU of the convolution and of the gate uses the Cephes single-precision polynomial [195], with a fixed clamp at ±88.3762626647949, a fixed chain of fused multiply-adds and the power of two built from exponent bits (constants in the specification), so that it does not depend on the C library. Elsewhere the platform's `expf`, `logf`, `log1pf`, `sqrtf`, `sinf` and `cosf` are used; `sqrtf` is correctly rounded, the others may differ by a few units in the last place between C libraries.

Guarantees. A *platform* here means one build of the runtime with one C library, run in the default floating-point environment (rounding to nearest, no flush-to-zero, no denormals-are-zero) in every thread that computes, the caller's included; the runtime neither sets nor checks that environment, which code built with fast-math options can change. On one platform every kernel gives the same bits, at any number of threads (Proposition 6). Across platforms results differ only through the C library's functions (rule 5), amplified by the layers: decisions and spans are expected to agree, raw values to a relative difference of about 10^{-4} or less. The Linux release binaries are static and carry their C library, but the Linux wheels use the host's glibc, so the same wheel on two distributions may count as two platforms. The tests compare accordingly: across implementations or platforms, window geometry, flags, labels, levels, spans and byte maps must be identical, head values must satisfy $|a - b| \le 10^{-5} + 10^{-4}|b|$ and final hidden states $|a - b| \le 10^{-4} + 10^{-3}|b|$; the observed differences use about 1.5 % of these tolerances.

Proposition 6 (Invariance to threads)**.** *For a given kernel on one platform, with every thread in the default floating-point environment, the bits of every window's hidden states and head values do not depend on the number of threads, on the team size or on the order in which windows are scheduled.*

Proof. Every stage of a window is split among the members of its team into disjoint ranges of independent outputs (Section 5.4); each output element is computed by one thread, with the operations of the single-thread code, and no value is reduced across threads. The SIMD kernels group rows and positions according to where a range starts and ends, but every grouping computes the canonical dot product of each row at each position, and each scan item (one head, eight of its channels, all positions) runs on one thread. Windows are independent, each result goes to a slot fixed by the window's index, and the heads run on one thread. □

Measured

- **Kernels**: every SIMD kernel matches the scalar kernel under `memcmp` on random data with lengths of 1, 7, 8 and 21, partial channel blocks, carried state and convolution taps before the first row; the NEON kernel is checked natively on macOS arm64 by the continuous-integration workflow and on Linux arm64 by the release workflow, and on x86-64 through SIMDe by a local tool.
- **Emulated FMA**: identical to the hardware instruction on 2,000,000 random triples with factor exponents between −30 and 29 and canceling products (no subnormal results, overflows, infinities or NaNs).
- **Models**: seven random models written from the specification (every block type and direction, every head type and pooling, 32- and 4-bit n-grams, gate, lookahead, queries, streams, gating, early exit, sizes that are not multiples of 8) give identical bits in six thread and kernel arrangements and match the NumPy reference and golden files within the tolerances above; the latency runs of Section 5.6 gave the same digests at every thread count, and the scalar and AVX2 kernels the same digest on a window of `secrets-code`.
- **Research engine**: before the release, 32 cases and 326,581 window results were identical to those of the research engine that validated the released models.

5.3 Kernels

Each kernel is a table of eight functions: the matrix product over the positions of a window, matrix-vector products, inverse RMS factors, the scaled normalization, the convolution with SiLU, SiLU, the gate and the scan. `auto` selects `avx2` when the CPU has it, else `neon`, else `scalar`; forcing a kernel the CPU cannot run returns `PB_ERR_UNSUPPORTED`. Table 7 summarizes them. No kernel skips the zeros of a ternary row: a row is expanded to floats (code times scale, exact) and multiplied in full. The AVX2 code lives in a translation unit compiled with `-mavx2 -mfma` (or `/arch:AVX2`), with internal linkage only, and runs after the CPU check.

The scalar kernel's speed. The scalar kernel runs one lane at a time and, on x86-64, with the emulated fused multiply-add of rule 2. It is the kernel of every x86-64 processor without AVX2 and FMA (the older Atom-class processors of many gateways and storage devices, or a virtual machine that hides AVX2) and of every architecture other than x86-64 and arm64, and it is about two orders of magnitude slower per core: in one session on the test desktop, one 64-byte window of `secrets-code` on one thread took a median of 524 ms with the scalar kernel against 5.1 ms with `avx2`, with the same digest (a MinGW-w64 GCC build; the machine was busy with other work, so only the ratio is meaningful).

Where the time goes. A profile taken before the current kernels found the ternary matrix products taking about two thirds of the time of a window and the scan most of the rest; heads, decoding and the n-gram memory took a few percent (the heads take 3 % of a one-thread window in the current build). The CPU time per window was then the same with 1 and 12 threads. With the current kernels it is not: a batch that keeps 12 threads busy spends 1.4 to 1.5 times the one-thread time per window (Section 5.6); lower all-core clocks, shared caches and memory may each contribute, and we do not claim that the engine is free of bandwidth limits. No profile of the current kernels is published. The multiply-adds of Proposition 4 over the one-thread latencies of `secrets-code` (Section 5.6) give about 45.7 G multiply-adds per second at 64 bytes and 43.0 G at 1,024 bytes on one core of the test machine.

5.4 Threads

A *session* owns the kernels in use, a pool of persistent threads (the caller is thread 0) and one workspace per team, grown on demand and reused; it runs one call at a time, and a model can be used by several

Table 7: The three kernels. All compute the operations of the numerics contract in its order; the table lists how they schedule them.

Kernel	Where	How
`scalar`	every platform	the executable specification and the oracle of the kernel tests; one row and one position at a time; FMA emulated in binary64 except in GCC or Clang builds for AArch64 (rule 2); about 100 times slower than `avx2` per core
`avx2`	x86-64 with AVX2 and FMA, detected at run time (CPUID, and XCR0 for the YMM state); the default on x86-64, including CPUs with AVX-512	tiles of $\max(4,\ 2^{17}/(4\,in))$ positions (128 for $in = 256$) whose input rows stay in L2; output rows in blocks of 8, or of 4 at two positions sharing each weight load; ternary rows expanded 8 codes at a time; RMS sums of 16 rows in binary64 lanes; the scan with 8 channels of a head in the lanes and the state n-major, so the accumulators are the canonical lanes, and a bit mask that keeps the D term's product and sum unfused; partial channel blocks use the scalar scan
`neon`	AArch64, whose baseline includes Advanced SIMD	each 8-lane accumulator is a pair of 4-lane registers, reduced by a lane-wise add and two pairwise adds; output rows in blocks of 4, or 4 at two positions; RMS sums by the scalar code; the MAXPS/MINPS semantics of the exponential's clamp emulated with compare-and-select

sessions at once. A call divides the pool into teams of $\min(8, \max(1, \lfloor threads/windows \rfloor))$ threads, or of the caller's `intra_threads`, so that when a call has fewer windows than threads the idle threads join the windows. Thread i joins team $\lfloor i/team \rfloor$ as member $i \bmod team$; threads left over stay idle for that call. Each team's leader takes the next window from a shared atomic counter and publishes it, the members run the same forward pass (single program, multiple data), and the leader runs the heads and writes the result into the window's slot. Within a window, every stage is split among the members into contiguous ranges whose inner boundaries are multiples of an alignment, the remainder spread over the first members: the output rows of the projections (aligned to 8), the positions of the normalizations (aligned to 16), of the step sizes, of the convolution and of the residual update, and the (head, block of 8 channels) items of the scan. Barriers separate stages that read one another's outputs: one when the leader publishes the window, one after the embedding, seven per causal `ssm_v2` block and one after the final normalization. A barrier spins up to 4,000 times with the processor's spin hint (PAUSE on x86-64, YIELD on AArch64), for a time that depends on the CPU and that we have not measured (the code's estimate is 0.1 to 0.5 ms on x86-64; on arm64 cores, where YIELD costs little, it is probably much shorter), then sleeps on a condition variable.

Why at most eight threads per window. On the test machine, the medians with 12 threads were higher than with 8 on the 64- and 256-byte windows of all three specialists and on the 1,024-byte window of `secrets-code` (Section 5.6), and an earlier version that let one 64-byte window take all 23 threads of the command-line default took 3.1 ms, slower than one thread. On the 1,024-byte windows of `pii` and `secrets-bin`, the medians with 12 threads were slightly lower than with 8 (13.485 against 13.677 ms; 22.787 against 24.472 ms), differences of 1.4 and 7 % that we did not compare with the spread between runs. The cap is a default that `intra_threads` overrides. It was chosen on a processor whose 12 cores sit on two dies of six, each with its own 32 MB L3 cache, so that a team of more than six threads spans both dies; it may reflect that topology. In the stream context, teams take whole inputs and run their chunks in order; `pb_encode`, which returns the hidden states of one sequence, uses all threads as one team. The command-line tool uses the hardware threads minus one by default (23 on the test machine).

5.5 Memory

A process that runs a model holds (1) the model file, read whole into one buffer (there is no memory mapping), from which float tensors and the 4-bit n-gram tables are used in place, table rows being decoded as they are looked up; (2) the ternary codes, unpacked at load to one signed byte per weight; (3) one workspace per team, grown to the longest window the team has run; and (4) the process itself: executable, C and C++ libraries, thread stacks, heap and the inputs being processed. For causal `ssm_v2`

Table 8: Where the memory of a process goes. Top: derived from the source code and the tensor shapes. Bottom: measured peaks, the maximum resident set size of the whole process (`/usr/bin/time`), on the test desktop under Linux x86-64 (WSL2). One decision: `purebyte decide` on a 29-byte input, 1 or 23 threads alike. Larger inputs, measured in a session other than that of Table 10, on other inputs for `secrets-code` and `pii` (whether the DLL is that table's was not recorded): `secrets-code` on 149 source files (1.8 MB, 15 s with 23 threads, 35 s with 4), `pii` redacting 133 kB of Markdown, `secrets-bin` on a Windows DLL of 4.8 MB (82 s).

	`secrets-code`	`pii`	`secrets-bin`
Model file buffer (bytes)	1,028,704	4,764,864	30,163,488
Unpacked ternary codes (bytes)	1,835,008	1,835,008	3,670,016
Workspace of a one-thread team, 512-byte window (bytes)	7,539,744	7,539,744	7,539,744
Peak, one decision on 29 bytes	9.4 MB	13 MB	39 MB
Peak, larger input, 23 threads	188 MB	182 MB	220 MB
Peak, larger input, 4 threads	42 MB	—	—

blocks without a lookahead, a window of T positions and a team of μ members, the workspace holds

$$2Td + T + S(T) + \mu M \text{ floats}, \qquad S(T) = T\,(2d + 1 + p_o + 2H + c_w + d_i) + (K-1)\,p_o + HPN, \tag{23}$$

for the residual stream and the hidden states, the scratch shared by the blocks (the largest over the blocks, independent of their number) and a private scratch of $M = \max(8\max(d, d_i) + H,\ w + 3d)$ floats per member (the second term with an n-gram memory only). For the released shape and $T = 512$, a one-thread team holds 1,884,936 floats, 7,539,744 bytes; for a 29-byte window, under 1 MB. Table 8 sets these components next to measured peaks.

For one small decision, the file and the unpacked codes account for 2.9, 6.6 and 33.8 MB of the peaks; the rest, about 5 to 7 MB in each case, is the process itself and a workspace of under 1 MB. For large inputs the workspaces dominate: when a call has at least as many windows as threads, every thread forms its own team, and 23 workspaces of about 7.5 MB hold about 173 MB (about 30 MB with 4 threads), which with the model and the process accounts for the measured peaks to within a few megabytes. In the window context the working memory grows with the number of threads, not with the length of the input, so the thread count is the control on a device with a memory budget. What a call keeps for an input does grow with it: the input, its results and, when redacting, the redacted copy (`pii` redacting 3.9 MB of JSON Lines with 23 threads peaked at 239 MB under Windows, about 50 MB more than the workspaces, the model and the input account for). A whole input run as one window (`PB_SCAN_WHOLE`) and `pb_encode` need, by Equation (23), a workspace of about 14.4 kB per byte of input.

5.6 Performance

Unless noted, the measurements were taken on an idle AMD Ryzen 9 5900X (12 cores and 24 threads, AVX2, no AVX-512; 1.8 % CPU load before the session) under Windows 10, with the release build of runtime 1.0.0 (MSVC, static runtime) and the released files, one request at a time, the model loaded; the engine does not pin its threads to cores, and we report medians only. Table 9 gives the latency of one window against the threads that share it (Figure 8 draws it), measured with the test suite's `window_latency` tool: one synthetic input run as one window by a session whose threads all work on it, two warm-up runs, then the median of 40. The timed call also computes a 64-bit digest of the window's hidden states and head values, which a normal decision does not: timed separately on this machine, about 0.06 ms for a 64-byte window and 1 ms for a 1,024-byte one, up to 9 % of the eight-thread times and about 2 % of the one-thread times. Whether the gated heads ran was not recorded, and the profile's post-processing is not included. The 1,024-byte rows are one window of 1,024 bytes, twice the released models' window; in normal use such an input runs as three windows of at most 512 bytes. Every thread count gave the same digests.

Table 9: Latency of one window in milliseconds against the threads that share it: medians of 40 runs after two warm-up runs, idle AMD Ryzen 9 5900X, Windows 10, release build, model loaded, the digest of the hidden states included. Bold: the lowest median. Speed-up: 1 thread against 8.

Specialist	Window	1 thread	2	4	8	12	Speed-up
`secrets-code`	64 B	2.966	1.617	0.910	**0.817**	0.923	3.6
	256 B	12.005	6.391	3.538	**2.807**	3.127	4.3
	1,024 B	50.431	26.507	14.486	**10.504**	11.413	4.8
`pii`	64 B	3.200	1.866	1.044	**0.886**	1.043	3.6
	256 B	13.069	7.002	4.125	**3.096**	3.408	4.2
	1,024 B	54.770	28.714	19.089	13.677	**13.485**	4.0
`secrets-bin`	64 B	5.917	3.158	1.737	**1.449**	1.709	4.1
	256 B	24.265	12.665	6.799	**4.951**	5.689	4.9
	1,024 B	102.463	53.250	30.449	24.472	**22.787**	4.2

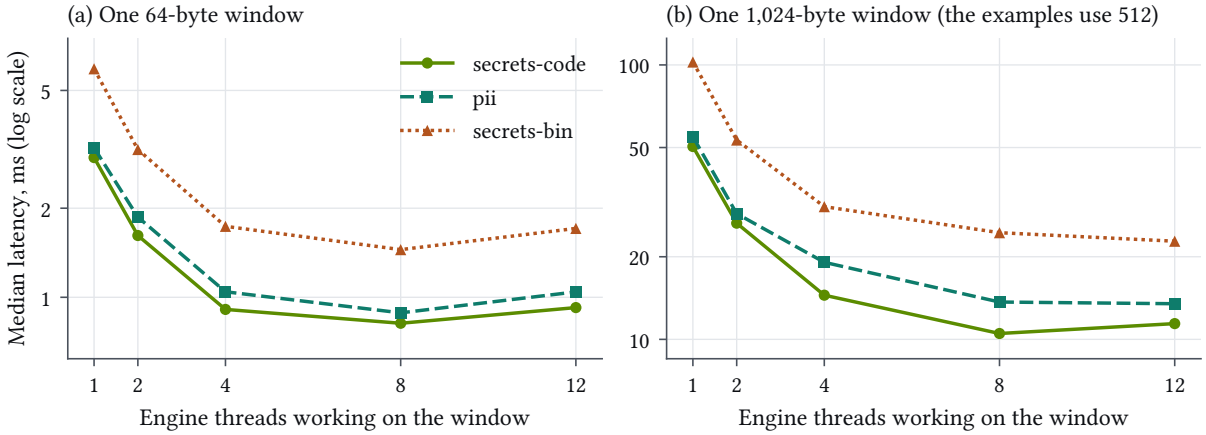


Figure 8: Table 9 as curves: median latency of one window against the engine threads that share it, for (a) 64-byte and (b) 1,024-byte windows, on a logarithmic axis. Short windows are fastest with eight threads; the 1,024-byte windows of `pii` and `secrets-bin` still gain a little at twelve.

With 8 threads a 64-byte window takes 0.817 to 1.449 ms and a 256-byte window 2.807 to 4.951 ms, depending on the specialist, and with one thread 2.966 to 5.917 ms and 12.005 to 24.265 ms. On one thread a 1,024-byte window takes 4.20, 4.19 and 4.22 times a 256-byte one, about 5 % more than the factor of 4 that the linear cost of Proposition 4 predicts. The lowest latency is not the lowest CPU time: eight threads for 0.817 ms are 6.5 ms of thread time, against 2.966 ms on one thread, and a batch that keeps 12 threads busy spends 12/2,739 s, about 4.4 ms of thread time, per 64-byte window (Table 10). A 64-byte decision of `secrets-code` thus costs 3 to 6.5 core-milliseconds, depending on how the threads are used.

Table 10 gives throughput and end-to-end times, and Table 11 the effect of the current kernels, of the two opt-in accelerations and of threads that share a window, each against the engine without the change. Except for the last row, those speed-ups were measured with the research engine on 22 and 23 September 2026, while the desktop was busy with other work; the compared configurations ran interleaved, so that the load weighed on both sides, and only the ratios are meaningful. The early-exit rows used a research model of `secrets-bin` with a probe after block 3 of 8, where 78 to 97 % of the windows of real binaries exited early; the released files have no exit head, so early exit is not available with them.

Against a general decision model. Laya [196] is a non-generative decision model of 421 M parameters (English checkpoint) that answers a typed question written at request time. On the same idle desktop, its `laya` 0.3.6 package with PyTorch 2.14 on the CPU answered one three-option question about

Table 10: Throughput and end-to-end times on the test desktop (AMD Ryzen 9 5900X; Windows 10 except in the last row), with the conditions of each row.

	`secrets-code`	`pii`	`secrets-bin`	Conditions
Windows per second	2,739	2,637	1,326	a batch of 48 windows of 64 bytes, 12 threads, automatic team size (teams of one), median of 4 runs of `window_latency`, digests included
`purebyte bench`: 64-byte, 1,024-byte input	0.92, 10.3 ms	1.21, 10.9 ms	2.01, 20.5 ms	default threads (23), medians of 40 runs; the model's windows (three for 1,024 bytes), model and span decoding, not the profile
Whole inputs	131 kB/s	126 kB/s	65 kB/s	command-line defaults, 23 threads, wall-clock, median of 3 runs: PureByte's source code with its training stack (2.1 MB, 16.3 s); 3.9 MB of the PII Masking Benchmark's sentences redacted (30.9 s); one Windows DLL of 5.0 MB (77 s)
Start, load, answer	0.01 s	0.03 s	0.16 s	`purebyte decide` on 29 bytes, Linux x86-64 in WSL2 on the same desktop, timed by `/usr/bin/time`, whose resolution is 0.01 s

Table 11: Measured speed-ups, each against the engine without the change, on the test desktop (AMD Ryzen 9 5900X, Windows 10). The first five rows come from the research engine while the desktop was busy, with the compared configurations interleaved: per window, one 512-byte window on one thread, medians of 25 rounds, in two sessions that gave the two ends of each range; source code, 2 runs of each configuration (they differed by under 1 %); prefilter, the best of 2 runs; early exit, one run. The last row comes from Table 9 (medians of 40).

Change	Where	Speed-up	Output
Current kernels (default)	per window, `secrets-code`-sized and `secrets-bin`-sized models	1.33–1.39, 1.29–1.32	bit-identical
Current kernels (default)	end to end, scanning source code	1.26	identical findings
String prefilter (opt-in)	end to end, real binaries; skips 81 to 89 % of their windows	5.2	loses only findings outside printable strings
Current kernels and early exit	end to end, real binaries, prefilter off, against the previous engine	2.05	the same number of findings in that test
Kernels, prefilter and early exit	end to end, real binaries, against the previous engine with none of them	13.2	loses only findings outside printable strings
Threads sharing one window	one 64-byte window, 8 threads against 1	3.6	bit-identical

one line of code (173 bytes, 190 tokens with the question) in a median of 373 ms with 12 threads (its fastest) and 1,382 ms with one (medians of 15 calls after 3 warm-up calls); `secrets-code` answers its own question about a 256-byte window, which holds that line, in 2.81 ms with 8 threads, 133 times less. Laya ran in PyTorch, without an optimized CPU runtime; even so, its time per operation was close to the specialist's (about 160 GFLOP for 421 M parameters over 190 tokens in 373 ms, against 1.1 GFLOP for the 256-byte window in 2.81 ms), so the ratio reflects mostly the difference in size and generality, not in engineering. The specialist's time is that of Table 9: a synthetic 256-byte input, digest included. Figure 9 adds the figures that Laya publishes for itself on a T4 GPU and quotes for a hosted decision API. The comparison is of time, not of quality, and the jobs differ: a specialist of 1.9 M parameters trained for one fixed question against a general model of 421 M that reads its question at every request; neither was evaluated on the other's tasks.

Limits of the claim

Every inference speed in this paper comes from one desktop with an x86-64 CPU and AVX2, apart from figures quoted from others, which are marked; none comes from an arm64 CPU, the NEON

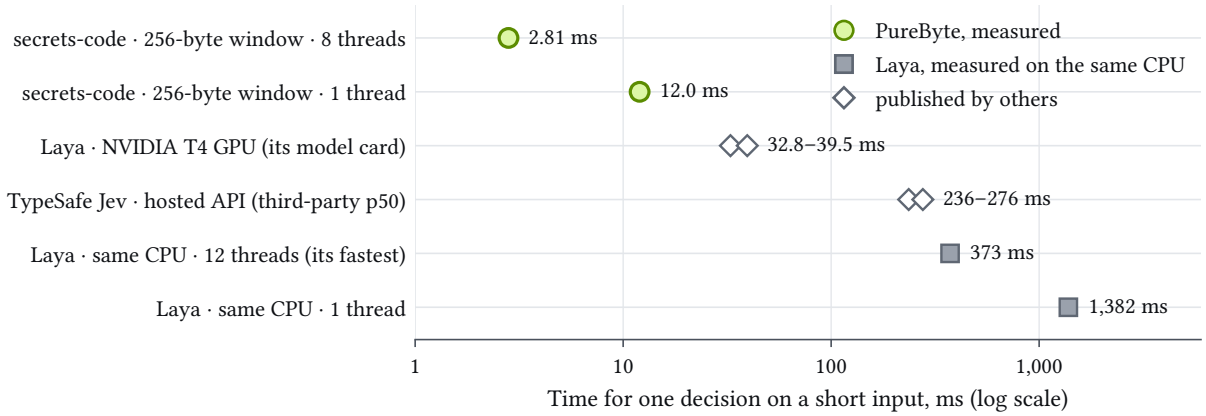


Figure 9: Time for one short decision, on a logarithmic axis. Measured on the same idle AMD Ryzen 9 5900X: PureByte `secrets-code` (the model loaded; medians of 40 runs) and Laya's English checkpoint (`laya` 0.3.6, PyTorch 2.14, one three-option question about a 173-byte line; medians of 15 calls). Open diamonds: figures published by others, as quoted in Laya's model card (Laya on an NVIDIA T4; TypeSafe Jev's hosted API, third-party p50), not measured by us. The specialist reads a 256-byte window, which holds Laya's line; on a logarithmic axis the horizontal distance between two points is their ratio.

kernel, or Linux or macOS hardware, and only the memory and loading figures were taken under Linux (WSL2, same desktop). We report medians without their spread and no worst-case latency, which is what a real-time use needs. The speed-ups of the research engine and the scalar kernel's ratio were measured while the desktop was busy. Absolute times depend on the machine and its load more than the ratios do; the cap of eight threads per window was chosen on this machine, and the profile of where the time goes predates the current kernels.

5.7 Interfaces

C API. `purebyte/pb.h` (ABI version 1) reports the failures of every fallible function as a status (`OK`, `ARGUMENT`, `IO`, `FORMAT`, `UNSUPPORTED`, `NO_MEMORY`, `BUFFER_TOO_SMALL`, `INTERNAL`) with a message, and keeps no global state. Option structures start with their size and only grow by appending fields, so a newer library accepts an older program's structure and an older library refuses a larger one. The calls load and describe models, create sessions (threads, team size, kernel), scan (`pb_scan`: every window's head values, spans, byte maps and digests, with overrides of the window, stride, prefilter, operating bias and queries), encode one sequence (`pb_encode`), detect with a profile and an optional ensemble (`pb_detect`, whose JSON results the output specification defines), redact, restore and expand archives.

Command line, HTTP and Python. `purebyte` has the commands `scan`, `decide` (one input of up to 4,096 bytes), `redact`, `serve`, `models`, `bench`, `info` and `version`, writes JSON, JSON Lines and SARIF, and sets the model or ensemble, profile, operating bias, threads (1 to 1,024), team size and kernel. `purebyte serve` listens on `127.0.0.1:8421` by default (`GET /health`, `GET /v1/models`, `POST /v1/scan`, `/v1/decide`, `/v1/redact`), runs one inference at a time on all its worker threads, accepts bodies of up to 80 MiB by default, checks the `Host` and `Origin` headers, requires an `X-PureByte` header on form-like requests, accepts an optional bearer token, writes nothing to disk and logs no content. The Python package binds the shared library with `ctypes`, needs only the standard library and Python 3.8 or newer, checks the ABI at import, and exposes scanning, deciding, redaction, restoration and archives (`Model`, `Session`, `Detector`), but not `pb_scan` or `pb_encode`.

Table 12: Platforms on which the repository's workflows build the runtime and run the whole test suite (unit, reference, golden, invariance, malformed-file, detection, command-line and Python tests): the continuous-integration workflow on every change, the release workflow natively on every platform it ships, before packaging. Runner names are those of the hosted service.

Workflow	Platform (runner)	Compiler
Every change	Linux x86-64 (`ubuntu-24.04`)	GCC; Clang
	macOS arm64 (`macos-15`)	Apple Clang
	macOS universal binary, arm64 and x86-64 (`macos-15`; the x86-64 slice's C++ tests under Rosetta 2 when available)	Apple Clang
	Windows x86-64 (`windows-2022`)	MSVC; MinGW-w64 GCC (UCRT64)
Release	Linux x86-64 and arm64 (`ubuntu-24.04`, `ubuntu-24.04-arm`), static binaries	the runner's compiler
	macOS arm64 and x86-64 (`macos-15`, `macos-15-intel`); Windows x86-64 (`windows-2022`)	Apple Clang; MSVC
	Linux wheels, x86-64 and arm64 (`manylinux_2_28`); container images, `linux/amd64` and `linux/arm64`	the image's compiler

5.8 Portability

The runtime builds with CMake 3.16 or newer and a C++17 compiler (GCC, Clang, or MSVC 19.14 or newer); the library needs no third-party code, and the command-line tool embeds the single-header HTTP library cpp-httplib. The SIMD kernel of the target (AVX2 on x86-64, NEON on arm64) is compiled in and chosen at run time; other architectures get the scalar kernel, provided they are little-endian (model files are little-endian, and the build stops with an error on a big-endian target) and meet rule 1 of the numerics contract. The release binaries carry their C and C++ runtimes (fully static on Linux) and run on Linux (the Python wheels need glibc 2.28 or newer), macOS 11 or newer and Windows 10 or newer, on the CPU; there is no GPU backend. Table 12 lists where the repository's workflows build the runtime and run its whole test suite, whose unit and invariance tests compare the SIMD kernel of the CPU they run on with the scalar kernel, bit for bit. Linux on arm64 and macOS on x86-64 hardware are tested by the release workflow only; on every change, the x86-64 slice of the universal macOS build runs its C++ tests under Rosetta 2 when the runner has it.

5.8.1 From servers to routers, robots and microcontrollers

What runs today. The same file runs, with the same results up to the C library differences of Section 5.2, on the x86-64 and arm64 platforms of Table 12, which are those of servers, desktops and laptops. Linux devices with an x86-64 or arm64 processor, such as the single-board computers of small robots and routers or gateways whose system-on-chip is x86-64 or arm64, can be targeted by the same build (on arm64 the NEON kernel runs; an x86-64 processor without AVX2 and FMA gets the much slower scalar kernel of Section 5.3), but none has been tested and no speed has been measured on one. For scale, the whole-input rate on the 12-core desktop, 131 kB/s with 23 threads or about 1 Mbit/s (Table 10), is far below the line rate of a router's links: a router could screen its configuration files, its logs or samples of its traffic, not all of it. For a robot the figure that matters is the worst-case latency, which we have not measured. By Section 5.5, `secrets-code` with one thread and full 512-byte windows needs about 1.03 MB for the file, 1.84 MB for the codes, 7.54 MB for the workspace and the process itself, about 15 to 17 MB in all; one small decision peaked at 9.4 MB. Other architectures (32-bit ARM, RISC-V, MIPS) would fall back to the scalar kernel, with the emulated fused multiply-add of rule 2, if they are little-endian and meet rule 1; we have not built the runtime for them.

What a microcontroller would need. A microcontroller typically runs without an operating system and keeps code and constants in flash memory, with far less RAM; the process, threads and heap of

Table 13: `secrets-code` on runtime 1.0.0 and on a hypothetical microcontroller port that keeps the same weights. Left: measured or derived from the source code. Right: analysis only; nothing in it has been built or measured.

	Runtime 1.0.0 today	A microcontroller port (analysis)
Weights	the 1,028,704-byte file, read into RAM	the same bytes in flash, read in place
Ternary codes	one byte per weight: 1,835,008 bytes of RAM	decoded from the 2-bit codes as they are read: no RAM, but decoding work at every multiply-add
Weight traffic	weights in RAM; the AVX2 kernel reuses each over a tile of positions (Table 7)	no reuse across positions: about 0.63 MB read from flash per position, about 30 MB/s at $R = 10^8$ (below)
Activations	a workspace per team that holds all positions of a window: 7,539,744 bytes for 512 bytes, under 1 MB for 29	one position at a time with the state carried: $4 \times 140{,}288$ bytes of state, about 13 kB of per-position vectors and 12 kB for the heads (label log-probabilities and Viterbi back-pointers of every position, a running maximum), about 0.59 MB. Windows start from a zero state, so the 128 bytes that consecutive windows share are computed twice: a third more compute in any case, with a second state (0.56 MB more) or by reading the overlap again (no more RAM)
Arithmetic	binary32 activations, binary64 sums in the normalizations and the decoder; the same bits on a platform	the same arithmetic keeps the bits, and is fast only with the core's single-precision FMA instruction (runtime 1.0.0 emulates it in binary64 on 32-bit ARM and RISC-V); int8 or int16 activations would shrink state and vectors 2 to 4 times but change results (an int8 research path changed spans in 23 to 29 of 14,100 windows)
Process	executable, libraries, threads, heap: about 5 to 7 MB	static buffers sized from the file, no threads, no file system; not estimated
Compute	2.12 M multiply-adds per position (Proposition 4); with a window every 384 bytes, 2.82 M per byte of input	at a sustained R multiply-adds per second, $R/2.82$ M bytes of input per second: 35 at $R = 10^8$, about 1.4 s for one 64-byte decision and 11 s for one 512-byte window
Fits	servers, desktops and laptops (Table 12)	parts with at least 0.6 MB of SRAM and 1.1 MB of flash (i.MX RT1060, Ensemble E3); not the nRF52840, RP2040, RP2350, or ESP32-S3 without external memory
Smaller shape	$d = 256$, $H = 8$, $P = N = 64$	$d = 128$, $H = 4$, $P = 64$, $N = 32$ ($G = 2$, $K = 4$): 133,120 multiply-adds per position and block, a quarter, and 37,376 bytes of state per block; with four blocks, about 170 kB of RAM and a file of about 0.38 MB, within the nRF52840, RP2350 and ESP32-S3; untrained, quality unknown

Section 5.5 do not exist there. Table 13 sets the needs of `secrets-code` on runtime 1.0.0 against those of a port that keeps the same weights. The analysis rests on one property: in a causal `ssm_v2` block every step at position t reads only positions up to t, and the order of operations of the contract does not depend on the window length, so a port that evaluates one position at a time can reproduce a window's hidden states bit for bit with a runtime built on the same C library functions, as the stream context does chunk by chunk. The heads need the whole window only through a running maximum, the Viterbi back-pointers and each position's label log-probabilities (for the span confidences); the gated `tag` head must run at every position, its result discarded if the gate is negative. The changes are thus the core's own single-precision fused multiply-add (runtime 1.0.0 would use the binary64 emulation of rule 2 on 32-bit ARM and RISC-V, even on cores that have the instruction), weights read in place from flash with the ternary codes decoded on the fly, position-by-position execution and static allocation sized from the file. Without reuse across positions, every position reads all the block weights again, about 0.63 MB

from flash, and decodes their 2-bit codes. Table 13 checks the result against the parts of Section 2.6, whose integer NPUs and vector units (Ethos-U55, the ESP32-S3's vector instructions) cannot run the binary32 contract. Devices without the RAM or the floating-point throughput would also need a smaller shape or integer activations, each a different model to train and verify like the released ones.

Limits of the claim

No microcontroller port exists: the right column of Table 13 is arithmetic on the specification and the source code, and its processing rate is an example of the formula, not a property of any device. Whether a smaller or integer model would keep the quality of the released specialists is unknown. Routers, single-board computers and other devices outside Table 12 can be targeted by the portable build but have not been tested or measured.

6 The training stack

This section describes how an AI Specialist is made, separating what the code enforces from what the documentation only asks for, because only the former holds when whoever runs the stack is not its author (Section 7). Three facts frame it. *Scope*: the factory trains one kind of task, byte spans of k types read by a `tag` head (BIOES labels) and gated by a two-class `choice` head over the window, as in the three released examples; other task types are refused, and the other head types of the format (Section 4) run in the runtime but have no training recipe yet. *From scratch*: no released recipe uses a pretrained body, an initialized or frozen n-gram memory, or an adapter. *Provenance*: the released models were trained with an earlier, private code base, and the factory ran on the imported results: its leak checks never ran on the data of `secrets-code` and `secrets-bin`, and they reported, but kept, the blocks that the data of `pii` share with its exams (Section 6.12). On a CPU the public trainer reproduces bit for bit the weights that code base produces.

6.1 Two repositories, and what never enters git

The runtime repository, `purebyte-ai/purebyte`, holds the engine, the format specification, a NumPy reference implementation, the benchmarks and the catalog of released models (Section 5). The training repository, `purebyte-ai/purebyte-train`, holds the `pbtrain` package (the byte model in PyTorch [197], the trainer, the factory, evaluation, export and releases) with its tests; one folder per specialist, with its task files and the plugin, generator and tests it defines; the data scripts with their pinned manifests; and pinned copies of the runtime's NumPy reference and benchmark scorers, which its CI workflow compares byte for byte with the originals. The format belongs to the runtime: a new block or head type enters the runtime before the trainer. No dataset, run, checkpoint or weight is committed to either repository: data and runs live under `$PUREBYTE_DATA` and `$PUREBYTE_WORK`, outside the source tree, and the released weights are files of the runtime's GitHub releases, listed with their SHA-256 in its catalog.

6.2 The task file and the plugin contract

A specialist version is one YAML *task file*, "the whole recipe of one version of a specialist": the specialist whose plugin builds the data, the task type, an output schema (labels, recall floor, bias policy), the plugin's `data` block, the `train` block (Section 6.4), pre-registered `criteria` (Section 6.8) and the `exams` whose fingerprints must exist. Loading refuses unknown keys, options that were tried and dropped (with the reason), any task type but `tag`, a `train` block that omits `steps`, `lr`, `batch`, `size` or `seeds`, and `decide: true` without three seeds and criteria. The `data` block is *watched*: the plugin receives it in a dictionary that records every read, and the `data` stage fails on a declared key that nobody read. These rules come from an incident in which a misspelled key (`step` for `steps`) trained 400 steps instead of 2,000, with a believable result. A second file, `specialist.yaml`, holds what does not change between versions: the plugin, what the exported file declares, the ensemble policy and the exams.

Definition 7 (Vertical)**.** A *vertical* is the plugin that knows a task. From a corpus of real bytes (or none, when every byte is generated), the task's `data` block and the split fractions, it builds three partitions, *train*, *select* and *test*, each a triple (X, Y, S) of window bytes $X \in \{0, \ldots, 255\}^{n \times T}$, a span mask $S \in (\{0, \ldots, k\} \cup \{255\})^{n \times T}$ and window labels $Y \in \{0, 1\}^n$. In S, 0 is outside every span, $v \in \{1, \ldots, k\}$ is inside a span of type $v - 1$, and 255 means *ignore*: the byte gets the target -100, is left out of the loss and of every metric, and never makes a window positive. The vertical also provides guards on what it built and a *classic rival*: the rule a competent engineer would write, tuned on select and scored on test exactly like the model.

The core checks the contract (types, shapes, mask values, and Y equal to "the window holds a span") unless the vertical checks its own, as the credential vertical does: its label means "a positive line was injected", and the value may fall outside the window. A vertical without a rival stops the `baselines` stage. The scaffold `new-vertical` writes a specialist with a toy generator and no rival, so its run stops at that stage until a real rival is written: a model is never delivered without a rival to beat. The rival is a rule set by definition: no classical learned baseline (a linear model or boosted trees over character n-grams, for example) is part of the gate, and none was run on the exams of Section 8.

6.3 Data

The data scripts hold manifests, never data. Every input is public and pinned (a git commit, a package version with its SHA-256, a dataset revision), every license is recorded, and the evaluation sets, the *exams*, are fetched and fingerprinted first, because they are what every build excludes. Builds are deterministic (candidates are visited in the order of a SHA-256 key, never in directory order or through a random generator) and record every pin, the output's SHA-256 and the exclusions per exam.

Real public data. The data scripts rebuild a code corpus of 300 MiB from 114 permissively licensed repositories pinned at their heads of 2026-09-23, with no exam or CredData repository, and a binary corpus of 300 MiB from 280 pinned artifacts (Debian packages, PyPI wheels, open-source Windows builds and others), with no Windows system file. The released weights predate both: the code of `secrets-code` came from repositories cloned on 2026-09-19 and not pinned by commit, among them four CredData TEST repositories and five of the clean-code exam `code-fp-c`, the code windows of `pii` from an earlier corpus, and the binaries of `secrets-bin` from the Linux system folders of the training machine (Section 6.12). The `pii` pool, which does rebuild byte for byte, holds 387,391 labeled TRAIN documents of four public synthetic-PII datasets in 23 languages, after a check that left out 29,149 candidates sharing a text, a sentence, or a masked template or sentence with an evaluation document. The generators also read the credential catalogs of gitleaks v8.30.1 and TruffleHog v3.97.6, and public name and place lists.

Generators grounded in real bytes. The credential generator cuts a stretch of the code or binary corpus on a line boundary and inserts one generated line, positive or negative with probability 1/2, in half of the windows among up to five extra lines, all traps in a negative window. Positives include provider formats sampled from the catalogs and validated against their regular expressions (without the catalogs the vertical refuses to build), connection strings, private keys and generic secrets under credential names; negatives are look-alikes such as placeholders, environment lookups, hashes, public keys and documentation examples. The `pii` generator writes records of 10 schemas in 16 formats, log lines, and real code windows whose own personal data are replaced by synthetic values; look-alikes such as order identifiers, dates and company names are labeled *ignore*, taught neither way. In every generator an option at its default draws no random number, so a new option never changes the data of an existing recipe; golden digests of the arrays lock this, not the factory.

Labeled real windows. The `secrets-code` recipe adds 4 real windows per step from a stream of 24,000 built from CredData, whose DEV/TEST split is that of the runtime's public scorer, which the

data scripts import (seed 20260922). The stream reads only 117 of the 169 DEV repositories: openssl is left out, and the other 51 are kept for choosing operating points and for error analysis; a hard check runs before any file is opened, and a DEV value equal to a TEST value (6 credentials) is not labeled. Windows come in slots of four (two around a credential, one around a labeled non-credential, one background); in half of the credential slots the value is replaced by a generated one of the same form, with a contrast (a placeholder, a lookup, an empty value) in the same hole, and the other half keep CredData's values, which its own script has replaced by random strings (Section 8.1). The released model was trained on this stream as built, although 847 of its windows share a 64-byte block with CredData TEST files (Section 6.12). The `secrets-bin` recipe adds 7 real negatives per step from 78,283 windows of Windows executables and libraries, twelve other formats and Python extension modules; many cannot be redistributed, and their stream's builder is not in the repository.

6.3.1 Exam fingerprints and the leak checks

Leakage is easy to miss [198]: during development, exam binaries once sat inside a training corpus, an old split function put a third of a benchmark's TEST repositories in "train", and an exclusion by file name missed copies. The stack therefore compares *content*, with one algorithm that the data scripts import from the trainer.

Definition 8 (Content fingerprint)**.** Position p of a byte string x is an *anchor* when

$$\left((u_p \cdot \texttt{0x9E3779B1}) \bmod 2^{32}\right) \gg 26 = 0, \qquad u_p = \textstyle\sum_{i=0}^{3} x_{p+i}\, 2^{8i}, \tag{24}$$

which holds at one position in 64 of uniformly random data. Its *fingerprint* is a 64-bit polynomial hash of the 64-byte block that starts there,

$$h_p = \textstyle\sum_{i=0}^{63} x_{p+i}\, K^{63-i} \bmod 2^{64}, \qquad K = \texttt{0x100000001B3}, \tag{25}$$

skipped when the block has fewer than four distinct byte values. The fingerprint file of an exam holds the sorted hashes of all its files, its role (*test* or *dev*) and the SHA-256 of its manifest.

Anchors depend on content alone, as in anchored file fingerprints [199], so two copies of a region carry the same anchors wherever they sit.

Proposition 9 (Detection of a shared region)**.** *Let an input (a window, or a corpus file at build time) and an exam file share a contiguous region of $L \geq 67$ bytes whose 64-byte blocks all have at least four distinct byte values. If each position is an anchor independently with probability 2^{-6}, the check misses the region with probability at most $(63/64)^{L-67}$: below 10^{-3} for $L = 512$, a whole window, and below 10^{-6} for $L = 1{,}024$, which only a longer input can hold.*

Argument. The anchor test at p reads only $x_p, \ldots, x_{p+3}$ and the hash only $x_p, \ldots, x_{p+63}$, so at each of the $L - 63$ positions of the region whose block lies inside it, the anchor decision and the hash agree between window and file. The region is missed only if none of them is an anchor: probability $(1 - 2^{-6})^{L-63} \leq (63/64)^{L-67}$, with $(63/64)^{445} \approx 9.0 \cdot 10^{-4}$ and $(63/64)^{957} \approx 2.9 \cdot 10^{-7}$. The statement keeps the four bytes of slack that the code's documentation uses; the argument gives the exponent $L - 63$. □

The independence of anchors is an assumption: periodic or low-entropy data, such as tables and padding, can break it, and the miss rate has not been measured on real corpora.

At build time. A build stops while a source and an exam share an *identity* key (a repository at any commit, a package version, a dataset split, a URL or a SHA-256) and, unless told to record the gap, fails without current fingerprints of every declared exam. By content, text loses the lines around each shared block plus 128 bytes on each side. A binary is never spliced: it keeps one span of at most 256 KiB free of shared blocks, or nothing below 4 KiB, because dropping whole files would leave no corpus (each of 250 Linux ELF files outside the exams shared at least 51 blocks with the binary exams). The finished corpus is verified as a whole.

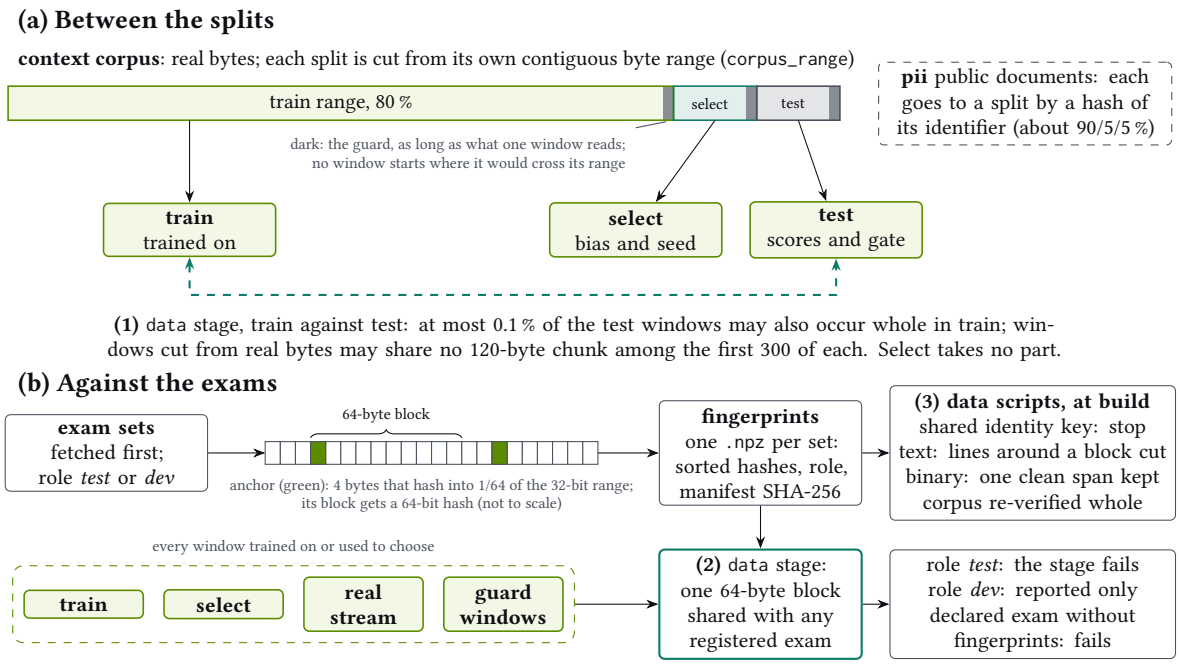


Figure 10: The leak checks. (a) The splits are disjoint by construction, and the `data` stage tests train against test. (b) One fingerprint algorithm serves twice: the data scripts keep exam content out of the corpora (3), and the `data` stage compares every window the model trains on or is selected with (2). The test split is not compared with the exams, and hits on development sets (role *dev*) are reported, not refused.

In the `data` stage. The factory repeats the check on the windows themselves (Figure 10):

(1) *Train against test.* By default, whole-window hashes, failing if more than 0.1 % of the test windows occur in train; for windows cut from real bytes (the credential vertical), 120-byte chunks of the first 300 windows of each split, failing on one shared chunk. Select takes no part, and the chunk test is a sample: the guarantee is by construction, since the splits come from disjoint byte ranges of the corpus with a guard as long as what a window reads, and public PII documents go to a split by a hash of their identifier.

(2) *Against the exams.* Every window of train, select, the real-window stream and the guard windows (Section 6.5) is compared with every registered fingerprint file. A block shared with an exam of role *test* fails the stage; one shared with an exam of role *dev* is only reported. A declared exam without fingerprints fails the stage. The test split is not compared with the exams.

(3) *The corpus.* A corpus whose build manifest records a failed guard is refused.

Not covered. A fragment shorter than 64 bytes holds no block. Exam files without a public download are excluded only if fingerprinted from a local copy. Development sets are not enforced: 194 windows of the CredData stream share a block with the DEV repositories kept for operating points, and are only reported. Two exams in the criteria of `secrets-bin` (`bin-pyext`, `bin-probe`) have no manifest in the data scripts, which therefore build no fingerprint for them. And the checks did not guard the data of the released models, which predate them (Section 6.12).

6.4 Recipes: the knobs that matter

The `train` block maps one to one onto the trainer's arguments (Table 14). *Stable* means part of a released or recommended recipe; *experimental*, off by default and, in the code's words, not (yet) shown to have "improved a released specialist on the real evaluations"; runs record the experimental options they use.

Table 14: The keys of the `train` block, their status in the code, and their values in the release recipes of `secrets-code`, `pii` and `secrets-bin`, in that order ("·": the default). All three are causal and train on 512-byte windows.

Key	Status	Default	Release recipes	Effect
`size`	stable	required	`nano`, `nano_n15`, `tiny_n18`	shape preset
`steps`; `batch`	stable	required	12,000, 12,000, 6,000; 16	steps; windows per step
`lr`	stable	required	$3 \cdot 10^{-4}$, $3 \cdot 10^{-4}$, 10^{-3}	peak of Equation (28)
`warmup`, `wd`, `clip`, `lam`	stable	30, 0.05, 1, 0.5	·	warm-up, decay, clip, λ
`seeds`; `decide`	policy	required; off	0–2, 0–5, 0–5; on	runs; decision
`released_seed`	policy	best on select	1 (forced), ·, ·	exported seed
`real_windows`, `real_frac`, `real_mode`	stable (`neg`), experimental	off	CredData stream, 0.25, `spans`; ·; binary negatives, 0.45, `neg`	real windows in each batch
`out_lr_mult`	experimental	1	·, ·, 0.25	rate of output projections
`beta2`, `span_weight`, `init_seed`, `ngram_init`, `adapter`, `guard_*`	experimental	0.95, 1, seed, off	·	β_2, span weight, init seed, starting points, tail guard
`ema`, `swa_frac`, `tern_thr`, `pooling`	refused	–	–	tried and dropped; the refusal states why

The labels lag behind the evidence twice: `real_mode: spans` and `out_lr_mult` are used by released recipes and have measured effects (Section 6.5).

The release presets are `nano` (four blocks of width 256), `nano_n15` (`nano` plus hashed n-gram tables of orders 2 to 4 with 2^{15} buckets) and `tiny_n18` (eight blocks, 2^{18} buckets). Three development results shaped them: more capacity did not help with generated labels and the n-gram memory helped most (both in Section 4.11), and on code the learning rate mattered more than a pretrained body (Section 6.5).

6.5 The training loop

Objective. Each mask becomes per-byte BIOES targets over the k types ($1 + 4k$ labels), and for a batch W the loss is

$$\mathcal{L} = \frac{1}{|\mathcal{T}|} \sum_{(w,t)\in\mathcal{T}} \mathrm{CE}\big(z^{\mathrm{tag}}_{w,t},\, y_{w,t}\big) \;+\; \lambda \frac{1}{|W|} \sum_{w\in W} \mathrm{CE}\big(z^{\mathrm{choice}}_{w},\, Y_w\big), \qquad \lambda = 0.5, \tag{26}$$

where $\mathcal{T}$ is the set of non-ignored bytes, z^{tag} the tag head's logits and z^{choice} the choice head's logits after max pooling over the window; there is no class weighting, focal term or label smoothing. Max pooling is a measured choice (Section 4.11).

Quantization-aware training. Each ternary projection of the blocks (Section 4) keeps a float latent weight W and, in every forward pass from the first step, uses

$$\widehat{W}_{o,i} = s_{o,j}\ \mathrm{clamp}\left(\mathrm{round}(W_{o,i}/s_{o,j}),\, -1,\, 1\right), \qquad s_{o,j} = \max\left(10^{-8},\, \frac{1}{g}\sum_{i'\in G_j} |W_{o,i'}|\right), \quad i \in G_j, \tag{27}$$

with G_j the j-th group of $g = 128$ inputs. The scale is recomputed at every pass and detached, not learned, and the backward pass is the identity straight-through estimator [69], $\partial\mathcal{L}/\partial W := \partial\mathcal{L}/\partial\widehat{W}$, so the optimizer updates the latent weights while every forward pass uses the codes that the exporter packs; about 31 % of Gaussian weights round to zero. There is no float warm-up, no conversion after training and no quantization of activations. The embedding, the Δ projection, the norms, the heads and the n-gram tables train in float32; the tables are the only part quantized after training (Section 6.9).

Optimization. AdamW [200] with $\beta = (0.9, 0.95)$ and decoupled weight decay 0.05 on matrices only, decided by parameter name (the A and Δ parameters, norms, convolutions, embeddings, n-gram tables and all vectors are never decayed); a global gradient-norm clip at 1.0; and a warm-up–stable–decay schedule [201] over S steps,

$$\eta_t = \eta \cdot \begin{cases} (t+1)/t_w, & t < t_w, \\ 1, & t_w \le t < t_d, \\ \max\big(0,\ (S-t)/(S-t_d)\big), & t \ge t_d, \end{cases} \qquad t_w = 30, \quad t_d = \lfloor 0.8\,S \rfloor. \tag{28}$$

Batches are drawn without replacement and reshuffled every epoch (drawing with replacement saw only 63 % of the data). Everything runs in float32: on the GPU used, half precision was not accelerated and changed the numbers. The chunked scan clamps accumulated exponents at −30; without the floor, the backward passes of two PyTorch backends diverged (gradient norms of 5.9 and 1,494 on one batch). A run returns its last step, with no early stopping.

Seeds, determinism and the control. On a given platform, PyTorch build and thread count, a run is a pure function of its data, recipe and seed: parameters are created, and random numbers drawn, in a fixed order. Seeds can train in parallel worker processes that inherit the parent's thread count, since CPU reductions depend on it, and are then bit-identical to seeds trained alone. Next to its seeds, every run trains a *control* on the first seed with the span masks, and the labels with them, permuted *between* windows: the spans then land where nothing marks them, and a control that still scores reveals a positional leak.

Real windows. With `real_windows`, round(batch · real_frac) real windows join every batch in the same forward pass, drawn from their own generators so that the generated windows of each step are those of the run without them. Mode `neg` adds real negatives with all-outside targets; `spans` adds windows labeled per span; `distill` teaches an ensemble's mean output through KL(teacher ‖ student).

Measured

- *Real negatives* (`neg`): on binaries, 44,947 real Windows windows cut false alarms on the PE files of `bin-fp-b` from 498 to 10; twelve more formats in the same quota diluted the effect (92) until the real share rose to 0.45.
- *Labeled real windows* (`spans`): the ensemble's line F1 on the complete CredData TEST half rose from 0.672 to 0.758; two earlier data changes had taken it from 0.462 to 0.585 and 0.672. Every step was scored on TEST, which thereby served as a development set (Section 8.1).
- *Learning rate, and a pretrained body, on code*: mean findings of three seeds on the clean repositories of `code-fp-b` were 107.7 from scratch at 10^{-3}, 103.0 with a body pretrained on 1.31 GB of code, 72.3 with that body at $3 \cdot 10^{-4}$, and 79.0 for a from-scratch control at $3 \cdot 10^{-4}$: the learning rate explains most of a gain first attributed to pretraining. No difference was tested, and three seeds per arm do not resolve the smaller ones (72.3 against 79.0).
- *Output projections at a quarter of the rate* (`out_lr_mult`): false alarms of six seeds of an earlier binary recipe on `bin-fp-a` (mean, worst seed) fell from 23.8 and 103 to 10.7 and 21; a 400-step warm-up (11.8, 52), clipping at 5 (22.2, 78) and $\beta_2 = 0.997$ (28.5, 64) helped less or not at all, and a zeroing threshold at a quarter of the scale ($s/4$, fewer zeros) did not help (43.5, 232; not resolved, Section 4.11). The difference in means is one tail seed: without it, the base recipe's other five seeds average about 8. At one tail run in six, six seeds without one happen a third of the time by chance, and the option was the best of five variants compared on the same exam.

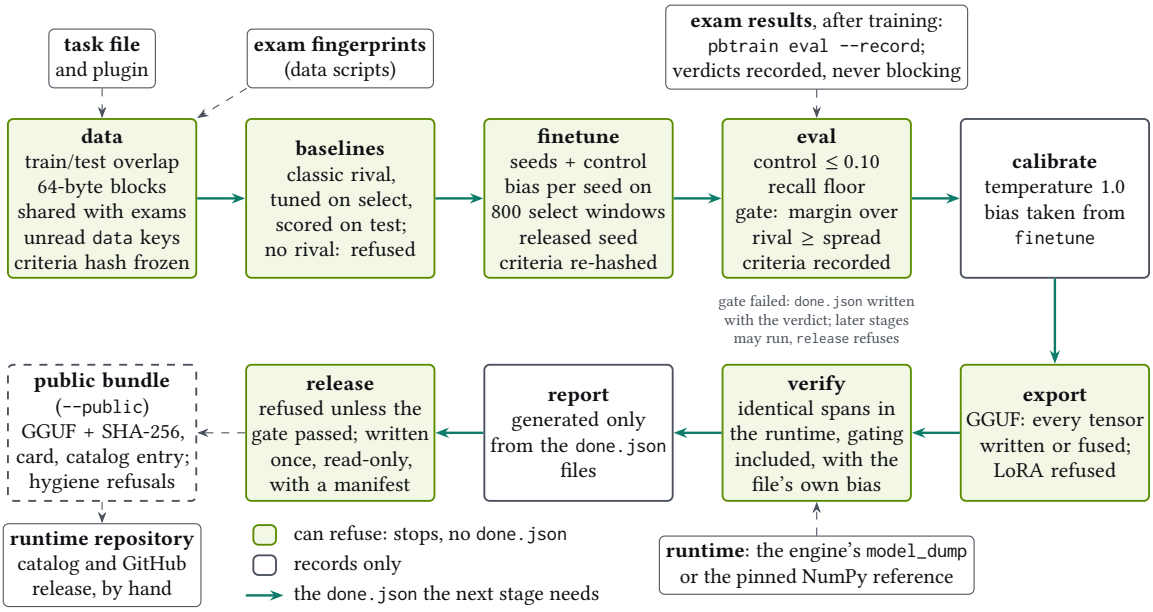


Figure 11: The factory's stage chain. Green stages can refuse: a failure stops the stage before it writes its `done.json`, so no later stage runs; gray stages only record. A failed gate is the exception: `eval` writes its verdict before it stops, the stages up to `report` may still run, and `release` refuses. Exams on real data are recorded after training and judged against the criteria of the task file; their verdicts do not block.

Experimental machinery. About one binary run in six ended with 50 to 230 false alarms on `bin-fp-a`, and fixing the initialization while varying the data order showed that the variance lives in the data order [202]. The *tail guard* was built for it: every few steps the model reads fixed real negative windows and a proxy counts those it flags, to keep the best late point or roll back. The proxy flagged none of its windows even on a bad seed, whose false alarms came from one unusual file; the code's note says the guard "is expected to be blind", and it stays off. Also off in every release: distillation (no script in either repository builds a distillation set), a pretrained body (it helped on binaries in development, false alarms on `bin-fp-a` of 2, 6 and 27 for three seeds against 43, 188 and 172 from scratch, a difference we attribute to data scale, but not on code; the training repository has no pretraining script), and adapters.

6.6 The factory: what each stage enforces

A task runs through nine stages (Figure 11). Each writes a `done.json` with what it read and produced, the hashes of its inputs and outputs and the task file's hash, and refuses to start while the previous one is missing. Table 15 lists every check in the code; an expected refusal ends with a one-line message and exit status 2, not a stack trace.

The chain checks that the previous `done.json` exists, not that the task is unchanged: an edited task shows as a different hash, and only the criteria are compared again (at `finetune` and `eval`). Several documented rules are not code: "three seeds inform, six decide" is a checklist item, and a FAIL of a pre-registered criterion is recorded and shown, while the code blocks a release only on the gate. The factory therefore blocks leaks in the data it builds, silent options and unverified exports; it records, but does not block, failed criteria, choices made on an exam and departures from written rules, and each of the three releases took at least one of them (Section 6.12).

Table 15: What the code checks at each stage. *Stop*: the stage fails and writes no `done.json`. *Record*: the result is written and nothing is blocked.

Stage	What the code checks	On failure
`data`	contract and guards; unread keys; corpus guard; train against test	stop
	train, select, real and guard windows against every registered exam	stop (*dev* exam: record)
`baselines`	a rival exists, tuned on select and scored on test	stop
`finetune`	criteria unchanged; per seed, a bias on select, off the grid's edge	stop
`eval`	control ≤ 0.10; the released seed's recall on select meets the floor	stop
	the gate (Equation (30))	verdict written, then stop
	synthetic criteria; "DECIDES" with `decide` and ≥ 3 seeds	record
`calibrate`, `report`	temperature 1.0 and the bias; a report from the `done.json` files only	record
`export`	every tensor written or fused; no LoRA	stop
`verify`	identical spans, gating included, on the first 40 test windows (default)	stop
`release`	the gate passed; the version is new	stop
public bundle	no internal string in card or metadata; matching versions	stop, bundle deleted
none	six seeds to decide; no FAIL or PENDING criterion; choices never on an exam	written rules only

6.7 Selection and calibration on held-out data

For per-byte log-probabilities ℓ_t of the tag head, the decoder (Section 4) returns

$$\hat{y} = \arg\max_{y \in \mathcal{V}_T} \sum_{t=1}^{T} \left(\ell_{t,y_t} + b_{y_t}\right), \qquad b_{\mathrm{O}} = 0, \quad b_{\mathrm{B}\text{-}e} = b_{\mathrm{I}\text{-}e} = b_{\mathrm{E}\text{-}e} = b_{\mathrm{S}\text{-}e} = \beta + \delta_e, \tag{29}$$

with $\mathcal{V}_T$ the valid BIOES sequences of length T; a larger *operating bias* β trades precision for recall without retraining. For each seed, `finetune` scores the gated span metrics on the first 800 select windows at 20 values of β from -6 to 6 (in steps of 0.5 from -4 to 3, and of 1 outside) and applies the task's policy: `conservative` takes the most precise bias whose recall reaches `min_select_recall`, `f1` the best F1 among those. If none reaches the floor, the best F1 is taken with a warning and the floor in `eval` decides; an optimum on the edge of the grid is an error. Typed models can add per-type offsets δ_e. The bias travels in the model file, where callers can override it. The release recipes use a floor of 0.60 and the `conservative` policy (`secrets-code`, `secrets-bin`: bias 0.0) or `f1` (`pii`: bias -0.5). The released seed is the best on select unless the task forces one, which the report then marks: `secrets-code` 1.0.0 forced its seed 1, recorded in the recipe as "the fewest findings of the three on the clean-repository exam code-fp-b, a CredData TEST F1 within 0.01 of the ensemble's", while seed 2 was best on select: a selection on an exam, made with the three seeds' CredData TEST scores already known (Section 8.1).

The `calibrate` stage fits nothing: the choice head keeps temperature 1.0 because its expected calibration error (ECE) [203] on the generated test windows is below what their number resolves (with 2,000 windows and saturated confidences, a perfectly calibrated model has a median ECE of about 0.001), which says little about real inputs. Reported confidences are documented as uncalibrated: a logistic model fitted on DEV ranked findings better (AUC 0.64 against 0.59, without intervals) but had an ECE of 0.108 on CredData TEST, one more use of TEST, and was not shipped.

6.8 Evaluation: control, rival and gate

Metrics decode as the engine does (float64 Viterbi, lowest label index on ties) and compare spans as sets with exact boundaries; the reading that counts is the *gated* one, with spans only in windows the choice head calls positive. "A specialist that does not beat the classic rival of its task is not delivered": the rival is tuned on select and scored on test like the model (for credentials, an entropy threshold over quoted literals; for personal data, patterns with checksums, context words and name lists). With $\bar{F}$ the mean

gated test F1 of the seeds and σ its population standard deviation, the gate passes when

$$\bar{F} - F_{\text{rival}} \;>\; \max\left(\sigma,\; 10^{-9}\right), \tag{30}$$

that is, when the advantage over the rival is more than the spread between seeds. The gate is a sanity check, not a significance test: with three seeds, an advantage of one population standard deviation is about 1.4 standard errors of the mean, and the rival is scored on the generator's own test windows. The control's gated F1, read at the more favorable of the biases 0 and +3 so that a leak cannot hide behind the bias, must be at most 0.10. On their synthetic tests, `secrets-code` scored a mean span-F1 of 0.891 over three seeds (0.880 to 0.898) against 0.330 for its rival, 72.6 spreads ahead; `secrets-bin` 0.890 over six seeds (0.874 to 0.901) against 0.330 for the same entropy rule, 51.0 spreads ahead; and `pii` 0.824 over six seeds (0.816 to 0.830) against 0.444 for its rules, 73.7 spreads ahead. All three controls scored 0.000.

A task that decides carries `criteria` written before training [204]: a date, a prediction, and thresholds on synthetic metrics (checked by `eval`) and on exam metrics, checked when `pbtrain eval` records the results of exams run with their own tools. Each criterion is PASS, FAIL or PENDING ("a FAIL is a result, not a bug"), and criteria on seed tails read the worst seed. The date is the author's record, not an external time stamp. The first `data` stage of a version freezes its criteria in a file that is never rewritten: a later `data` stage with other criteria under the same version is refused before it touches anything, and the report and the release manifest say when the criteria were frozen. The conditions under which an exam metric is read (operating point, exclusions, ensemble configuration, scanner version) are not in the frozen block, and the released recipes predate this check. Exam metrics enter as a file of values, and the criterion of `pii` was moved to another operating point without a change to its thresholds (Section 8.3).

Measured

Synthetic scores are not the result, and every run's report says that they "do NOT predict real-world performance". Section 8.5 lists seven cases from the development of the three recipes in which a synthetic or development signal and the real exam disagreed.

6.9 Export and verification

The exporter writes one GGUF file [191], in the lowest format version that can express it, and enforces *coverage*: every tensor of the body and the heads is written or fused, or the export fails, as it does for a LoRA adapter, which the format cannot express. Ternary weights become 2-bit codes with one float32 scale per group of 128, 2.25 bits per weight; the n-gram tables are stored at 4 bits with one symmetric scale per row, which shrank the `secrets-bin` file from 193.8 to 28.8 MiB. The exporter never invents behavior: profile, window, labels and operating point come from `specialist.yaml` and the calibration.

The `verify` stage runs the exported file in the runtime and requires the *same spans* as the PyTorch checkpoint, window by window, gating included, at the file's own operating point; comparing probabilities would not be enough, since a different tie-break in Viterbi changes the spans while barely moving the numbers. The runtime is the C++ engine's `model_dump` tool when available, otherwise the pinned NumPy reference, on the first 40 test windows by default. For 4-bit tables, the Python side quantizes its tables as the exporter does and counts apart the windows changed by quantization alone: about 1 % for `secrets-bin` (22 of 2,000 test windows). Before release, `verify` found the same spans on 2,000 of 2,000 test windows for `secrets-code` and `secrets-bin`, and on 40 of 40, the default, for `pii`, a sample too small to rule out a mismatch rate of several percent; for `pii`, 4-bit storage alone changed the spans of 2 of those 40 windows. Tests also export ten layouts and require the reference to match PyTorch's hidden states within 10^{-4} (relative), log-probabilities within 10^{-3} and spans exactly; the CI workflow builds `model_dump` so that the C++ engine is compared too.

6.10 Releases

The `release` stage refuses when the gate did not pass and when the version exists: a version is written once, read-only, with the GGUF of the released seed and of the other ensemble members, the report, the

Table 16: The three release recipes on one AMD Radeon RX 6800 XT (16 GB) under Windows 10, with nightly ROCm builds of PyTorch, in float32. `secrets-code` was timed with the fused kernels of the Radeon backend. The fine-tuning stage is training plus the bias sweep and the checks.

	`secrets-code`	`pii`	`secrets-bin`
Preset; parameters without heads	`nano`; 1,927,520	`nano_n15`; 8,268,128	`tiny_n18`; 54,170,048
Model file	1,028,704 B	4,764,864 B	30,163,488 B
Generated training windows	80,000	192,000	40,000
Steps × windows per step	12,000 × (16 + 4 real)	12,000 × 16	6,000 × (16 + 7 real)
Bytes read per run	123 M	98 M	71 M
Seeds + controls; jobs at a time	3 + 1; 4	6 + 2; 1	6 + 1; 2
Training	1,468 s	432–461 s per seed	3,459 s
Fine-tuning stage	1,590 s	1,802 s + 1,783 s	3,665 s

task file, every `done.json`, and a manifest with every SHA-256, the code commit (marked `+dirty` when the training code had local changes), the seeds and the machine. The *public bundle* adds public names, one `.sha256` per file, a model card and a catalog fragment, and refuses, deleting what it wrote, when the card or the metadata carry an internal string (a drive path, a home directory, a private pattern) or another version than the bundle's. The card's rules are written, not checked: every number with its conditions [205], "TBD" for what was not measured, and what the model does not detect. Publication happens in the runtime repository, by hand, and `purebyte models pull` must verify the files before a release is announced. `secrets-code` ships its released seed and two more as an optional two-of-three ensemble; `secrets-bin` ("a jury of three did not generalize out of sample") and `pii` ship one model each.

6.11 Hardware and cost

The released examples were trained on one workstation: an AMD Ryzen 9 5900X (12 cores), 32 GB of RAM and an AMD Radeon RX 6800 XT (16 GB, RDNA2, a consumer card from 2020), under Windows 10 with nightly ROCm builds of PyTorch, in float32 (Table 16). The three recipes took about 2.5 GPU-hours together, including their bias sweeps and checks, and each data stage one to two minutes; the exams on real data ran on the CPU and took longer (1 h 47 min for those of `secrets-code`, 4 h 9 min for the binary exams). Across the project, logged fine-tuning runs add up to at least 37.5 GPU-hours on that card, and the release recipes are the survivors of about forty factory versions.

The stock step of these small models is dominated by kernel launches, so the stack carries a `radeon` backend of fused HIP kernels for the RX 6800 XT: on `nano` with batches of 16 windows of 512 bytes, a step fell from 74.0 to 32.0 ms in one process and to 22.6 ms in aggregate over four. The GPU still runs at about 17 % of its peak, and a rented RTX 5090 with stock PyTorch was only 1.86 to 1.88 times faster than the RX 6800 XT with those kernels, for five times the peak throughput. At that batch shape a `nano` job needs 1.8 GB of VRAM and a `tiny_n18` job 3.9 GB.

Training takes `--device cpu|cuda|mps|auto`: the CPU is the default and the reference, `cuda` an NVIDIA (CUDA) or AMD (ROCm) GPU, `mps` Apple silicon, and `auto` the first found. A requested GPU that is missing is an error, never a silent CPU run, and a lock admits one GPU job at a time. The Radeon backend is deterministic from run to run, but not bit-identical to the CPU (the loss agrees to seven digits, the worst gradient to $1.3 \cdot 10^{-5}$). The device-parity test (opt-in, not run in CI) trains a `nano` model for three AdamW steps on four random 512-byte windows on the CPU and on every GPU found, and requires the losses to agree within 10^{-3} relative: it shows that a device computes the same training step, not that a full recipe on it lands where the Radeon runs did. The test status on 2026-09-25: the CPU is the reference, and the CI workflow runs the training tests on it under Linux, Windows and macOS; an RX 6800 XT under ROCm on Windows trained the released examples and passed the device-parity test that day; AMD GPUs under Linux (ROCm), NVIDIA GPUs (CUDA) and Apple silicon (MPS) use the same stock PyTorch code in float32 and have not passed the device-parity test; an RTX 5090 was used

only to time training steps.

6.12 Reproducibility: guarantees and gaps

On a CPU, the public trainer reproduces bit for bit the weights of the earlier code base that trained the released models: 27 golden cases cover every trainer path, each the SHA-256 of every weight after 20 steps of a two-block model of width 64 on one thread, and four more cover the `pii` paths. They were recorded with PyTorch 2.4.1 (CPU), NumPy 2.2.1 and Windows on AMD64 and are compared only there; the CI workflow skips them because CPU kernels can differ between processors of one platform (AVX2 or AVX-512), so they are verifiable on the recording machine, while a determinism test (the same run twice, the same bits) runs everywhere. The credential generator matches bit for bit the arrays of the generator the released models were trained with (on a generated test corpus), the PII pool rebuilds byte for byte, the CredData download gave byte for byte the 11,564 files of the copy the released stream was built from (the labeled files, 11,030 in the complete dataset, and the license files that CredData copies with them), plus one labeled file that an antivirus had removed from that copy, and releases record the code commit and every digest.

Limits of the claim

- **The released weights are not reproducible bit for bit.** They were trained on a GPU, which rounds differently from the CPU reference, with an earlier, private code base of which only the fingerprints are public; the public trainer matches that code on a CPU, verifiably only on the recording machine. A retraining is a new draw from the distribution of the recipe's seeds: expect results within their spread (on CredData, 0.788 to 0.799 over three seeds; on clean binaries, Section 8.4), not the released figures. No retraining with the public code has yet been scored on the exams.
- **Earlier corpora.** The code corpus of the released `secrets-code` predates the data scripts: cloned on 2026-09-19 and not pinned by commit, it held four CredData TEST repositories, which its published CredData figures exclude, and five repositories of the clean-code exam `code-fp-c` (Section 8.2). The code windows of `pii`, 4 % of its training windows, also come from an earlier corpus, and the binaries of `secrets-bin` from the Linux system folders of the training machine. None of them rebuilds byte for byte.
- **A missing builder.** No script in either repository builds the negative windows of `secrets-bin`.
- **A Python version.** The CredData stream's arrays are bit for bit the released ones only with Python 3.14, because its generator validates provider values with catalog rules that compile differently across versions (99 of the 120 sampled rules on 3.14, 96 on 3.12, 116 on 3.10).
- **Release data that today's factory refuses.** The released `secrets-code` was trained on the full CredData stream, 847 of whose 24,000 windows (392 with a credential) share a 64-byte block with files of 136 of the 168 CredData TEST repositories, and the released `pii` on a pool of which about 3 % of the documents share one with the PII Masking Benchmark or `pii-bench-test` (the factory's held-out sample of 4,122 documents from the test and validation splits of three of its source datasets). The runs of `secrets-code` and `secrets-bin` were imported after training, so the factory's leak check never ran on their data; that of `pii` reported the shared blocks and kept them, as trained. The `data` stage now refuses both data sets, so neither release recipe runs end to end on its release data; `--exam-overlap drop` builds data without them (23,153 stream windows when only CredData TEST is fingerprinted), which the factory accepts and which is not the release data. The measured effects on the published scores are small: without the labeled TEST lines inside shared blocks (12 positive, 122 negative), the CredData F1 falls by 0.00001, and the benchmark's mean F2 is 0.7686 instead

of 0.7689 without the 1,018 sentences that share a block, a count that cannot see shared templates (Section 8.3).

- **Written rules the releases bent.** `secrets-code` was decided on three seeds, its released seed was chosen on an exam while its CredData TEST scores were known, and two of its pre-registered criteria failed; the probe-recall criterion of `secrets-bin` failed for two of six seeds; the criterion of `pii` passes only at the operating point it was moved to after it was written, and its data mix ships against the pre-registered tie rule (Section 8.3). All were released, which the code allows.
- **One workstation**, with one GPU, produced every training measurement except the RTX 5090 step timing.

7 Vibe Training: an AI that trains small AIs

Definition 10 (Vibe Training)**.** Vibe Training is training an AI Specialist by describing the job to an AI coding agent, the way vibe coding builds software. The agent turns the description into a task file, data, success criteria written before training, trained seeds, examinations on real data and a released model file. A factory makes those steps checkable by the person who asked: it blocks leaks between training data and examinations, options that change a result silently and exports that the runtime does not reproduce; it records, but does not block, a failed criterion, a choice made with an examination in view and an override of a pre-registered rule (Section 6). Each of our three releases took at least one of those recorded exceptions (Section 8).

A request can be one sentence: *"Train an AI Specialist that flags the customer names and addresses in our support-ticket CSV exports, and tell me honestly how good it is."* What follows that sentence is ordinary machine-learning engineering (Figure 12): write or adapt a data generator grounded in real bytes, fetch public data and fingerprint the examination sets first, write the criteria, train three to six seeds and a control, compare with a classic rule, choose an operating point on held-out data, verify that the runtime gives the same answers as PyTorch, and write the model card. None of it is new. What is new is who does it, how cheaply, and what the result costs to run. This section argues that for byte-level jobs that meet the conditions of Hypothesis 11, an agent that trains a specialist is a better use of an agent than one that writes rules for the job or one that reasons about every input at run time. We state the claim as a hypothesis with its conditions, give a cost model with explicit assumptions, and separate what we measured from what we did not.

7.1 Ways an agent can do a byte-level job

Take a job whose input is bytes and whose answer is a decision, a score or a set of byte ranges: is there a credential in this file, which bytes of this ticket are personal data, is this log line anomalous, is this blob a known format. An AI agent asked to automate it has three basic options.

(a) **Write a program**: rules, regular expressions, parsers. It is cheap to run and easy to audit, and it is exact where the pattern has a fixed form. It misses what has no fixed form (a password in a variable called `p`, a surname without a title), and each new miss costs another rule.

(b) **Ask a general model at run time**: send every input, with the job's description, to a large model, hosted or local. It needs no training and adapts to a new question at once. Every input then pays a large model's compute, and, if the model is hosted, a network round trip and a copy of the data at the provider.

(c) **Train a specialist** (Vibe Training): the agent spends its effort once, on data and evaluation; the result is a model file of 1 to 29 MiB that answers a short input in about a millisecond with

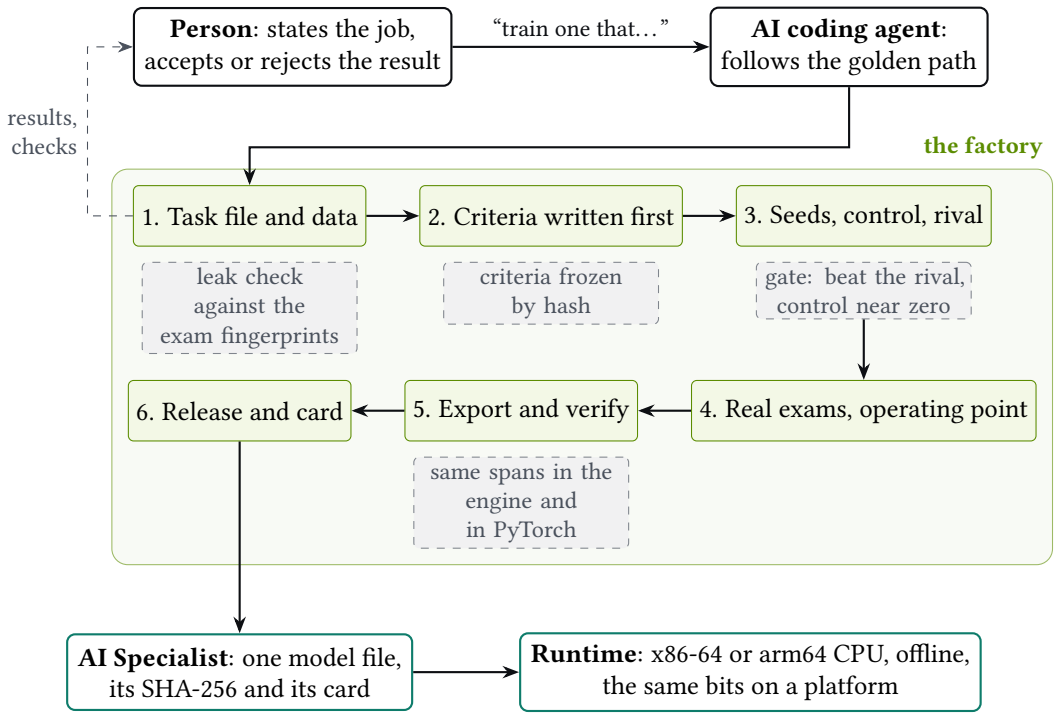


Figure 12: The Vibe Training loop. A person states the job and accepts or rejects the result; the agent does the engineering through the golden path of the training repository; the factory's checks (gray) stop leaks, silent options, changed criteria and runtime mismatches with a message that says why. The output is a file that the runtime executes on ordinary CPUs, with no agent in the loop.

eight threads of a desktop CPU (3 to 6 ms on one core), offline, with the same bits every time on a platform. It answers only the question it was trained for, and needs data that describe that question.

Three more options combine these. (d) *Extract candidates with rules and verify them*, with a learned classifier or a large model that sees only the candidates: the state of the art in secret scanning, where a machine-learning filter or an 8-billion-parameter model classifies the candidates of regular expressions [148, 152]; it divides the large model's calls by the ratio of candidates to inputs. (e) *Fine-tune a small pretrained encoder*, as the leaders of the PII benchmark did (Section 3.3). (f) *Specialize a small local generalist* with a small adapter per job, as Program-as-Weights does with a 0.6-billion-parameter model [124] and as on-device foundation models do with low-rank adapters [206]. Option (c) keeps its advantage where there are no candidates to extract (a value defined by its context, a credential inside machine code), where no pretrained model fits the device, where an audit needs the same bits every time, and in cost per byte on a CPU; (d) and (e) are its closest competitors, and we have measured neither against it, except (e) through the PII benchmark's leaderboard.

Table 17 compares (a), (b) and (c) on what we can measure with the released runtime. It is a choice of where to spend an agent's intelligence: once, at development time, or again on every input.

7.2 The hypothesis and a cost model

Hypothesis 11 (Specialists pay off)**.** Let a job map byte strings to a decision, a score or byte ranges, where (i) the answer depends on local patterns that a model can learn from examples, (ii) examples can be collected or generated, with a held-out real examination set, (iii) the question is stable for the life of the deployment, and (iv) the job runs many times. Then a specialist trained by an agent costs less in total than asking a general model at run time beyond a number of decisions N^* that the model below estimates under stated assumptions, provided its errors cost no more than the general model's; and it costs less than rules written for the job beyond a number of decisions that falls as a miss costs more,

once a missed positive costs more than about ten to twelve seconds of a person's time (on CredData as distributed and on its original values, with a false alarm costing a minute), because on the patterns without a fixed form it makes fewer errors than a mature rule set.

The four conditions are where the hypothesis stops. Without (i), there is nothing to learn from bytes: questions that need knowledge of the world ("is this company sanctioned?") belong to a large model or a database. Without (ii), the specialist cannot be trained or, worse, cannot be examined. Without (iii), the specialist must be retrained as the question moves, which the cost of maintenance below has to carry. Without (iv), the one-time cost is never repaid: a question asked ten times should be asked of a large model.

A cost model. Let each approach s have a one-time cost D_s to build it, a cost M_s to maintain it over the deployment (retraining, rule updates), a marginal cost c_s per decision, and an error rate per decision: a fraction m_s of decisions that miss a positive and f_s that raise a false alarm, which cost C_{miss} and C_{fa} each (for example, minutes of a person's triage). Over N decisions its total is

$$T_s(N) = D_s + M_s + N\,(c_s + e_s), \qquad e_s = m_s C_{\text{miss}} + f_s C_{\text{fa}}, \tag{31}$$

and a specialist ($s = \text{spec}$) beats an alternative $s = \text{alt}$ once

$$N > N^* = \frac{(D_{\text{spec}} + M_{\text{spec}}) - (D_{\text{alt}} + M_{\text{alt}})}{(c_{\text{alt}} + e_{\text{alt}}) - (c_{\text{spec}} + e_{\text{spec}})}. \tag{32}$$

In the estimates below we set M_s aside, which favors the specialist in proportion to how often it must be retrained.

Against run-time reasoning. We did not measure a general model's errors on our examinations, so we can only state the crossover when the error terms are equal, which is the hypothesis's proviso. In units of waiting time, with the measured $c_{\text{spec}} = 0.82$ ms for a short decision (on eight threads, so 3 to 6.5 milliseconds of CPU cores) and $c_{\text{llm}} = 236$ ms, the hosted decision API quoted above, a specialist whose building took a full day of an agent's and a person's time ($D_{\text{spec}} = 86{,}400$ s, far more than the 25 minutes of GPU time it needs) is repaid after $N^* \approx 367{,}000$ decisions: about an hour of a service that decides a hundred times per second. Figure 13 draws $T_s(N)$ over the ranges of Section C: the crossover lies between about 7×10^4 and 4×10^5 decisions against the quoted hosted API, and between about 8×10^3 and 4.5×10^4 in the separate, assumed scenario of a model that reasons for 2 s. In money the crossover moves by orders of magnitude: with an assumed 500 to 1,000 dollars for a person's day and the agent's tokens, and an assumed 10^{-5} to 10^{-3} dollars per short hosted call, N^* lies between 5×10^5 and 10^8 decisions. And the agent's own work is itself run-time reasoning: if a day of agentic engineering uses 10^7 tokens and a run-time decision 10^3, the build alone equals 10^4 decisions of option (b). The waiting-time numbers illustrate Equation (32); they are not a measurement of Vibe Training, since we did not measure how long an agent takes to build a specialist (Section 7.5).

Against rules. Here the build and the run favor rules at every N (Figure 13), and the case for the specialist rests on the error terms, which we measured on CredData's labeled lines. On its test half without the four repositories that overlap the training corpus (164 of 168; `secrets-code` learned partly from labeled lines of the other half), about 9.5×10^5 windows of 384 new bytes of code, `secrets-code` finds 3,077 labeled credential lines with 422 false alarms, and gitleaks with its default rules finds 875 with 92 (Section 8.2): 2,202 more hits for 330 more false alarms. With a false alarm costing a minute of triage, the specialist makes fewer costly errors as soon as a missed credential costs more than about ten seconds of a person's time, and from there the crossover of Equation (32) is finite: between 1.1×10^5 and 7.4×10^5 windows (40 to 270 MiB of code) if a miss costs a minute, between 9×10^3 and 6.4×10^4 (3 to 23 MiB) if it costs ten minutes, and a few megabytes if it costs an hour (Figure 14). A leaked credential

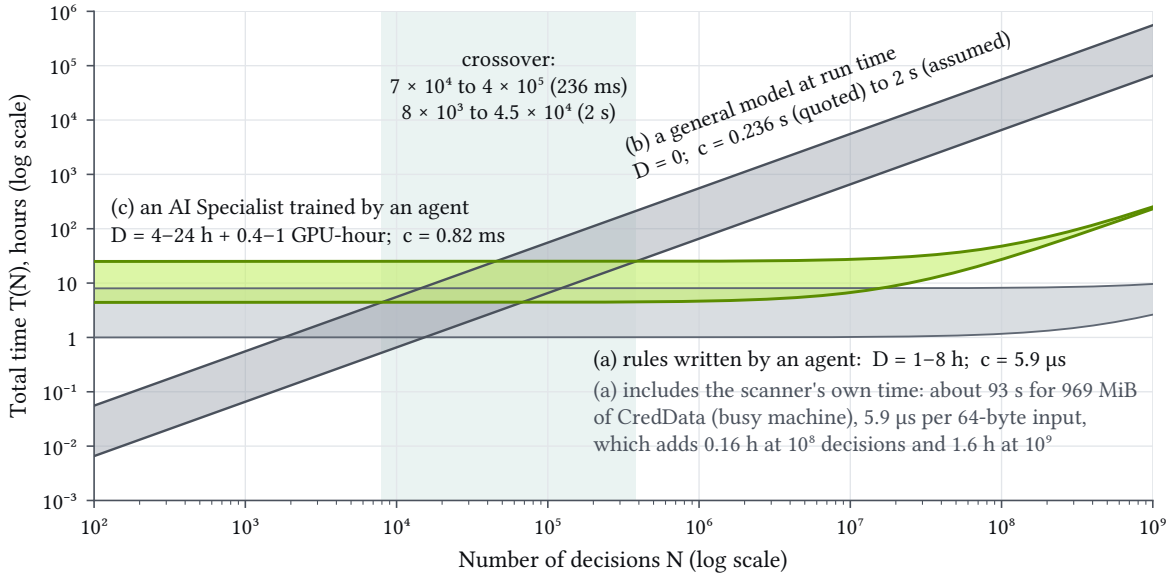


Figure 13: The cost of computing N decisions (Equation (31) with $e_s = M_s = 0$), in hours of waiting time, over the parameter ranges of Section C. The specialist's per-decision cost is measured; the general model's is quoted (236 ms, a hosted decision API) or assumed (2 s, a model that reasons before answering); the build costs are assumptions. In this view rules cost less than the specialist at every N, and the specialist's line rises past 10^7 decisions; against rules, the specialist pays off only through the errors it avoids (Figure 14).

usually costs far more than an hour. These rates come from one benchmark as distributed, against one mature rule set. On its original values, which flatter the specialist less (Section 8.2), it finds 1,610 more lines for 321 more false alarms (2,460 and 413 against 850 and 92), which moves the threshold from about ten to about twelve seconds. In both, false alarms are counted on labeled lines only, for both tools, so the thresholds leave out flags on unlabeled lines (on 36 clean repositories the specialist reports 639 findings in 10,820 files, a rate not measured for gitleaks).

7.3 Why a trained model, and not a program the agent writes

An agent that writes rules produces what a careful engineer would: patterns for the known formats, keywords for the rest. We expect such rules to plateau on patterns without a fixed form, but we did not measure rules written by an agent against the effort spent on them; the rival we measured on real data is gitleaks, a mature collection of community rules, not rules of comparable cost (against those we compared only on generated windows, Section 6). On CredData, gitleaks has the higher precision (0.905 against 0.879) and a recall of 0.207; the released `secrets-code` model finds three and a half times as many labeled credential lines (3,077 against 875; recall 0.729) on the 164 of CredData's 168 test repositories that do not overlap its training corpus (3.1 times on all 168, 2.9 times on the original values), after learning partly from the benchmark's other half, and with the caveats of Section 8.2. Most of the difference sits where the form is not fixed: passwords and tokens identified by their context. Rules come close only for private keys (recall 0.84 against 0.90); on provider patterns, a fixed form, the model also leads (0.85 against 0.28), a gap we have not explained. Rules keep far more speed per byte. For personal data, a pipeline of rules and a named-entity recognizer (Presidio with spaCy's large English model) scores F2 0.586 on the PII Masking Benchmark (its leaderboard entry), and the 8.3-million-parameter `pii` specialist 0.769; Presidio's recognizers are English and typed, while the benchmark counts every labeled character as one type in 23 languages, so the comparison favors a model trained on the benchmark's own generators.

Vibe Training does not remove rules; it moves them into the data. The generators of the credential specialists read the rule catalogs of gitleaks and trufflehog to produce their positives (Section 6), and the model then learns what the rules cannot express from the context around those values. The agent's

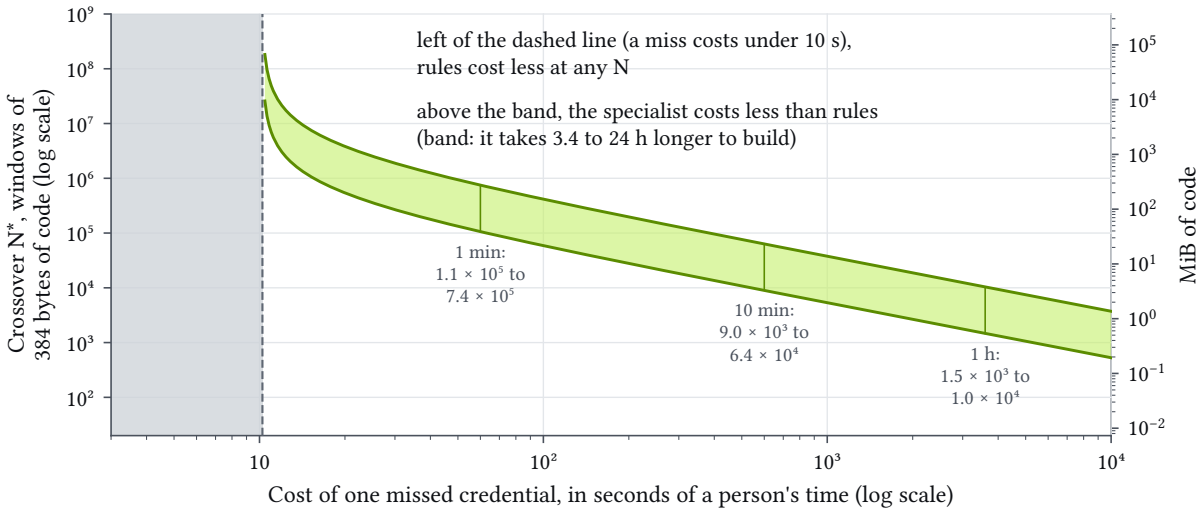


Figure 14: Against rules, the specialist pays off through the errors it avoids: the crossover N^* of Equation (32) between `secrets-code` and gitleaks with its default rules, in windows of 384 bytes of code, as a function of the cost of one missed credential. Error rates per window measured on the published scope of CredData's test half; a false alarm assumed to cost 60 s of triage; the band spans a specialist that takes 3.4 to 24 hours longer to build than the rules; costs per window of 2.9 ms (the specialist on the whole desktop) and about 0.04 ms (gitleaks). Left of the dashed line, rules cost less at any N.

effort moves from writing exceptions to writing data and examinations, which is where an agent is already strong: code for generators, scripts for data, tests.

7.4 Why not reason at run time

A large model can often answer these questions well; the argument against asking it on every input is cost, latency, privacy and determinism, and against a local model only the first two remain.

- **Latency and throughput.** 2.81 ms with eight threads of a desktop CPU for a 256-byte window that holds Laya's test line (0.82 ms for a 64-byte input, 3.0 ms on one core), against 373 ms for Laya with twelve threads of the same CPU, and, as quoted by Laya's model card, 33–40 ms for Laya on a data-center GPU and 236–276 ms for a hosted decision API (Section 5); the specialist scans a code base at 131 kB/s on the whole desktop.
- **Privacy.** A secret scanner or a personal-data filter that sends the data to a remote model has moved the problem instead of solving it. A specialist, like any local model, runs in the process that holds the data.
- **Determinism and audit.** The runtime gives the same bits for the same input on every kernel and at any thread count of a platform, and decisions that are expected to agree across platforms (Section 5.2); a model file has a SHA-256, so a finding can be rerun with the same file.
- **Cost at scale.** Condition (iv) of Hypothesis 11: pre-commit hooks, log pipelines and data exports make millions of decisions.
- **Where it runs.** Megabytes of weights and of memory reach machines where no large model runs (Section 5.8.1).

Large models remain the right tool for questions that change every time, need knowledge of the world, or are asked a few times. They are also the natural *authors* of specialists: Vibe Training puts the large model where its generality pays, at development time.

7.5 This project as a case, and what we did not measure

The runtime, the training stack, the three example specialists and this paper were developed by the author working with an AI coding agent, which wrote most of the code, generators and documentation, ran the training and evaluation campaigns (about forty versions of the factory's tasks) and reported the results; the author set the goals, the criteria and the decisions to release. This is the author's account, not a controlled measurement, and it is the one data point we have on the cost of building: at least 37.5 GPU-hours, generators of 2,354 and 2,662 lines, and real examinations that took longer to build than the models (Sections 6 and 8), well above the 4 to 24 hours that Figure 13 assumes for a new specialist once the stack exists. The account still shows the loop working, and its checks catching real problems:

- the `data` stage, added after the release was trained, refuses the real-window stream that the released `secrets-code` learned from, because 847 of its windows share 64-byte blocks with files of CredData's test half; future versions train on the stream rebuilt without them, and the measured effect on the released model's score is a change of −0.00001 in F1 (Section 8.2);
- seed-to-seed variance exceeded most of the architecture effects we measured (Section 4.11), which is why three seeds inform and six decide;
- after the release, a document memory that looked valuable on constructed documents was measured on the real examination set and *withdrawn*: copying the model's decisions across a file copied its errors (Section 9.4).

It also shows the limits. Vibe Training still writes code, and "most of the work of a new specialist is its data". Several of the factory's rules are recorded rather than enforced (Definition 10), and our own releases took those exceptions. And we have not measured the agent itself: how long a new specialist takes, how many attempts, how often a person must intervene. That measurement, a benchmark of Vibe Training tasks with success rates and costs in the spirit of benchmarks of agents doing machine-learning engineering [114, 116], is the next step.

7.6 Train-yourself, and small models as the tools of agents

Thesis. Train-yourself

We conjecture that agents will increasingly meet repetitive, high-volume, byte-level questions inside larger tasks: triaging logs, scanning uploads, filtering data exports, checking configurations. Instead of answering each instance, an agent can train a specialist for the question, check it against a held-out examination with criteria written beforehand, and then call it as a tool: fast, private, deterministic and cheap, on the machine where the data are. The large model does the thinking once; the small model does the work every time. Each retraining needs labeled data and a fresh or reusable examination [208], whose cost the model of Equation (31) carries only in M_s.

This view extends the argument that small language models suit many of the repetitive calls inside agentic systems [3], and the practice of letting models call tools [135, 136], in one direction: the small model need not exist before the agent needs it, because the agent can make it. It covers the calls that decide, score or locate; an AI Specialist cannot stand in for the calls that must generate text or structured output, such as a tool call written in JSON. And it stops where the evidence stops: we have shown the pieces (the factory, the checks, three specialists and their measurements), not an agent that trains specialists unattended in production.

Limits of the claim

Hypothesis 11 is a position, argued with measurements of the specialists and with a cost model whose build costs and prices are assumptions. We did not measure agents building specialists, nor large models answering these questions on our examinations, so the crossover against run-time

reasoning assumes equal error costs. The crossover against rules rests on the error rates of one benchmark against one rule set. The specialists answer only the question they were trained for, from the bytes they see; they need data and an examination set, and a person must still define the job and accept the result.

8 Case studies: three AI Specialists

The release ships three AI Specialists: `secrets-code` finds credentials in source code and configuration, `pii` masks personal data in any text, and `secrets-bin` finds credentials inside compiled binaries. They are examples of one architecture (Section 4) and one factory (Section 6) on three kinds of bytes (Table 18). This section reports how they were measured, what they score under which conditions, where they fail, and what they teach together. Results come from the release's benchmark report and model cards, and development measurements from the records behind Section 6.

8.1 How we measure

The three specialists were measured under the same seven rules, stated in the release before its figures.

(1) **Public data, where it exists**: CredData [150] at commit `c09c0c5` (2026-09-06; 337 repositories, 11,030 labeled files, obfuscated by the official scripts) for `secrets-code`, and the PII Masking Benchmark (PIIMB) [161] at revision `7d797b9f` (150,022 sentences in six tasks; CC BY-NC 4.0: never trained on, used to measure and, during development, to compare variants) for `pii`. CredData is scored by *line units*: a unit is one labeled line range, positive when any of its labels marks a credential. The snapshot has 64,474 units, 14,935 of them positive (about a third are openssl test vectors), and its policy counts test and example values, UUIDs, salts and nonces as credentials; these units are not the "true credentials" of CredData's first README (4,583 in 297 repositories), an earlier and smaller version. CredData's script replaces only the values labeled as credentials, with random strings of the same character classes that avoid words such as `key` or `pass`, repeated characters and sequences; negatives keep their original values. We know of no public, labeled benchmark of credentials in binaries; `secrets-bin` is examined on our own exams.

(2) **Splits fixed before measuring.** CredData is split 50/50 by repository with the seed 20260922: repositories sorted by positive units are paired, and a seeded coin sends one of each pair to DEV (169) and the other to TEST (168). The public scorer computes the split from the dataset's metadata, and the training-data builder imports it (Section 6.3).

(3) **Criteria written before training**, in the recipe, with a date and a prediction in words (Section 6.8). The date is our own record, not an external time stamp; today's factory freezes the thresholds of a version, not the conditions under which an exam metric is read, and the released recipes predate that check. Every verdict is reported, and so is every change to a criterion or to a written rule. By the criteria as first written, each of the three released models failed at least one: two false-alarm guards of `secrets-code`, the probe-recall floor of `secrets-bin` for two of its six seeds, and the precision floor of `pii` at the operating point its criterion first named, which was moved to the default point after the criterion was written; `pii` also ships the data mix that its pre-registered tie rule did not keep. All three were released.

(4) **Several seeds**: three for `secrets-code`, six for the others, plus controls on shuffled span labels that must score about zero; the spread is reported beside the released file.

(5) **The released model is picked on held-out data, with one exception.** For `pii` and `secrets-bin`, the seed with the best F1 on the recipe's own *select* split, before any exam was read. For `secrets-code`, the seed with the fewest findings on `code-fp-b`, a set of clean public repositories, among seeds whose CredData TEST F1 was within 0.01 of the ensemble's (Section 6.7). The three seeds' TEST

scores (0.788 to 0.799 on the published scope) were known when it was chosen, so the choice can move the headline F1 by up to 0.011; the chosen seed is not the best of the three on TEST.

(6) **Rivals in the same harness, where possible.** gitleaks 8.30.1 [145], with its default rules, and TruffleHog 3.97.6 [146], without verification so that nothing was sent to any service, ran on the same CredData files and machine and were scored by the same code. Neither follows CredData's labeling policy. CredSweeper's rules without its learned validator [148], the strongest rule-based baseline for CredData's categories, were not run, nor was any classical learned baseline, such as a linear model over character *n*-grams trained on the same DEV lines. Other PIIMB models are quoted from the public leaderboard (retrieved 2026-09-23). For binaries, the rival is `strings -n 8` with regular expressions, run on the probe only.

(7) **Bootstrap intervals over repositories.** CredData F1 comes with a 95 % interval from 1,000 bootstrap resamples of the repositories that hold a positive unit or a false positive of the tool (pooled counts, 2.5th to 97.5th percentile), which leaves a Monte Carlo error of a few thousandths on each end. It covers the sampling of repositories, not the variation between training runs. Only F1 carries an interval: differences between tools and the per-family figures have none. A change between two scorings of the same findings is resampled in pairs, with the paired mode of the public scorer. On PIIMB we give the six seeds' range instead, which is not a sampling interval.

Training data are compared with every exam through 64-byte content fingerprints (Section 6.3.1), but the checks came after the released models were trained (Section 6.12); what they find in the release data is declared below with its measured effect. CredData TEST is counted, never inspected: errors are studied on the 51 DEV repositories that fed no training window (194 training windows share a 64-byte block with their files). Speed is measured apart (Section 5).

Exams that were also development sets. Several exams also chose among development variants (Table 19). With about forty factory versions compared on them (Section 6.11), their figures are those of the survivors: partly in-sample, and optimistic as estimates for unseen data. Only the held-out extension modules of `secrets-bin` were set apart before any model was trained and never inspected.

8.2 `secrets-code`: credentials in source code

The job and the model. Find API keys, tokens, passwords, private keys and connection strings in code, configuration and other text, and report file, line, column, byte offsets, a severity and a snippet in which the detected value is masked (other text on the line is shown as it is), without ever contacting a service. The profile decodes base64 layers, ignores documentation examples such as AWS's `AKIAIOSFODNN7EXAMPLE`, and reports findings in test, example and documentation paths as warnings, which do not fail a scan. An optional ensemble adds the recipe's other two seeds with a two-of-three vote, for three times the work.

The data and the recipe. Most training windows are real code, from repositories cloned on 2026-09-19, with generated lines injected, credentials or look-alikes, of which only the credentials are marked (Section 6.3); the recipe draws 80,000. One window in five of every step holds labeled lines of 117 of the 169 CredData DEV repositories, without openssl (about a third of CredData's positive lines are its test vectors) and the 51 kept for analysis; half of their credential values are replaced by generated values of the same form, and the other half keep CredData's obfuscated values. Three seeds and a control ran 12,000 steps at $3 \cdot 10^{-4}$ in 24.5 minutes on one RX 6800 XT. During development, the steps that moved the ensemble's F1 on the complete TEST half were all changes of data (Section 6.5), each scored on TEST (Table 19).

Results on CredData. On CredData's TEST half without the four repositories that overlap its training corpus (164 of 168 repositories, files up to 4 MB, obfuscated values; Table 20 and Figure 15), `secrets-code`,

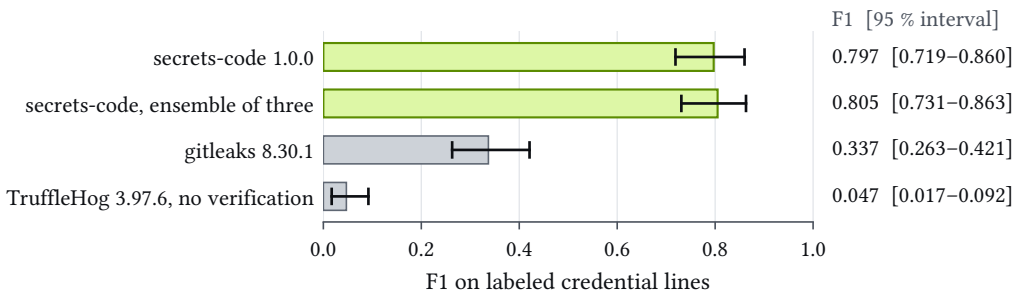


Figure 15: F1 on CredData TEST without the four repositories that overlap the training corpus of `secrets-code` (164 of 168 repositories, files up to 4 MB, 4,219 labeled credential lines, obfuscated values), with 95 % bootstrap intervals over repositories (1,000 resamples). Same machine and scorer for the four tools (the model on regions around the labels, the rivals on whole files); gitleaks with its default rules; TruffleHog, whose detectors target provider formats, without verification. Counts, precision and recall in Table 20.

which learned from labeled lines of the DEV half under the same labeling policy, finds 3.5 times as many labeled credential lines as gitleaks with its default rules (3,077 against 875; 3.1 times, 3,491 against 1,119, on all 168 repositories), at a close precision on labeled lines (0.879 against 0.905), and still misses about one in four (recall 0.729). At the operating point chosen on DEV for precision (`--bias -2`, Table 21a), it is more precise than gitleaks on labeled lines (0.932) and still finds 3.2 times as many (recall 0.663). The ensemble adds little (0.805) for three times the work; TruffleHog, whose detectors target provider formats and which we ran without verification, finds 104 lines. Precision counts labeled lines only, as in CredData's own benchmark: a line flagged outside every labeled unit is neither rewarded nor penalized, for any tool, and we have not counted such lines. The line rule also credits a finding that touches a labeled line whatever part of the line it covers, so it does not measure whether a masked copy hides the whole value. For orientation only, not as a validation, CredData's 2022 table, on an older version of the dataset, another version of gitleaks and the whole dataset, gives gitleaks an F1 of 0.334, consistent with the 0.344 of our harness on the complete TEST half. Files are limited to 4 MB, which covers 98 % of the dataset's labeled credential lines and 94 % of those of the published scope; over all sizes (4,474 positive units) the F1 is 0.772 [0.676–0.852], and 0.347 [0.269–0.426] for gitleaks.

Why four repositories are left out. The code corpus of the released weights held four repositories of CredData TEST: django (`f710ac3c`), kubernetes (`fc8343f4`), curl (`e13d78bf`) and ohmyzsh (`c8aa9b49`). No TEST label was used, but the model saw their code, as ordinary code with injected credentials, so the published figures leave them out, whatever the direction of the effect. Leaving them out favors the model: its F1 is 0.797 without them and 0.753 with them, the ensemble's 0.805 and 0.758, while gitleaks' is 0.337 and 0.344, and the ratio of lines found to gitleaks' rises from 3.1 to 3.5. The two halves of Table 20 isolate the four: they hold 1,076 of the 5,295 positive units, on which the model finds 414 (recall 0.385, against 0.729 elsewhere), the ensemble 436 (0.405) and gitleaks 244 (0.227, against 0.207 elsewhere). Whether the overlap causes the model's weakness there is untested; one possible reason is that only the injected credential of a generated window is marked, so any real credential in that code was taught as ordinary code. These four are left out whole, while the TEST lines that share 64-byte blocks with the DEV stream are handled line by line (below); how many positive units of the four fall in blocks shared with the code corpus was not measured.

Where the difference lies. By family (Figure 16), the gap is largest where a value has no fixed form: passwords (recall 0.88 against 0.06 for gitleaks), URL credentials and Basic authentication (0.99 against 0.01), generic keys (0.82 against 0.49) and tokens (0.69 against 0.44). Where the form is fixed, rules come close only for private keys (0.90 against 0.84); on provider patterns the model leads (0.85 against 0.28), a gap we have not explained: the mapping of CredData's categories to this family, gitleaks' allowlists for test and example values and the obfuscation of values are the first suspects. These family figures are

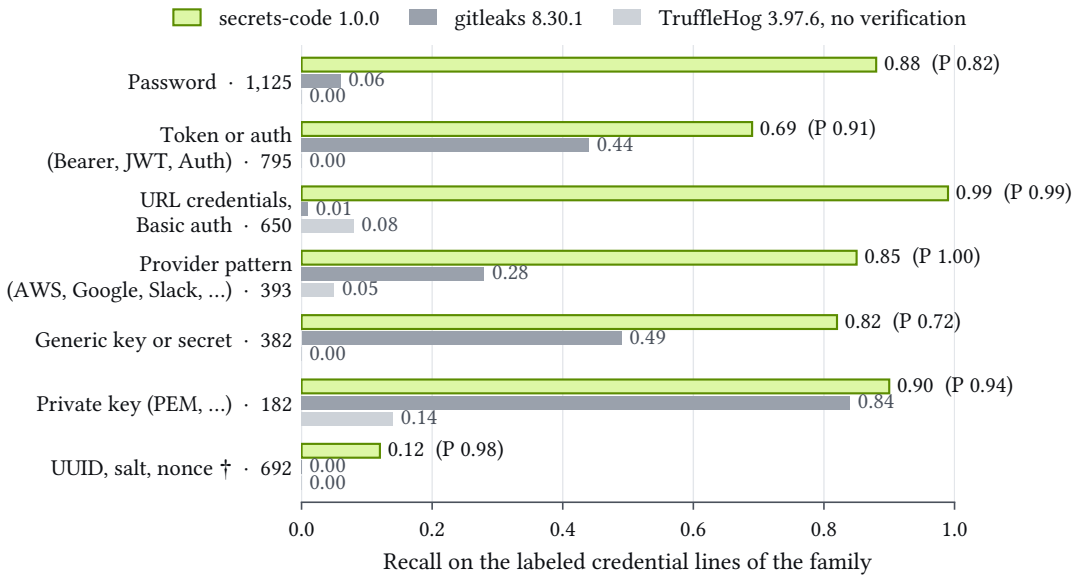


Figure 16: Recall by family of CredData categories, same data and conditions as Figure 15; after each family, its number of positive line units, and for `secrets-code` its precision on the family (P). The public scorer derives a unit's family from its official categories. † By design, `secrets-code` reports a UUID, salt or nonce only under an unequivocal credential name; CredData labels most of them as credentials.

partly in-sample: the recipe's development was steered by per-family recall targets on this TEST half (Table 19). UUIDs, salts and nonces, which CredData labels as credentials, are reported only under an unequivocal credential name, by design (recall 0.12, precision 0.98); an exploratory slice, defined after this weakness was known, leaves out the lines that hold only such values (3,527 positive units): there the F1 is 0.863 [0.800–0.909], and 0.389 [0.302–0.498] for gitleaks.

Operating points and weak spots. Two operating points were chosen on the 51 held-out DEV repositories before the released model was scored on TEST (Table 21a); 194 training windows share a 64-byte block with their files, so they are not wholly independent of training. On those repositories (1,042 positive units; precision 0.876, recall 0.728), the misses concentrate in UUID lines (recall 0.06 on 228 units), values with no name next to them (0.79 on 70), short quoted passwords (0.88 on 210) and `Authorization` headers (0.70 on 10), and 48 of the 107 false positives are short ordinary values under a credential-like name.

False alarms on clean data. CredData's precision counts labeled lines only. On 36 popular repositories without live credentials (Table 21b), the CLI reports 639 findings in 10,820 files, 531 of them warnings (keys, certificates and `Basic` headers of tests, fixtures and documentation). The 108 errors come mostly from integrity hashes in `package-lock.json` files, base64 images in a notebook and demo keys of example applications (44 in awesome-compose). Nine of the repositories were in the training data, so their counts can be off in either direction: five are CredData DEV repositories whose labeled lines fed the stream, at other commits (tornado, starlette, got, devise and awesome-compose), and five were in the code corpus of the generated windows (awesome-compose again, nginx, helm-charts, dotfiles and home-manager). Without them, the sets give 463 findings in 7,299 files, 420 of them warnings. Eight of the 36 are CredData repositories (the five of DEV above, and phoenix, ansible-for-devops and helm-charts of TEST), in which CredData labels test and example values as credentials: what the model was trained and scored to report on CredData counts here as a false alarm. The count on `code-fp-b` is optimistic in either row, because the released seed was chosen there, and no rival was run on these sets, so the counts have no reference.

Criteria and release. The recipe's four criteria were written on 2026-09-23, before training, with a prediction. The two on CredData passed: the ensemble's F1 and precision on the complete TEST half had to reach 0.55 and 0.85, and the research code measured 0.755 and 0.880 at release, with all three members at bias 0 (the released ensemble, each member at its own point, gives 0.758 and 0.867 with the CLI). The F1 threshold sat far below what the recipe without the real stream already scored on the same half (0.672 when scored like the released ensemble, 0.690 in the research count), so it could fail only on a collapse; the precision threshold was close to that recipe's precision (0.885 and 0.848). The written prediction, an F1 0.04 to 0.08 above that recipe, held in the research count on which the verdicts were computed (+0.065) and was exceeded when scored like the released ensemble (+0.086); its precision part, a fall of 0.01 to 0.03, held only in the latter (0.885 to 0.867), since the research count rose (0.848 to 0.880). The two false-alarm guards on `code-fp-b`, at most 129 findings for the worst seed and 107 for the ensemble, failed: the research scanner counted 571, 360 and 455 findings for the seeds and 410 for the ensemble (the CLI reports 408 for the released file). We released anyway, after reading 40 such findings of the recipe's previous candidate, 93 % of them in test, example and documentation paths and most of them credential-shaped test data: a judgment made after seeing the result. The recipe also trained three seeds where the release checklist asks for six, and the factory's rule would have released another seed, the best on *select*; the released one was chosen for its fewest findings on `code-fp-b` among the seeds with a CredData TEST F1 within 0.01 of the ensemble's, with the three seeds' TEST scores already known.

What reading context gets, and where rules win. The model decides from a value and its surroundings: the name it is assigned to, the syntax, the kind of file, the lines around it. That is how it finds a password that no pattern describes, and leaves alone a line named like a password that only reads an environment variable. Rules keep a slightly higher overall precision on labeled lines (0.905 against 0.879), come close on private keys, and keep speed (in an earlier session, on a busy machine, gitleaks read CredData roughly 80 times faster than the model reads code), verification of live credentials, which TruffleHog is built for and we disabled, and audit rule by rule. The release's advice: run both.

Training-data overlap. Beyond the four repositories, the released model was trained on the full stream of 24,000 windows built from DEV, and 847 of them (392 with a credential) share at least one 64-byte block with files of 136 TEST repositories; no TEST credential value is in the stream. Today's factory refuses that stream (Section 6.12). Only 12 of the 4,219 positive TEST lines and 122 of the negative ones fall inside those blocks, and scoring without them lowers the F1 by 0.00001 (paired 95 % interval −0.00093 to +0.00092), as computed by the public scorer's paired mode on a list of lines that we do not publish. Three caveats remain. DEV shares TEST's labeling policy (credential-shaped values in test code count), so part of the gain over rules may be CredData's conventions, which the model learned and gitleaks' default rules do not follow. The obfuscation favors the model: on CredData the credentials are random strings with no words in them, while negatives keep their natural form (Section 8.1), and half of the DEV values the model learned from keep that random form. We measured how much (Table 22): the same 4,737 files of the published scope, scanned as distributed and with the original values restored (fetched from the original repositories at the pinned commits, scanned, then deleted; no value was printed or stored; the fetching script is not published, because it handles third-party credential values), differ only inside the 4,219 credential lines. On the original values the recall of `secrets-code` falls from 0.729 to 0.583 (paired 95 % interval of the change −0.189 to −0.114) and its F1 from 0.797 to 0.694, while gitleaks barely moves (recall 0.207 to 0.201, F1 0.337 to 0.329): the model still finds 2.9 times as many labeled credential lines as gitleaks (2,460 against 850), not 3.5. The loss is in passwords (recall 0.881 to 0.482), most of all in passwords that are ordinary words in the original (646 lines, 0.870 to 0.389); authorization headers, URL credentials, private keys and UUIDs barely move. The figures of Table 20 are those of the benchmark as distributed, and they overstate the model's recall on real passwords.

Finally, CredSweeper [148], whose rules target CredData's categories, is not compared: its published figures come from a learned validator trained on CredData and are reported on the whole dataset without

a split, and we did not run it, not even with its rules alone.

Measured

CredData TEST without the four repositories that overlap the training corpus (164 of 168 repositories, files up to 4 MB, 4,219 labeled credential lines, obfuscated values; the same machine and scorer for every tool): `secrets-code` 1.0.0, 1.9 M parameters without heads in a 0.98 MiB file, trained partly on labeled lines of CredData's DEV half, has precision 0.879, recall 0.729 and F1 0.797 [0.719–0.860]; gitleaks 8.30.1 with its default rules has 0.905, 0.207 and 0.337 [0.263–0.421]: the model finds 3.5 times as many labeled lines (3,077 against 875; 3.1 times on all 168 repositories, 3,491 against 1,119); on the original values, which the benchmark obfuscates, 2.9 times (2,460 against 850) and F1 0.694 (Table 22). The recipe's three seeds score 0.788 to 0.799; on the complete TEST half (168 repositories) the released model scores 0.753 [0.665–0.841]. On 36 repositories without live credentials the CLI reports 639 findings, 531 of them warnings; 463 without the nine repositories that were in the training data. The count on `code-fp-b` is optimistic, because the released seed was chosen there.

8.3 `pii`: personal data in text

The job and the model. Mask much of the personal data in any text (names, contact details, identity and account numbers, dates, places, companies, credentials) as one type, `PII`: its character recall on PIIMB is 0.77, with an F2 far lower on legal texts and protocol traces (0.33 and 0.54); the share of entities left wholly or partly in clear was not measured, and a redacted copy can still identify a person. The `redact` profile replaces only the model's spans by markers such as `[PII_1]`, the same value always with the same marker, which keeps equal values linkable within a document, and checks that every other byte is unchanged, a check of the copy's integrity, not of its privacy. The policy is the training data's and the benchmark's: companies, occupations, dates and cities are masked too. 6,291,456 of the `nano_n15` model's 8,268,128 parameters (without heads) are entries of its hashed tables of byte 2-, 3- and 4-grams. Its stored operating point, a bias of −0.5 with the window gate on, was chosen for F1 on 512-byte windows of the pool's own *select* documents.

The data and the recipe. 192,000 windows of 512 bytes, each seen once. Eighty percent come from 387,391 documents of the TRAIN splits of four synthetic datasets, OpenPII-1M in 23 languages [209], Nemotron-PII [210], Gretel PII masking EN [211] and Gretel synthetic PII finance [212], after a check that left out 29,149 candidates sharing a sentence, a line or a masked template with an evaluation set. The rest are generated records (10 %), log lines (6 %) and code (4 %). Six seeds, from two training runs of three with a control each, ran 12,000 steps at $3 \cdot 10^{-4}$, 432–461 s per seed, and the factory released the best of the six on *select*. On the pool's test windows the seeds reach a span-F1 of 0.824 (0.816 to 0.830) against 0.444 for the factory's rule baseline (patterns with checksums, context words and name lists).

The criterion, written first. On 2026-09-23, before any model of the recipe was scored on PIIMB, the recipe required the mean of its six seeds to beat openai/privacy-filter [160] on the mean F2 (0.6621) and on ai4privacy-en (0.8426), with a mean precision of at least 0.70, so that no win is bought by over-masking, read at the "PIIMB mode" operating point below. openai/privacy-filter has 1.4 B parameters (1,399,486,865 as counted by the PIIMB leaderboard; its model card says 1.5 B), about 50 M active per token according to its model card. On 2026-09-24, after the pool-only mix of the recipe had missed the precision floor at that point (0.686 over its three seeds), we moved the reading to the default operating point, the one the file ships with, before any model of the released recipe was scored and without a change to the thresholds. At the default point the verdict is a pass (0.7774, 0.9624 and 0.765), and every seed passes each threshold alone. At the point first written, the released recipe passes both F2 thresholds but misses the precision floor (0.686 over its six seeds): by the criterion as first written, the verdict is a fail. The prediction written with the criterion, for the batch of recipes the released one comes from and at the "PIIMB mode" point, gave a 45 % chance that some version would pass all three thresholds there,

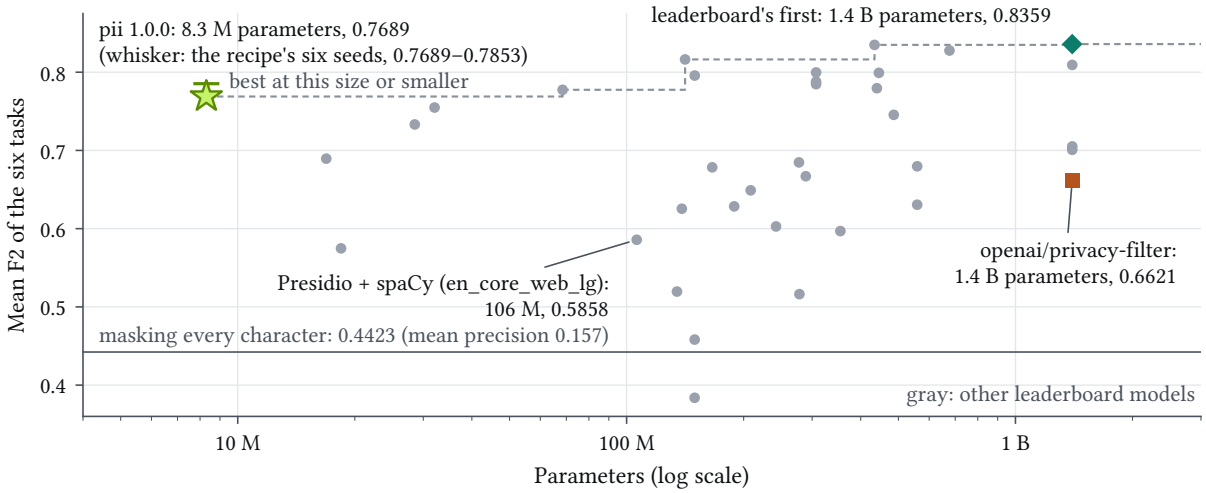


Figure 17: Mean F2 of the six tasks of PIIMB against parameters, for the 35 models of the public leaderboard (retrieved 2026-09-23; parameters as the leaderboard lists them) and `pii` 1.0.0 (star), measured by us with the leaderboard's metric on the same sentences. The whisker spans the recipe's six seeds; the dashed step is the best score at each size or smaller; the horizontal line is the score of masking every character. `pii` has 8,302,567 parameters with its heads, 6.3 M of them in its *n*-gram tables.

expected the batch's base to score a mean F2 of 0.58 to 0.66, expected privy and mapa-eur-lex to stay below openai/privacy-filter, and gave a mean F2 above 0.8426 a 2 % chance. No version measured at that point met the precision floor (0.61 to 0.69), the base scored 0.757, privy stayed below (0.518 against 0.536), mapa-eur-lex did not (0.374 against 0.313), and no mean F2 reached 0.8426. Until 2026-09-25 the recipe's copy of the prediction named the default point and the leaderboard's first place (0.8359) instead.

Results on PIIMB. The metric is the leaderboard's: character-level, label-agnostic masking, micro-averaged per task, ranked by the mean F2 of the six tasks (the F-measure with $\beta = 2$, which weights recall above precision); every sentence is scanned alone. Our scorer is a replica of the benchmark's metric code, not yet checked against a leaderboard model's published predictions. Trained on the TRAIN splits of the datasets behind four of the benchmark's six tasks, `pii` scores a mean F2 of 0.7689 (Table 23), 13th of 36 on the leaderboard and above openai/privacy-filter (0.6621; 1.4 B parameters, about 50 M active per token), a general filter with a narrower policy; on the two tasks from sources `pii` did not learn from, the two are level (privy 0.5360 against 0.5362; mapa-eur-lex 0.3325 against 0.3129, a gap smaller than the spread of the recipe's seeds there, 0.3325 to 0.4002). The model card of openai/privacy-filter lists eight categories (account numbers, addresses, e-mail addresses, persons, phone numbers, URLs, dates and secrets) and names over-redaction of organizations and locations as a failure mode, while PIIMB, like `pii`'s training data, counts companies, occupations and places as personal data. `pii` has the higher mean precision (0.773 against 0.751) and is smaller than all 35 leaderboard models; the twelve above it have 68 M to 1.4 B parameters (Figure 17), and the recipe's six seeds would rank 10th to 13th. The leaderboard models nearest to it in size, two encoders fine-tuned on Nemotron-PII, one of `pii`'s sources, score 0.7548 (32 M parameters) and 0.7775 (68 M). Most of the gap to the first place lies in privy, protocol traces (0.5360 against 0.8176), and mapa-eur-lex, legal texts in 21 languages (0.3325 against 0.3728; Figure 18). The released file, the factory's choice on *select*, is the lowest of the six seeds on the benchmark.

Operating points, weak spots and false alarms. F2 rewards over-masking: masking every character scores 0.4423 at a mean precision of 0.157, hence the precision floor. "PIIMB mode" takes for each seed the bias and window gate that maximize the byte F2 on the *select* documents cut into sentences, never on the benchmark; for the released seed (bias 2.0, gate off) the mean F2 rises to 0.8187 and the precision

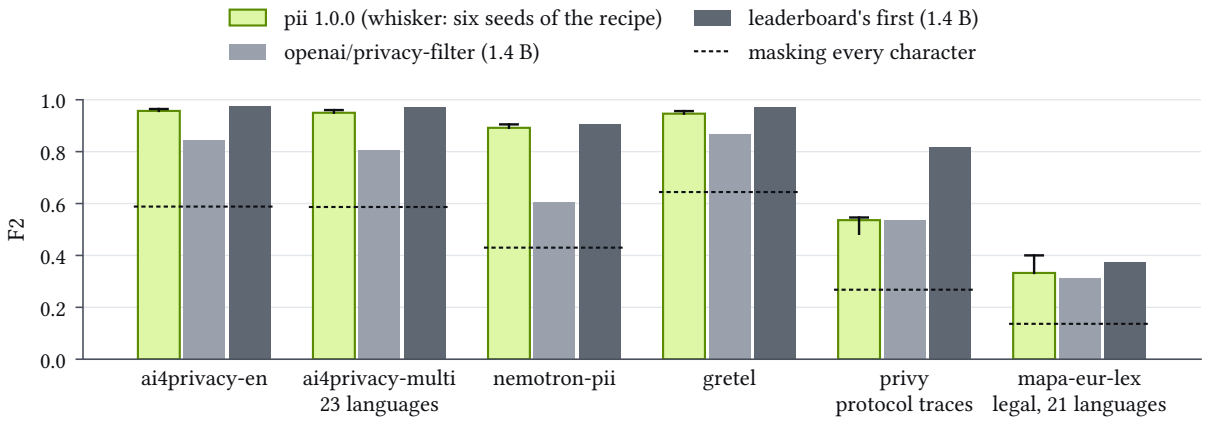


Figure 18: F2 by task on PIIMB: `pii` 1.0.0 at its default operating point (whiskers: the lowest and highest of the recipe's six seeds), openai/privacy-filter and the leaderboard's first, both quoted from the leaderboard, and the score of masking every character (dashed). Values in Table 23.

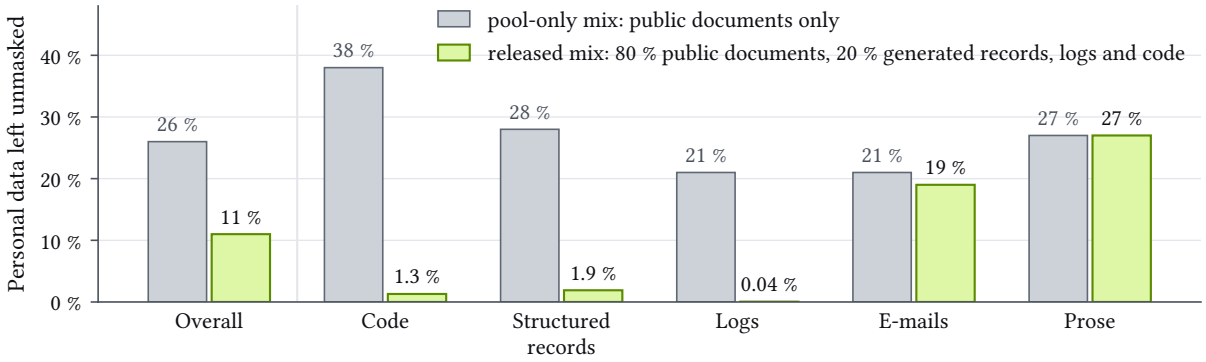


Figure 19: Personal data left unmasked by kind of input, on 2,000 generated documents built from held-out value lists, formats and code, for the two data mixes of the `pii` recipe (mean of three training runs of each). The test comes from the same generator family as the generated windows of the released mix and favors it, and its builder is not published, so the figure cannot be reproduced from the repositories; on PIIMB the two mixes tie (0.7862 and 0.7789).

falls to 0.680, below the floor, and over the six seeds to 0.8146 (range 0.8112–0.8187) at 0.686 (range 0.680–0.700). It is computed in PyTorch, because the CLI cannot switch the gate off (at the default point the two agree within 0.0006); the criterion as first written was read at this point, and since the change it is reported, not judged. The gain measures a weak spot: on single sentences the window head, trained on 512-byte windows, says "nothing here" too often. The body reads left to right, so a span can start late or end early. There is no exam of clean text; false alarms are measured as the per-character precision on the benchmark, 0.773 on average.

Which data mix ships. The recipe was also trained on the public pool alone, three seeds per mix. On PIIMB the pool-only mix scored 0.7862 and the released mix 0.7789, a tie by the pre-registered rule (a difference under 0.01), which keeps the mix measured first, the pool-only one. On 2,000 generated documents built from held-out value lists, formats and code, the pool-only mix left 26 % of the personal data unmasked and the released mix 11 % (Figure 19), with the largest differences in code, structured records and logs. The released mix ships against the tie rule, a product decision taken on a test that comes from the same generator family as its structured windows and favors them, and whose builder is not in the repositories; on the benchmark it is the lower-scoring mix, so the choice inflates nothing there, but the benchmark did serve to compare the mixes.

What reading context gets, and where rules win. A surname or a street has no pattern; it is personal data because of where it stands. The rules of the factory's gate reach a precision of 0.847 but a recall of 0.301 on the pool's test windows, and the leaderboard's Presidio with spaCy's large English model [156] (106 M parameters) scores 0.5858 on PIIMB. That entry runs English recognizers with a typed policy on a benchmark in up to 23 languages that counts every labeled character as one type, so, like the Privacy Filter, it scores lower there by construction, and the comparison favors a model trained on the benchmark's own generators. Rules and typed recognizers keep what this specialist lacks: the kind of each entity, a policy that keeps some kinds, and exact matching of formats with checksums.

Training-data overlap. `pii` learned from the TRAIN splits of three datasets from which the benchmark samples its tests, so four of the six tasks come from generators it learned from. Despite the pool check, one generator's documents share phrases across its splits: 1,018 of the 150,022 sentences share a 64-byte block with the pool (71 in ai4privacy-en, 79 in ai4privacy-multi, 799 in nemotron-pii, 69 in gretel, none in privy or mapa-eur-lex). Without them the released file scores 0.7686 and the six seeds 0.7771 (range 0.7686–0.7851). The 64-byte measure detects shared text, not shared templates: two sentences built from one template with other values share no block, so the four tasks remain in-distribution whatever this count says.

Measured

PIIMB, `sentences` subset (150,022 sentences, six tasks), the leaderboard's character-level label-agnostic F2: `pii` 1.0.0, 8.3 M parameters without heads in a 4.5 MiB file, trained on the TRAIN splits of the datasets behind four of the six tasks, scores a mean F2 of 0.7689 at a mean precision of 0.773, 13th of 36 on the leaderboard; the recipe's six seeds score 0.7689 to 0.7853 (mean 0.7774). openai/privacy-filter, 1.4 B parameters with about 50 M active per token and a narrower policy, scores 0.6621 (quoted from the leaderboard); on the two tasks from sources `pii` did not learn from, the two are level. The pre-registered criterion passes at the default operating point, to which it was moved after it was written, and fails on precision at the point it first named (0.686 against 0.70). Without the 1,018 sentences that share a 64-byte block with the training pool, the released file scores 0.7686.

8.4 `secrets-bin`: credentials inside compiled binaries

The job and the model. Find the credentials inside binaries that a text scanner skips (executables and libraries, object files, bytecode, WebAssembly, firmware), from the raw bytes, with no disassembler, `strings` pass or tokenizer, and report byte offsets with the surrounding printable string, masked. The CLI opens zip, gzip and tar containers; by default a finding must touch a printable string of at least 16 ASCII or UTF-16LE characters (`--all-bytes` reports the others). 50,331,648 of the `tiny_n18` model's 54,170,048 parameters (without heads) are table entries, and the n-gram memory takes 94.5 % of the file (Section 4.10). One model ships: an ensemble of three did not generalize out of sample.

The data and the recipe. The credential generator runs in binary mode over binaries from the Linux system folders of the training machine, not over the 280 public artifacts from which the data scripts rebuild a comparable corpus (Section 6.3): 70 % of the inserted blocks sit between NUL bytes like a string table, and long hexadecimal digests are negatives. Seven real negative windows join each step of 16 generated ones, drawn from 78,283 windows of binaries without credentials (Windows executables and libraries, twelve other formats, extension modules of 63 Python packages; no exam file), many of them not redistributable. Six seeds and a control ran 6,000 steps at 10^{-3}, output projections at a quarter of it, in 58 minutes. In development, the n-gram memory (Section 4.11) and the real negatives (Section 6.5) cut the false alarms most, as measured on `bin-fp-a` and on the PE files of `bin-fp-b`, two of the exams below.

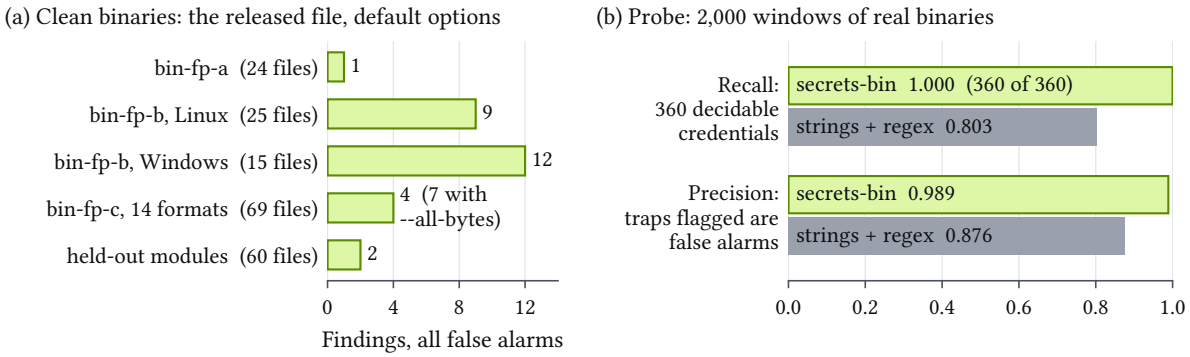


Figure 20: `secrets-bin` 1.0.0 on our own binary exams (AMD Ryzen 9 5900X, Windows 10, the `purebyte` CLI 1.0.0). (a) Findings on clean binaries with the default options, every one a false alarm; with `--all-bytes` the counts are equal except on bin-fp-c; the rival was not run on these sets. (b) The synthetic probe: 2,000 windows of 512 bytes cut from real binaries (1,200 untouched, 400 with a credential planted by the training generator, 400 look-alike traps from the same generator), each scanned as a file; recall on the 360 credentials decidable from the bytes, and precision counting every flagged trap as a false alarm. Neither tool flags an untouched window. The exams cannot be redistributed as they are.

False alarms on clean binaries. We know of no public, labeled benchmark, so the figures come from our exams, run with the `purebyte` CLI 1.0.0 and default options on the released file (AMD Ryzen 9 5900X, Windows 10; 3 to 12 threads, which do not change the results); they cannot be reproduced as they are, because some binaries cannot be redistributed and the probe is cut from our internal corpus. Five sets (`bin-fp-b` in two parts), 193 files, count every finding as a false alarm (Table 24, Figure 20a): the released file reports 1 to 12 per set. The 9 on Linux files all fall in 4 of the 8 files that overlap the training corpus. On bin-fp-c, 69 files in 14 formats absent from the corpus of the first binary models, the CLI opens 13,921 members of 9 containers and reports 4 findings, in two class files. Before the profile's rule for signed JARs, a rule written after this exam showed the failure, those containers gave 373 more, 371 on `SHA-256-Digest:` lines: the model flags base64 digests, and a rule removes them. The six seeds are measured with the research code (Table 24, right), which reports every span, scans containers as they are and keeps the tables at 32 bits; storing them at 4 bits changes the spans of about 1 % of windows (22 of 2,000), so the released seed's two sets of counts differ by a finding here and there.

Recall on planted credentials. The probe cuts 2,000 windows of 512 bytes from real binaries: 1,200 untouched, 400 with a credential written the way a compiler or packager leaves it (`NAME=value`, a value alone among other C strings, `--flag=value`, embedded JSON), and 400 look-alike traps (hashes, UUIDs, public keys, placeholders). The probe is synthetic and in-distribution: its credential values, names and look-alike traps come from the generator the model was trained with, in the layouts of its binary mode, and its background windows from an earlier corpus of binaries from the same machine folders as the training corpus (not checked for overlap with the training windows). It shows that the model learned what its generator makes, not how it does on real leaks, and the rival, `strings` with regular expressions, was not built for the generator's formats. 360 of the 400 credentials can be decided from the bytes; the other 40 are generic values with no name next to them. Each window is scanned as a file with the default options, and a credential counts as found when a finding overlaps it. `secrets-bin` finds all 360 at a precision of 0.989, counting every flagged trap as a false alarm, against a recall of 0.803 and a precision of 0.876 for `strings -n 8` with regular expressions for known prefixes, URL credentials and a suspicious name with a high-entropy value (Figure 20b). 360 of 360 is a recall of 1.000 with a 95 % Clopper–Pearson interval of 0.990 to 1.000; over the recipe's six seeds, 0.989 to 1.000 (two seeds found 356 of the 360). Neither tool flags an untouched window; the model flags 1 of the 40 undecidable values, the rival none. A stronger rival, the gitleaks catalog from which the generator draws its provider values, run over the output of `strings`, was not tried.

Criteria and weak spots. The recipe's criteria, written on 2026-09-23, bounded the false alarms and the probe recall of the six seeds. The six false-alarm criteria passed, but each of the three thresholds that the written prediction covers sat at its lenient end (`bin-fp-a` 5 to 10, the Windows DLLs 10 to 20, `bin-fp-c` 15 to 25; observed 2.7, 13.2 and 13.0), and that of the held-out modules is relative: half the mean of the previous candidate, 2.33, measured in the same exam run. The probe criterion, a recall of at least 0.99 for every seed, failed for two seeds, which missed 4 of the 360 decidable credentials (0.989), and the model was released with it failed; the released seed missed none. The exams are few (193 files, where one finding moves a mean) and partly optimistic: `bin-fp-a`, `bin-fp-b` and `bin-fp-c` were chosen after earlier models failed on them, this recipe's negatives were built to fix those failures, and `bin-fp-a` and the Windows part of `bin-fp-b` also chose the recipe's options (Table 19). Structured-looking data (bitmaps, lookup tables, runs of a repeated byte) can be flagged. A credential in a printable string shorter than 16 characters, with no longer string around it, is reported only with `--all-bytes`, a cost the probe does not measure, and recall on real leaks is not measured.

What reading bytes gets, and where rules win. `strings` keeps only printable runs and the rules match their form; the model reads every byte of the window and, on a probe whose credentials and traps its own generator made, finds the fifth of decidable credentials that the rules miss while flagging fewer traps. The rules stay faster and auditable; they were not run on the clean sets.

Training-data overlap. Parts of 10 exam files (8 of the 25 Linux files of bin-fp-b and 2 files of bin-fp-c) are also in the binary corpus of the generated windows; the six-seed figures leave them out. The held-out modules were set apart before training; the probe's background windows come from an earlier corpus of the same machine folders and were not checked for overlap with the training windows. The two exams that the criteria name for them have no manifest in the data scripts, so no fingerprint protects them (Section 6.3.1).

Measured

On five sets of clean binaries (193 files), `secrets-bin` 1.0.0 (54.2 M parameters without heads, 50.3 M of them in *n*-gram tables; 28.8 MiB), scanned by the `purebyte` CLI with default options, reports 1, 9 (all in files that overlap the training corpus), 12, 4 and 2 findings, every one a false alarm; these exams were chosen after earlier models failed on them, and two of them also chose the recipe's options. On a synthetic probe of 2,000 windows of 512 bytes cut from real binaries, whose credentials and traps come from the model's own training generator, it finds all 360 decidable credentials (95 % interval 0.990 to 1.000; six seeds 0.989 to 1.000) at a precision of 0.989, against a recall of 0.803 and a precision of 0.876 for `strings -n 8` with regular expressions, a rival not built for the generator's formats; neither flags an untouched window.

8.5 What the three examples teach

The three recipes are the survivors of about forty versions of the factory's tasks. Five lessons held across them.

Seed variance is larger than most architecture effects. The six `pii` seeds span 0.0164 in mean F2. Against the right yardstick, the standard error of a difference between two means of three seeds (about 0.005 with the spread of these six), the two data mixes differ by 1.4 standard errors, which does not resolve them; the two body directions, compared in an earlier recipe whose seeds spread less, differ by one standard error on this yardstick and by 2.3 on their own seeds (Section 4.11), a tie only by the pre-registered margin of 0.01. The worst of six `secrets-bin` seeds has 27 false alarms on the Windows DLLs and 32 on `bin-fp-c`, where the released seed has 12 and 0, and in development a single tail seed, whose variance lives in the data order, made most of the difference between binary recipes (Section 6.5). Hence the rule that three seeds inform and six decide.

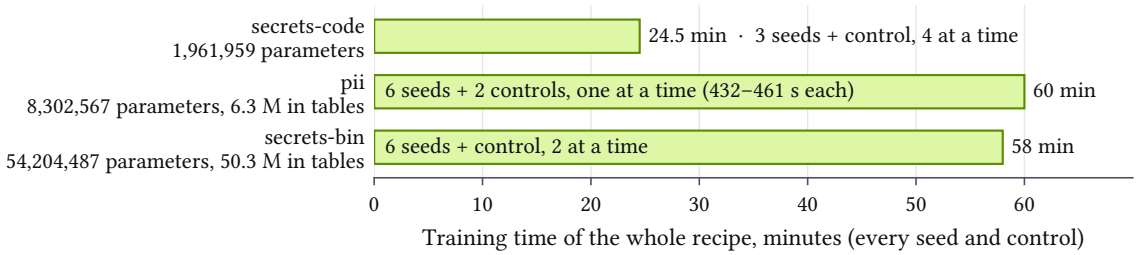


Figure 21: Wall-clock training time of each release recipe on one AMD Radeon RX 6800 XT (16 GB), every seed and the controls, with the model's parameters including the two heads. `secrets-code`: 1,468 s for three seeds and a control, four jobs at a time; `pii`: 432–461 s per seed, about 60 minutes for six seeds and two controls, one at a time; `secrets-bin`: 3,459 s for six seeds and a control, two at a time. The bias sweeps and checks of the fine-tuning stage come on top (Table 16).

Synthetic scores do not predict real ones. A generated test set shares the generator's regularities with the training set; an exam does not. Every recipe passes its synthetic gate by a wide margin, and the margin says little about the exams (Table 25). The last row adds the other half of the lesson: choosing on an exam is training on it, which our exams did not escape (Table 19).

More capacity hurt with generated labels. On code, growing the body raised the false alarms on clean repositories, while shrinking it kept the F1 on generated test windows (Section 4.11). For `pii`, measured on PIIMB, which thereby served as a development set, the larger `tiny_n18` preset lost 0.013 in mean F2, while twelve times more public documents gained 0.060. The capacity that helped was lookup memory, whose cost per byte does not grow with its size.

Pretraining helped only on binaries. On code, a body pretrained on 1.31 GB of code seemed to reduce false alarms on clean repositories until a from-scratch control at the same learning rate came close to it: the learning rate explained most of the gain. On binaries, a body pretrained on 1.31 GB of binaries did help, which we attribute to data scale, since the binary recipe reads fewer bytes with twice the body (Section 6.5). No released recipe uses a pretrained body.

The real exams cost more than training. Each recipe trains in minutes to an hour on a consumer GPU from 2020 (Figure 21); the three took about 2.5 GPU-hours with their bias sweeps and checks. The exams on real data ran on the CPU for longer, 1 h 47 min for CredData and the false-alarm sets of `secrets-code` and 4 h 9 min for the binary exams, and building them cost more still. For Vibe Training (Section 7) the scarce resource is the examination, not the training run.

Limits of the claim

- **Three jobs**, all span detection, trained by one factory, measured by the team that built them, on one desktop CPU and one GPU: they show what the architecture does on those jobs, not on byte jobs in general, and not against a classical learned baseline or a large generative model, neither of which was run on these exams.
- **CredData**: the published scope leaves out four TEST repositories, which favors the model (F1 0.753 on all 168); precision counts labeled lines only, and the line rule credits a finding on the right line whatever it covers; DEV, which the model learned from, shares TEST's labeling policy, and gitleaks ran with its default rules, CredSweeper's rules not at all; values are obfuscated, and on the original values the model's recall falls from 0.729 to 0.583 and its lead over gitleaks from 3.5 to 2.9 times (Table 22); TEST was scored at each of the recipe's last four versions against per-family recall targets (counted, never inspected), and the released seed was chosen with TEST in view.

- **PIIMB**: four of six tasks come from generators `pii` learned from, and the 64-byte overlap count cannot see shared templates; the rivals have narrower, typed policies; other models are quoted; no interval, only the seeds' range; the benchmark also compared development variants; the criterion's reading point was moved after it was written, and the released data mix ships against the pre-registered tie rule.
- **Binaries**: our own exams, few, partly chosen after failures and partly used to choose the recipe, not reproducible from public material; the probe is synthetic, made by the training generator, and its rival weak; no recall on real leaks, and no rival on the clean sets.
- **Differences between tools** carry no interval: the precision gap, the ratio of lines found and the per-task gaps on PIIMB are point estimates.

9 Research direction: an attachable document memory

The released specialists read windows of 512 bytes (Section 4). A state-space block carries a fixed-size state, which summarizes what it has read; it does not store it. Some jobs need more than a window: whether an identifier in a log line already appeared a megabyte earlier, whether a surname refers to a person named in full forty pages before, whether the value assigned to a key is the one this key was given at the top of a configuration. This section reports a study, run on the released models once they were fixed and *not part of the release*, of an exact document memory that can be attached to a released specialist without retraining it. We report what worked, what did not, and one result on real data that changed our own recommendation.

9.1 Design

The memory sits after the frozen body and before the heads. At every byte t it forms an address from the k bytes that end at t ($k = 8$ in the main design): a 64-bit polynomial hash of the k-gram with a mixing finalizer. A learned gate decides from the body's hidden state h_t whether to write. The gate is hard (0 or 1): training adds logistic (binary Gumbel) noise before the threshold and passes gradients with a straight-through estimator [69, 213, 214], so that training sees hard, bounded writes, as inference does with a deterministic threshold. For every address the memory keeps the sum of the values written there (a learned projection of h_t with 32 dimensions), how many writes it received and the position of the last one. Each byte reads its address *before* writing, which keeps the memory causal.

From the read-out the block builds features: the normalized stored value and its product with a query of the current state, the logarithm of the count, and exact integer features that no learned similarity provides reliably:

- **copy runs**: how many consecutive bytes have matched a stored k-gram whose position advanced by one at each step, that is, the length of the exact copy being read;
- **breaks**: whether a copy broke inside the current token (bytes of `[A-Za-z0-9_-]`);
- **coverage**: whether the copy reaches the first byte of the token and, at the token's end, whether the whole token was covered by one contiguous copy.

A small causal mixer (a depthwise convolution over sixteen feature rows and a multilayer perceptron) and a lookahead of 48 bytes, of the runtime's form (Equation (21)), turn the features into a residual added to h_t. The lookahead gives the block 48 bytes of right context, which the runtime's stream context does not allow today (Section 4.8). The residual's output projection starts at zero, so the specialist with the memory attached computes exactly what it computed before; we measured a maximum difference of 0.0 in log-probabilities on a specialist's test windows at initialization. Only the memory and the lookahead are trained (about 76,000 parameters); the body and the heads stay frozen.

The block is a table and a few small products per byte. In an unoptimized single-threaded C++ implementation it costs 5.3 µs per byte, of which 0.15 µs are sequential (hash, table read and write, copy length) and the rest are per-position products that parallelize like the body; the body itself costs about 68 µs per reported byte on one thread (about 26 ms per 512-byte window with stride 384). The memory therefore adds about 8 % on one thread. These timings were taken on the development desktop while a GPU job was running, so they are indicative. A byte-by-byte NumPy implementation matches the PyTorch streaming code within 2.4×10^{-7} (4.8×10^{-7} with the mixer).

This is close in spirit to side networks over frozen backbones [215, 216], to datastores of hidden states read by nearest neighbors [217, 218], to exact n-gram lookup over a corpus [219], to hashed n-gram memories gated by the context [81] and to n-gram heads that attend to what followed the same exact n-gram [220]. Its copy runs follow an earlier occurrence as induction heads do [221], and its copy length is the classic match model of context-mixing compressors [222]; its exact address avoids the "distraction" of learned keys that overlap as the memory grows [223]. State-space models are known to trade state size for associative recall [57, 58]; an exact memory next to the state is one way out.

9.2 Instruments

Long-context benchmarks for chat models measure generation. We measured recall directly, without generation, with four instruments, and wrote the success criteria before running any of them:

I1, recall by distance with a role.
Documents built from real code and configuration bytes, in which values are defined with a role (`token = "...";`, marked, or mentioned without one, unmarked) and reused later in neutral contexts at distances from 128 bytes to 512 KiB. Positives: reuses of marked values. Negatives: reuses of unmarked values, fresh values, and *near-copies* of marked values (one or two characters changed). Trained on 16 KiB documents, evaluated on streams of 1 MiB. Metric: balanced accuracy (chance: 0.5).

I2, key-value binding.
`KEY = "VALUE"` pairs and later probes `KEY -> VALUE'`: tag `VALUE'` only if it is the value of that key. The hard negative is the value of another defined key.

I3, real logs.
Public system logs: an identifier (IP address, UUID, hexadecimal or job identifier) is positive if the same identifier appeared at least 512 bytes earlier and not closer. Labels come from regular expressions.

I4, attached to a released specialist.
The memory attached to the released bodies of `secrets-code` and `pii`: retention on the specialist's own test windows, and reuses of values it tagged elsewhere in long documents.

The criteria were these: I1 passes with a balanced accuracy of at least 0.95 in every bucket up to 512 KiB, near-copy false alarms of at most 0.05 (the written criterion says neither per bucket nor on average; we read it on average) and at most 16 MB of state; I2 with a balanced accuracy of at least 0.90 at 32 KiB; I4 with a retention loss of at most 0.5 points of F1 and a recall of far reuses of at least 0.90 at false alarms of at most 0.02; the cost with at most 25 % more time per byte. Each instrument can be solved exactly by a short program (a regular expression for the role or the identifier and a table of the strings seen); the instruments test whether a learned block attached to a frozen body can do what such a program does, on real bytes, without being told the rule. The designs *h2* to *h10* of Table 26 were developed on the same eight evaluation documents of I1 (the coverage feature of *h9* was designed from an earlier design's errors on them), so their figures are the best of about nine designs, not held-out estimates, and their seeds vary only the training run.

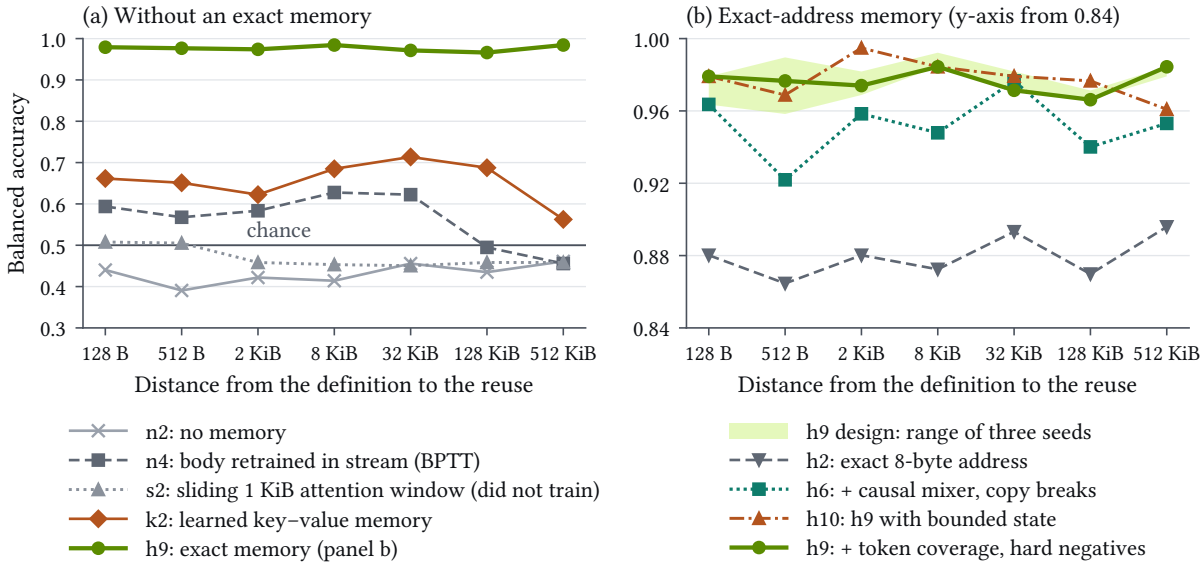


Figure 22: I1, recall by distance on streams of 1 MiB (trained on 16 KiB): balanced accuracy in each of seven distance buckets, 64 positives and 32 of each negative kind per bucket, eight documents. (a) Among the designs that trained, none without an exact memory stays far above chance (0.5) beyond 32 KiB; the sliding attention window did not train in 3,000 steps, and the exact-memory design *h9* is repeated for reference. (b) The exact-address designs, on an axis from 0.84; the band is the range of *h9* over three seeds.

9.3 Results

Measured

Streams of 1 MiB, balanced accuracy over seven distance buckets from 128 B to 512 KiB, 64 positives and 32 of each negative kind per bucket, eight documents (Table 26 and Figure 22). The best design, *h9*, holds a balanced accuracy of at least 0.966 in every bucket (mean 0.977; near-copy false alarms 0.027 on average, 0.062 at most) with seed 0, and at least 0.969 and 0.958 with two more seeds, whose near-copy false alarms average 0.054. The pre-registered I1 criterion therefore holds for seed 0 only, and only with its near-copy limit read on average (per bucket no seed meets it; seed 0 reaches 0.062): seeds 1 and 2 meet its accuracy and state limits but not its near-copy limit of 0.05. A variant with bounded state, *h10*, reaches at least 0.961 (mean 0.978) with 0.49 MB of state on 1 MiB streams, a state that stays bounded as the stream grows; it pays in noise, flagging 205 bytes of filler per MiB (*h9*: 0.8), and its near-copy false alarms average 0.040 and reach 0.125 in one bucket. On one constructed stream of 16 MiB (one document and one seed; 16, 16, 16 and 12 reuses at 1, 4, 8 and 15 MiB of distance; filler from the training split, because the held-out split holds about 11 MiB), both recall every reuse (12 of 12 at 15 MiB, a 95 % lower bound of 0.74); *h10* ends with 3.0 MB of state (*h9*: 84 MB). Removing the memory at inference drops recall to 0. Of the other criteria, I2 was met only with an exact association table, I4 in part for `secrets-code` and not for `pii` (Section 9.4), and the cost was met (about 8 % more time per byte on one thread).

Three results stand out. First, the fixed-size state of our 1.9-million-parameter body does not recall far: neither the frozen body read as one stream nor a body retrained with full backpropagation through 16 KiB discriminates beyond 32 KiB; truncated backpropagation through time is needed to *stream*, but it does not give *recall*. Second, learned keys confuse many values (the learned key-value memory reached 0.91 at 16 KiB and fell to 0.66 at 1 MiB with 4,096 or 65,536 entries), while exact addresses do not. Third, what closed the gap to the criterion was not a larger network but exact features: coverage of the token by one contiguous copy is what separates a reuse from a near-copy whose change falls outside the matching k-grams.

For key-value binding (I2), the learned comparison between stored value and current state was not

enough (balanced accuracy 0.74–0.85, with 0.50–0.95 false alarms on the value of another key); an exact association table (address: the previous word of at least three bytes; content: the first token that followed it) reached 0.94–0.98. On real logs (I3), false alarms on first occurrences were 0.0, and exact recall was 0.63 and 0.66 for reuses 512 B–2 KiB and 2–8 KiB away (1,164 and 342 positives), 0.51 at 8–32 KiB (151), and 0.34 and 0.29 at 32–128 KiB and beyond (44 and 7 positives: the last is 2 of 7); addresses of 4 and 8 bytes together made it worse (recall 0.59, false alarms 0.033), because short k-grams repeat too often in logs.

9.4 Attached to a released specialist, and a lesson from real data

Attached to `secrets-code` with its body and heads frozen, the memory and its lookahead together raised the specialist's gated span F1 on its generated test windows from 0.880 to 0.905 (one seed); the lookahead alone may explain part of this gain, and we trained no lookahead-only control. In constructed documents of 1 MiB, the attached model tagged 80–100 % of the long-distance reuses that the specialist alone missed. The I4 criterion was met for retention and only in part for those reuses: the attached model's recall of all the reuses it should tag was 0.62 to 0.85 below 128 KiB, 0.90 at 128 KiB and 1.00 at 512 KiB, with false alarms of 0.030 at 8 KiB, 0.052 at 512 KiB and at most 0.013 elsewhere, so both limits held together only at 128 KiB. Attached to `pii`, the memory kept retention within the criterion (a loss of 0.4 points) but failed on far reuses: recall 0.45 to 0.64, false alarms 0.20 to 0.67 (some of them may be personal data that the metric does not credit, since the negatives include new personal values). In the constructed documents, moreover, a plain string search did as well or better: run the specialist, then mark every later exact occurrence of every value it tagged (recall 0.88 to 1.00 of the reuses the specialist missed, with no false alarm, against 0.80 to 1.00 with up to 0.052 for the memory).

We then measured that "consistency pass" on real data: the complete TEST half of CredData [150] (all 168 repositories, 5,295 labeled credential lines; Section 8 also reports the published figures, which leave out four repositories that overlap the training corpus), with the flags of the released model (the published seed at its stored bias of 0.0). We scored about ten variants: copying to every other occurrence in a file or only to later ones, values of at least 8 or 16 bytes, within a file or across a repository, and the flags of the published seed or of a vote of three seeds, all at bias 0.0 (Figure 23 shows those of the published seed). None raised F1 by more than 0.002. The pass over whole files with values of at least 8 bytes, the only variant whose interval excludes zero, *lowered* F1 from 0.7525 to 0.7427 (paired bootstrap over repositories, 95 % interval of the change $[-0.024, -0.001]$): it added 11 true positive lines and 141 false positive lines, and 135 of the false positives came from copying a value that the model had flagged on a line labeled as a non-credential (the other 6 from unlabeled lines). Copying only later occurrences, as a one-pass memory would, changed F1 by -0.003 $[-0.006, +0.000]$. Real files repeat their placeholder passwords and example keys more often than their real secrets. A perfect judge that copied only correct flags would gain little: an estimated +0.001 in F1 within a file and +0.009 across the files of a repository for the published seed (+0.003 and +0.012 for the vote), with 95 % upper bounds of +0.002 and +0.029 (+0.010 and +0.034). A cheap judge (the three seeds agreeing) does not filter the errors, because the seeds share them.

Limits of the claim

The memory is lexical: it recalls identical bytes, not paraphrases or transformed values. Most of I1 is built from real bytes with planted values. In short source files, where the window already sees most repetitions, attaching a memory cannot add much, and copying decisions copies errors. We therefore do not ship it. Where it might help is where the evidence is far away or cannot be read again: streams of logs and telemetry, long documents, database dumps and key-value consistency. That is a hypothesis: on real logs our block's recall fell to about 0.3 beyond 32 KiB, and key-value binding needed an exact table. Such a memory must be trained with real repetitions that are *negative*, not only with "copy what you saw".

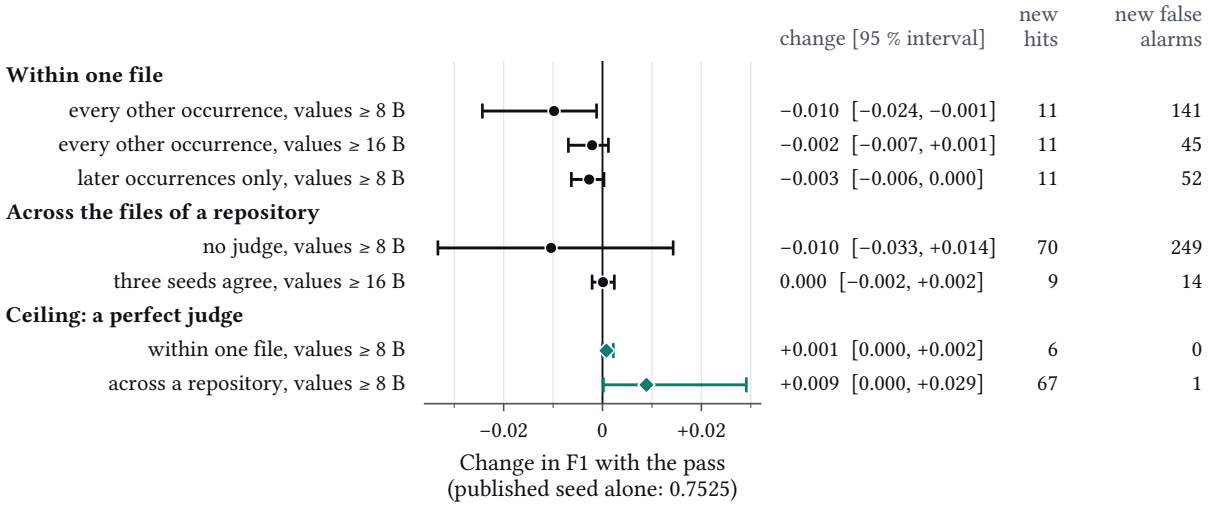


Figure 23: The consistency pass on the complete TEST half of CredData (168 repositories, files up to 4 MB, 5,295 labeled credential lines), with the published seed of `secrets-code` (F1 0.7525 alone): change in line F1 with its 95 % interval (paired bootstrap over repositories, 1,000 resamples), and the lines it adds. Copying flagged values within a file or across a repository adds more false alarms than hits; only the first interval excludes zero. A perfect judge that copies only correct flags would gain an estimated +0.001 within a file and +0.009 across a repository (95 % upper bounds +0.002 and +0.029).

10 Discussion

10.1 SLMs are the future, in a precise sense

The title makes a claim about the future, so it must say what it means. By *small language models* (SLMs) we mean small learned models of sequences, in the broad sense of models of sequences of symbols; AI Specialists are byte-level and non-generative, which puts them at the small end of the family, two to three orders of magnitude below the generative SLMs of a few billion parameters [10, 12, 13]. The claim is not that small models will replace large ones, and it is not that small task-specific models are new: counted by decisions, spam filters, fraud scores and ranking models already make most of them. It is a claim about the decisions over data that are now made by rules, or by calls to general models, and about where the new ones will go:

Thesis. SLMs are the future

The decisions over data that AI makes at scale (deciding, scoring, labeling, locating and filtering) are repetitive, narrow and high-volume. They are best made by small models trained for each job, run where the data are, and made by larger models or by people with the help of larger models. Generalists will remain the authors of specialists and the fallback for the inputs a specialist cannot decide; specialists will do the work. The thesis is wrong if small shared generalists with a small adapter per job reach the specialists' quality at a similar cost per decision, on the same machines.

Three arguments support it. First, the measurements of this paper: models of two to fifty-four million parameters, trained in minutes to an hour on one consumer GPU, that beat a mature rule set on recall on CredData (under the conditions of Section 8.2) and, trained on the sources of four of the PII benchmark's six tasks, score above some models with hundreds of times their total parameters, while twelve larger models, the smallest of 68 million parameters, score above `pii`; they answer a 64-byte input in 0.8 to 1.5 ms with eight threads of a desktop CPU, 3 to 6 ms on one core (Sections 5 and 8). Second, independent arguments that small models suffice for many of the calls inside agentic systems and are cheaper to run [3], and measurements showing that multi-purpose generative models spend far more energy per task than task-specific models on the same tasks [1]. Third, the economics of

Section 7: in waiting time, a specialist is repaid after 7×10^4 to 4×10^5 decisions against a quoted hosted decision API (from 8×10^3 against an assumed model that reasons for 2 s) under the assumptions of Section C, later in money; and against gitleaks on CredData it pays off through the errors it avoids once a missed credential costs more than about ten to twelve seconds of a person's time, with a false alarm at a minute (one benchmark, one rule set, false alarms counted on labeled lines). A fallback to a generalist needs calibrated confidence or an answer that abstains, which the released models do not have yet. The evidence has limits, listed in Section 11: three examples, one machine, and no measurement yet of agents building specialists.

10.2 The competing future

The strongest alternative to this thesis is that generalists absorb these jobs from the small end. One small general model on each device, with a small adapter per job, could do the work of many specialists: Program-as-Weights turns natural-language specifications into adapters for a 0.6-billion-parameter model [124], on-device foundation models ship with low-rank adapters for their tasks [206], and local models already answer most real single-turn queries, a share that grew quickly between 2023 and 2025 [224]. Running many separate models also has an engineering cost that a single shared model avoids [225]. Where that future arrives first, specialists trained from scratch lose their main argument.

They keep it where a shared model does not fit or cannot be trusted: on machines with no accelerator and megabytes of memory, on bytes that no pretrained model has read (machine code, protocol traces), where an audit needs the same bits every time, and wherever the cost per byte matters more than the cost per model, as in scanning whole repositories or data exports. The observation that would settle it is a comparison we have not made: a small generalist with per-job adapters and a specialist trained from scratch, on the same examinations and the same processor, with their cost per decision. Small pretrained encoders fine-tuned for one job, which we count as specialists, already sit between the two (Section 3.3).

10.3 The ARM of AI: an analogy and its limits

The phrase we use for PureByte's ambition is *the ARM of AI*. By Arm's own account, ARM did not win by making the fastest processor [164]: it won with an architecture designed for efficiency (the first ARM chip ran on about 0.1 W [165]), licensed so that others could build on it, and small enough to go into every device; more than 350 billion Arm-based chips had been shipped by March 2026. The analogy maps point by point:

- **Efficiency as the design goal.** ARM optimized work per watt; PureByte optimizes work per decision: a model the size of the job, one linear pass over bytes, kernels that use the CPU's vector units, no accelerator required; its ternary weights and 4-bit tables shrink the bytes a device must hold (Sections 4 and 5).
- **An architecture that others build on.** ARM licenses its instruction set for a fee; PureByte publishes its model format, its runtime and its training stack under Apache-2.0, as RISC-V publishes its instruction set, so that anyone can train a specialist (Section 6), or have an agent train one (Section 7), and ship it as a file that any conforming runtime executes. The weights of the three released examples are under a separate model license that allows their use but not their redistribution, models derived from them or a hosted service whose main value is the model: what is open is what others need to build their own.
- **Everywhere.** ARM runs in phones, cars and sensors. An AI Specialist runs on servers and laptops, is built for x86-64 and arm64 Linux devices such as some routers and small robots (not yet tested there), and, with the changes we quantify, could run in a microcontroller (Section 5.8.1).
- **An ecosystem.** ARM's value is in its partners' chips; PureByte's will be in the specialists that others train for their own jobs and data, of which the three released examples are the first.

The analogy has limits, and we state them. ARM is hardware intellectual property with decades of silicon and an industry behind it; PureByte is a young software architecture with one runtime, one training stack and three example specialists. ARM's efficiency is measured in silicon per watt; ours in operations and milliseconds per decision on existing processors. And on the smallest devices efficiency comes from integer arithmetic on neural accelerators and vector units, which runtime 1.0.0's binary32 contract cannot use: an integer profile would need its own contract and its own validation, since an int8 path we measured changed spans (Section 4.11). "The ARM of AI" is therefore a design principle and an ambition, not a description of where we are: efficiency first, an open format, every device, and an ecosystem built by others.

10.4 Origin: a laboratory for efficient training

PureByte began as a laboratory question: how small can a useful model be, and how cheaply can it be trained, if every number is measured with its conditions? The answers so far are in this paper: networks of two to four million parameters (plus lookup tables) for three real jobs; training from scratch, with no pretrained body, on one consumer GPU, from 7.5 minutes per seed to 58 minutes for six seeds and a control; about 2.5 GPU-hours for the three examples with all their checks; and a factory that refuses some of the shortcuts that make small models look better than they are and records the rest. The laboratory stance explains the paper's style: criteria written before measuring, with every change to them reported; controls for every result and rivals where they could be run; the failures reported next to the successes; and a published withdrawal (Section 9.4).

10.5 Your data, your device

The practical meaning of *train with your data, not a generalist* is that the organization that has the data can own the model. A specialist trained on an organization's own labeled logs, forms or binaries answers questions that no public model was trained for, never sends the data anywhere, and runs on the organization's own machines. Our examples were trained on public and generated data; a model trained on private personal data carries information about it in its weights and tables, and should be kept where the data are. The same property reaches small devices: a gateway that screens its own configuration files and logs, a robot that classifies its own messages, an industrial controller that checks its own inputs. Each is a byte job with local data and a processor with megabytes to a few gigabytes of memory, at rates far below those of full network traffic: the throughput measured on a desktop, about 131 kB/s with 23 threads, is two orders of magnitude below a router's line rate, so traffic would have to be sampled, and for a robot the worst-case latency, which we have not measured, matters more than the median. The runtime is built for x86-64 and arm64 Linux machines of that class (tested in continuous integration on the server versions of those architectures, not on such devices), and Section 5.8.1 quantifies what a microcontroller would additionally require. We present microcontrollers as the next target, not as a result.

10.6 Privacy, energy and cost

Three consequences follow from size. *Privacy*: a model that runs in the process that holds the data needs no network, and the runtime opens no outgoing connection except to download a model (Section 5); detectors of secrets and personal data that send what they inspect to a remote service defeat their purpose. Any local model shares this property; what the specialist adds is that it fits where a local generalist does not. *Energy*: the operations per byte of Table 2 are 400 to 800 times fewer than those of a small generative model before it generates any answer; across hardware, our rough estimate puts the energy per byte 7 to 28 times lower, and we did not measure energy (Section 3.4). *Cost*: the marginal cost of a decision is a few milliseconds of a processor the user already owns. None of these is a new idea; what is new is that small models built for the job make them available to jobs that until now needed a large model or a set of brittle rules.

11 Limitations and threats to validity

We collect here what limits the claims of this paper, so that a reader can weigh each result against it.

Scope of the evidence. Three example specialists, two of them about credentials and one about personal data, are the whole empirical base for the claims about specialists. Two of their jobs have public benchmarks and widely used tools; the third, credentials in binaries, has neither that we know of. They are not a sample of all byte jobs, and all three answer the same type (byte ranges with a gating decision); the other answer types run in the runtime but have no trained model. Jobs whose answers need knowledge of the world, free-text answers, or inputs unlike anything in the training data are outside what we measured (Section 3.3).

The pre-registration record. None of the three released models met every pre-registered criterion as first written. `secrets-code` failed its two false-alarm guards on a clean-repository examination and was trained on three seeds instead of six; its released seed was chosen on that examination, among seeds within 0.01 of the ensemble's CredData F1, with the three seeds' scores on CredData's test half already known. `secrets-bin` failed its probe-recall floor on two of six seeds. The criterion of `pii` was moved to another operating point after it was written, and fails its precision floor at the point first written; its data mix shipped against the pre-registered tie rule. The dates of the criteria are the author's record, not an external time stamp; today's factory freezes the thresholds of a version, not the conditions under which an examination is read, and the released recipes predate that check (Section 8).

Examinations. Several examinations were also used during development to compare variants: CredData's test half steered the data steps of `secrets-code`, a clean-repository examination chose its learning rate and its released seed, the PII Masking Benchmark chose among variants of `pii`, and the clean-binary sets chose options of `secrets-bin`. With about forty versions compared on the same examinations, their figures are those of survivors, not of untouched test sets. CredData randomizes only the values it labels as credentials, and `secrets-code` learned from such values in the benchmark's other half; on the original values, which we measured, its recall falls from 0.729 to 0.583 and its F1 from 0.797 to 0.694, mostly on passwords that are ordinary words, while gitleaks barely moves. Its figures leave out four test repositories that overlap its training corpus, which favors the model, and nine repositories of its clean-repository examinations were in its training data. `pii` was trained on the sources of four of the benchmark's six tasks and masks by the benchmark's broad policy; its main rival has a narrower policy, and its model card names no training data. The planted-credential probe of `secrets-bin` is synthetic and in-distribution, and some of its examinations use binaries that cannot be redistributed. Rival tools were run in the same harness on CredData and the probe; on the PII benchmark their figures are the leaderboard's.

Missing comparisons. We did not run CredSweeper's rules without its validator, the strongest rule baseline for CredData's categories, nor any rival on the clean code and binary sets; nor did we compare classical learned baselines (logistic regression or gradient boosting over character features, fastText [226]) on the same data, nor a candidate extractor with a learned verifier, nor a small generalist with per-job adapters, nor large generative models on our examinations. The evidence supports "train a small model on the job's data"; it does not isolate the contribution of this architecture against every learned alternative.

One machine. Every speed, latency and memory figure we measured comes from one desktop (AMD Ryzen 9 5900X, 12 cores), on Windows for the runtime's timings and on Linux under WSL2 for the memory figures and the start-and-load times; figures quoted from others are marked. Continuous integration builds and tests the runtime on Linux, macOS and Windows, x86-64 and arm64, but it checks correctness, not speed, and no router, robot or microcontroller was tested.

Training hardware and reproducibility. The released specialists were trained on one AMD Radeon RX 6800 XT; a rented NVIDIA RTX 5090 was used only to time training steps. Training on NVIDIA GPUs and on Apple silicon uses the same stock PyTorch code in float32; a device-parity test exists but has not been run on those devices, and no recipe has been trained on them. On a CPU the public trainer computes, bit for bit, what the code that trained the released models computes, which can be verified on the machine that recorded the fingerprints; GPU runs are not bit-reproducible, so a retraining is a new draw from the recipe's seeds, not the released weights, and no retraining with the public code has yet been scored on the examinations. Some data cannot be rebuilt from the repositories (the negative binaries of `secrets-bin`; the code corpus of the release was not pinned), and the data the releases were trained on overlap examination sets that today's factory refuses unless the overlap is dropped (Section 6).

What the factory enforces. Leak checks against examinations of role `test`, the gate, the control, runtime equivalence, criteria frozen per version and the immutability of releases are enforced in code. Other rules are recorded but do not block a release: a failed pre-registered criterion, a choice made with an examination in view, an override of a pre-registered rule, and "six seeds decide", which is a checklist item. The released examples took some of these exceptions, as the paragraphs above and Section 6 report.

Vibe Training. The argument of Section 7 rests on measurements of the specialists and on a cost model whose build costs and prices are assumptions. We did not measure agents building specialists (time, attempts, human interventions), nor large generative models answering our examinations. The account of this project's development is the author's.

The theses. "SLMs are the future", "the ARM of AI" and "train with your data" are positions. We argue them with the evidence above and with independent work, and we state where each stops and what would refute it (Sections 3.3 and 10); they are not results.

The document memory. The memory of Section 9 is unreleased research, measured mostly on constructed documents with real bytes, and on one real examination set where it did not help; its pre-registered criteria held only in part: I1 for one of three seeds (its near-copy limit read on average), I2 only with an exact association table and I4 in part; its designs were developed on the evaluation documents of I1.

12 Conclusion

PureByte is a bet on the small end of AI. Its AI Specialists read raw bytes, answer with decisions, scores, labels and byte ranges, and are sized for one job: two to fifty-four million parameters, one to twenty-nine mebibytes, 3 to 6 ms per short (64-byte) decision on one core of a desktop CPU and 0.8 to 1.5 ms with eight of its threads, the same bits every time on a platform. They are trained from scratch in minutes to an hour on one consumer GPU by a factory that blocks leaked examinations, silent options and unverified exports and records failed criteria, while some of its rules are written rather than enforced (its leak checks came after the three released examples, whose overlaps we measured); and they can be trained by an AI agent from a description of the job, which is what we call Vibe Training.

The three released examples show that the approach holds on three real jobs, two of them with widely used rival tools, within the conditions we state for each. On 164 of the 168 repositories of CredData's test half, a 1.9-million-parameter model finds three and a half times as many labeled credential lines as a mature rule set with its default rules, at a close precision on labeled lines (3.1 times on all 168), after learning partly from the benchmark's other half, and 2.9 times on the original values, which the benchmark obfuscates. An 8.3-million-parameter model scores F2 0.769 on the PII Masking Benchmark, 13th of 36 and above a privacy filter with 170 times its parameters (6 times its active parameters) and a

narrower policy, after training on the sources of four of the benchmark's six tasks (on the other two tasks the two models are level); the best small pretrained encoders remain ahead at a few times its cost per byte. None of the three met every pre-registered criterion as first written, and we report each. The research on an attachable document memory shows the architecture's modularity, with an exact memory that recalled reuses 15 MiB away on one constructed stream, and a real-data test that stopped us from shipping it where it would have copied errors.

The theses of the paper go further than its measurements, and we have marked where and what would refute them. We believe that most of the decisions over data now made by rules or by calls to general models will be made by small models made for their jobs, often by larger models; that efficiency, an open format and every device are the right principles for that world, which is what we mean by the ARM of AI; and that the organization that owns the data should own the model. The next steps are measurable: specialists for jobs and answer types beyond the three examples, comparisons with learned baselines and with small generalists with adapters, a benchmark of agents doing Vibe Training, microcontroller targets, and an exact memory trained on real negative repetitions. The code, the model format and the training stack are open, so that others can check the numbers and build the specialists we have not.

Acknowledgements and disclosure

This work was carried out by the author with the assistance of an AI coding agent (Anthropic's Claude), which wrote most of the code, ran the training and evaluation campaigns, and drafted this paper under the author's direction; the author defined the goals, the success criteria and the decisions to release, and is responsible for the content. The development of PureByte is therefore itself an instance of the loop described in Section 7. The expert reviews that preceded publication were also run with AI agents, each instructed to refute the paper; their findings were applied or reported as limits.

Competing interests and funding. The author develops PureByte and is the licensor of the weights of the released specialists. This work received no external funding.

Data. The public examinations are CredData (Samsung) and the PII Masking Benchmark; the binary examinations of `secrets-bin` are our own and partly not redistributable (Section 8.4). The rival tools and models were run in the same harness as the specialists or taken from the benchmark's public leaderboard, as Section 8 states for each; the binary rival was run on the probe only. Section D lists every dataset with its license, and the attributions the licenses require. No credential found during this work was verified against any service, and no credential value or personal datum appears in this paper.

References

[1] Sasha Luccioni, Yacine Jernite, and Emma Strubell. Power hungry processing: Watts driving the cost of AI deployment? In *The 2024 ACM Conference on Fairness, Accountability, and Transparency (FAccT '24)*, pages 85–99, 2024. doi: 10.1145/3630106.3658542.

[2] David Golchinfar, Daryoush Vaziri, and Alexander Marquardt. Playing DOOM with 1.3M parameters: Specialized small models vs large language models for real-time game control. *arXiv preprint arXiv:2604.07385*, 2026. URL https://arxiv.org/abs/2604.07385.

[3] Peter Belcak, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov. Small language models are the future of agentic AI. *arXiv preprint arXiv:2506.02153*, 2025. URL https://arxiv.org/abs/2506.02153.

[4] Zhenyan Lu, Xiang Li, Dongqi Cai, Rongjie Yi, Fangming Liu, Xiwen Zhang, Nicholas D. Lane, and Mengwei Xu. Small language models: Survey, measurements, and insights. *arXiv preprint arXiv:2409.15790*, 2024. URL https://arxiv.org/abs/2409.15790.

[5] Chien Van Nguyen, Xuan Shen, Ryan Aponte, Yu Xia, Samyadeep Basu, Zhengmian Hu, Jian Chen, Mihir Parmar, Sasidhar Kunapuli, Joe Barrow, et al. A survey of small language models. *arXiv preprint arXiv:2410.20011*, 2024. URL https://arxiv.org/abs/2410.20011.

[6] Fali Wang, Zhiwei Zhang, Xianren Zhang, Zongyu Wu, Tzuhao Mo, Qiuhao Lu, Wanjing Wang, Rui Li, Junjie Xu, Xianfeng Tang, Qi He, Yao Ma, Ming Huang, and Suhang Wang. A comprehensive survey of small language models in the era of large language models: Techniques, enhancements, applications, collaboration with LLMs, and trustworthiness. *ACM Transactions on Intelligent Systems and Technology*, 16:1–87, 2025. doi: 10.1145/3768165.

[7] Akanksha Gupta, Bijo Thomas, Harshita Asnani, Phanindra Reddy Madduru, Samia Feroze, Shreyas Subramanian, Vikram Elango, and Mecit Gungor. Small language models (SLMs) can still pack a punch: A survey. *arXiv preprint arXiv:2501.05465*, 2025. URL https://arxiv.org/abs/2501.05465. Updated 2026.

[8] Lihu Chen and Gaël Varoquaux. What is the role of small models in the LLM era: A survey. *arXiv preprint arXiv:2409.06857*, 2024. URL https://arxiv.org/abs/2409.06857.

[9] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*, 2023. URL https://arxiv.org/abs/2306.11644.

[10] Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, et al. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*, 2024. URL https://arxiv.org/abs/2404.14219.

[11] Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J. Hewett, Mojan Javaheripi, Piero Kauffmann, et al. Phi-4 technical report. *arXiv preprint arXiv:2412.08905*, 2024. URL https://arxiv.org/abs/2412.08905.

[12] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, et al. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*, 2024. URL https://arxiv.org/abs/2408.00118.

[13] Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Gabriel Martín Blázquez, Guilherme Penedo, Lewis Tunstall, Andrés Marafioti, Hynek Kydlíček, Agustín Piqueres Lajarín, Vaibhav Srivastav, et al. SmolLM2: When smol goes big – data-centric training of a small language model. *arXiv preprint arXiv:2502.02737*, 2025. URL https://arxiv.org/abs/2502.02737.

[14] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025. URL https://arxiv.org/abs/2505.09388.

[15] Meta AI. Llama 3.2: Revolutionizing edge AI and vision with open, customizable models. https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/, 2024. 25 September 2024; accessed 2026-09-25.

[16] Loubna Ben Allal, Anton Lozhkov, and Elie Bakouch. SmolLM – blazingly fast and remarkably powerful. Hugging Face blog, https://huggingface.co/blog/smollm, 2024. 16 July 2024; accessed 2026-09-25.

[17] Olivier Lacombe, Kathleen Kenealy, Kat Black, Ravin Kumar, Francesco Visin, and Jiageng Zhang. Introducing Gemma 3 270M: The compact model for hyper-efficient AI. Google Developers Blog, https://developers.googleblog.com/en/introducing-gemma-3-270m/, 2025. 14 August 2025; accessed 2026-09-25.

[18] Ronen Eldan and Yuanzhi Li. TinyStories: How small can language models be and still speak coherent English? *arXiv preprint arXiv:2305.07759*, 2023. URL https://arxiv.org/abs/2305.07759.

[19] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015. NIPS 2014 Deep Learning Workshop.

[20] Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alexander Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes. In *Findings of the Association for Computational Linguistics: ACL 2023*, 2023. arXiv:2305.02301.

[21] Yao Fu, Hao Peng, Litu Ou, Ashish Sabharwal, and Tushar Khot. Specializing smaller language models towards multi-step reasoning. In *International Conference on Machine Learning (ICML)*, 2023. arXiv:2301.12726.

[22] Justin Zhao, Timothy Wang, Wael Abid, Geoffrey Angus, Arnav Garg, Jeffery Kinnison, Alex Sherstinsky, Piero Molino, Travis Addair, and Devvret Rishi. LoRA land: 310 fine-tuned LLMs that rival GPT-4, a technical report. *arXiv preprint arXiv:2405.00732*, 2024. URL https://arxiv.org/abs/2405.00732.

[23] Yubo Ma, Yixin Cao, YongChing Hong, and Aixin Sun. Large language model is not a good few-shot information extractor, but a good reranker for hard samples! In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 10572–10601, 2023. doi: 10.18653/v1/2023.findings-emnlp.710.

[24] Chengwei Qin, Aston Zhang, Zhuosheng Zhang, Jiaao Chen, Michihiro Yasunaga, and Diyi Yang. Is ChatGPT a general-purpose natural language processing task solver? In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1339–1384, 2023. doi: 10.18653/v1/2023.emnlp-main.85.

[25] Martin Juan José Bucher and Marco Martini. Fine-tuned 'small' LLMs (still) significantly outperform zero-shot generative AI models in text classification. *arXiv preprint arXiv:2406.08660*, 2024. URL https://arxiv.org/abs/2406.08660.

[26] Mohammad Firas Sada, John Graham, Mahidhar Tatineni, Dmitry Mishin, Seungmin Kim, and Frank Würthwein. LLMs or naive Bayes? old gems or new ways. In *Proceedings of the Practice and Experience in Advanced Research Computing 2026 (PEARC '26)*, pages 1–6, 2026. doi: 10.1145/3785462.3815880.

[27] Ido Levy, Eilam Shapira, Yinon Goldshtein, Avi Yaeli, Nir Mashkif, and Segev Shlomov. TabAgent: A framework for replacing agentic generative components with tabular-textual classifiers. *arXiv preprint arXiv:2602.16429*, 2026.

[28] Lingjiao Chen, Matei Zaharia, and James Zou. FrugalGPT: How to use large language models while reducing cost and improving performance. *Transactions on Machine Learning Research*, 2024. arXiv:2305.05176.

[29] Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Ruhle, Laks V. S. Lakshmanan, and Ahmed Hassan Awadallah. Hybrid LLM: Cost-efficient and quality-aware query routing. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2404.14618.

[30] Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M. Waleed Kadous, and Ion Stoica. RouteLLM: Learning to route LLMs with preference data. *arXiv preprint arXiv:2406.18665*, 2024.

[31] Harsha Nori, Yin Tat Lee, Sheng Zhang, Dean Carignan, Richard Edgar, Nicolo Fusi, Nicholas King, Jonathan Larson, Yuanzhi Li, Weishung Liu, et al. Can generalist foundation models outcompete special-purpose tuning? case study in medicine. *arXiv preprint arXiv:2311.16452*, 2023. URL https://arxiv.org/abs/2311.16452.

[32] Arnav Gudibande, Eric Wallace, Charlie Snell, Xinyang Geng, Hao Liu, Pieter Abbeel, Sergey Levine, and Dawn Song. The false promise of imitating proprietary LLMs. *arXiv preprint arXiv:2305.15717*, 2023. URL https://arxiv.org/abs/2305.15717.

[33] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020. URL https://arxiv.org/abs/2001.08361.

[34] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Emergent abilities of large language models. *Transactions on Machine Learning Research*, 2022. arXiv:2206.07682.

[35] Rylan Schaeffer, Brando Miranda, and Sanmi Koyejo. Are emergent abilities of large language models a mirage? In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. arXiv:2304.15004.

[36] Richard Sutton. The bitter lesson. http://www.incompleteideas.net/IncIdeas/BitterLesson.html, 2019. 13 March 2019; accessed 2026-09-25.

[37] Arihant Tripathy, Ch Pavan Harshit, and Karthik Vaidhyanathan. SWEnergy: An empirical study on energy efficiency in agentic issue resolution frameworks with SLMs. In *Proceedings of the 2026 International Workshop on Agentic Engineering (AGENT 2026)*, pages 104–111. ACM, 2026. doi: 10.1145/3786167.3788406.

[38] Ranit Karmakar and Jayita Chatterjee. AgentFloor: How far up the tool use ladder can small open-weight models go? *arXiv preprint arXiv:2605.00334*, 2026. URL https://arxiv.org/abs/2605.00334.

[39] Esmail Gumaan. Feedback that backfires: Why small language model agents repeat the call they just watched fail. *arXiv preprint arXiv:2608.23651*, 2026. URL https://arxiv.org/abs/2608.23651.

[40] Chenyang Yang, Xinran Zhao, Tongshuang Wu, and Christian Kästner. Better harnesses, smaller models: Building 90% cheaper agents via automated harness adaptation. *arXiv preprint arXiv:2607.08938*, 2026. URL https://arxiv.org/abs/2607.08938.

[41] Linting Xue, Aditya Barua, Noah Constant, Rami Al-Rfou, Sharan Narang, Mihir Kale, Adam Roberts, and Colin Raffel. ByT5: Towards a token-free future with pre-trained byte-to-byte models. *Transactions of the Association for Computational Linguistics*, 10:291–306, 2022. doi: 10.1162/tacl_a_00461.

[42] Jonathan H. Clark, Dan Garrette, Iulia Turc, and John Wieting. CANINE: Pre-training an efficient tokenization-free encoder for language representation. *Transactions of the Association for Computational Linguistics*, 10:73–91, 2022. doi: 10.1162/tacl_a_00448.

[43] Lili Yu, Dániel Simig, Colin Flaherty, Armen Aghajanyan, Luke Zettlemoyer, and Mike Lewis. MEGABYTE: Predicting million-byte sequences with multiscale transformers. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. arXiv:2305.07185.

[44] Junxiong Wang, Tushaar Gangavarapu, Jing Nathan Yan, and Alexander M. Rush. MambaByte: Token-free selective state space model. In *Conference on Language Modeling (COLM)*, 2024. arXiv:2401.13660.

[45] Kevin Slagle. SpaceByte: Towards deleting tokenization from large language modeling. *arXiv preprint arXiv:2404.14408*, 2024. URL https://arxiv.org/abs/2404.14408.

[46] Artidoro Pagnoni, Ram Pasunuru, Pedro Rodriguez, John Nguyen, Benjamin Muller, Margaret Li, Chunting Zhou, Lili Yu, Jason Weston, Luke Zettlemoyer, Gargi Ghosh, Mike Lewis, Ari Holtzman, and Srinivasan Iyer. Byte latent transformer: Patches scale better than tokens. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 9238–9258, 2025. doi: 10.18653/v1/2025.acl-long.453.

[47] Sukjun Hwang, Brandon Wang, and Albert Gu. Dynamic chunking for end-to-end hierarchical sequence modeling. *arXiv preprint arXiv:2507.07955*, 2025. URL https://arxiv.org/abs/2507.07955.

[48] Shangda Wu, Xu Tan, Zili Wang, Rui Wang, Xiaobing Li, and Maosong Sun. Beyond language models: Byte models are digital world simulators. *arXiv preprint arXiv:2402.19155*, 2024. URL https://arxiv.org/abs/2402.19155.

[49] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles Nicholas. Malware detection by eating a whole EXE. *arXiv preprint arXiv:1710.09435*, 2017. URL https://arxiv.org/abs/1710.09435.

[50] Maxwell Horton, Sachin Mehta, Ali Farhadi, and Mohammad Rastegari. Bytes are all you need: Transformers operating directly on file bytes. *Transactions on Machine Learning Research*, 2024. arXiv:2306.00238.

[51] Aleksandar Petrov, Emanuele La Malfa, Philip H. S. Torr, and Adel Bibi. Language model tokenizers introduce unfairness between languages. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. arXiv:2305.15425.

[52] Grégoire Delétang, Anian Ruoss, Paul-Ambroise Duquenne, Elliot Catt, Tim Genewein, Christopher Mattern, Jordi Grau-Moya, Li Kevin Wenliang, Matthew Aitchison, Laurent Orseau, Marcus Hutter, and Joel Veness. Language modeling is compression. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2309.10668.

[53] Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. In *International Conference on Learning Representations (ICLR)*, 2022. arXiv:2111.00396.

[54] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. In *Conference on Language Modeling (COLM)*, 2024. URL https://arxiv.org/abs/2312.00752.

[55] Tri Dao and Albert Gu. Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality. In *International Conference on Machine Learning (ICML)*, 2024. arXiv:2405.21060.

[56] Aakash Lahoti, Kevin Y. Li, Berlin Chen, Caitlin Wang, Aviv Bick, J. Zico Kolter, Tri Dao, and Albert Gu. Mamba-3: Improved sequence modeling using state space principles. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2603.15569.

[57] Samy Jelassi, David Brandfonbrener, Sham M. Kakade, and Eran Malach. Repeat after me: Transformers are better than state space models at copying. In *International Conference on Machine Learning (ICML)*, pages 21502–21521, 2024. arXiv:2402.01032.

[58] Simran Arora, Sabri Eyuboglu, Aman Timalsina, Isys Johnson, Michael Poli, James Zou, Atri Rudra, and Christopher Ré. Zoology: Measuring and improving recall in efficient language models. *arXiv preprint arXiv:2312.04927*, 2023. URL https://arxiv.org/abs/2312.04927.

[59] Roger Waleffe, Wonmin Byeon, Duncan Riach, Brandon Norick, Vijay Korthikanti, Tri Dao, Albert Gu, Ali Hatamizadeh, Sudhakar Singh, Deepak Narayanan, et al. An empirical study of Mamba-based language models. *arXiv preprint arXiv:2406.07887*, 2024. URL https://arxiv.org/abs/2406.07887.

[60] Opher Lieber, Barak Lenz, Hofit Bata, Gal Cohen, Jhonathan Osin, Itay Dalmedigos, Erez Safahi, Shaked Meirom, Yonatan Belinkov, Shai Shalev-Shwartz, et al. Jamba: A hybrid Transformer-Mamba language model. *arXiv preprint arXiv:2403.19887*, 2024. URL https://arxiv.org/abs/2403.19887.

[61] Liliang Ren, Yang Liu, Yadong Lu, Yelong Shen, Chen Liang, and Weizhu Chen. Samba: Simple hybrid state space models for efficient unlimited context language modeling. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2406.07522.

[62] Xin Dong, Yonggan Fu, Shizhe Diao, Wonmin Byeon, Zijia Chen, Ameya Sunil Mahabaleshwarkar, Shih-Yang Liu, Matthijs Van Keirsbilck, Min-Hung Chen, Yoshi Suhara, Yingyan Lin, Jan Kautz, and Pavlo Molchanov. Hymba: A hybrid-head architecture for small language models. *arXiv preprint arXiv:2411.13676*, 2024. URL https://arxiv.org/abs/2411.13676.

[63] NVIDIA, Aaron Blakeman, Aarti Basant, Abhinav Khattar, Adithya Renduchintala, et al. Nemotron-H: A family of accurate and efficient hybrid Mamba-Transformer models. *arXiv preprint arXiv:2504.03624*, 2025. URL https://arxiv.org/abs/2504.03624.

[64] Alexander Amini, Anna Banaszak, Harold Benoit, et al. LFM2 technical report. *arXiv preprint arXiv:2511.23404*, 2025. URL https://arxiv.org/abs/2511.23404.

[65] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. BinaryConnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 3123–3131, 2015. arXiv:1511.00363.

[66] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: ImageNet classification using binary convolutional neural networks. In *European Conference on Computer Vision (ECCV)*, Lecture Notes in Computer Science, pages 525–542, 2016. doi: 10.1007/978-3-319-46493-0_32.

[67] Fengfu Li, Bo Zhang, and Bin Liu. Ternary weight networks. *arXiv preprint arXiv:1605.04711*, 2016. Author list as in v2; v3 (2022) lists Li, Liu, Wang, Zhang and Yan.

[68] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. Trained ternary quantization. In *International Conference on Learning Representations (ICLR)*, 2017. arXiv:1612.01064.

[69] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

[70] Hongyu Wang, Shuming Ma, Li Dong, Shaohan Huang, Huaijie Wang, Lingxiao Ma, Fan Yang, Ruiping Wang, Yi Wu, and Furu Wei. BitNet: Scaling 1-bit transformers for large language models. *arXiv preprint arXiv:2310.11453*, 2023.

[71] Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. The era of 1-bit LLMs: All large language models are in 1.58 bits. *arXiv preprint arXiv:2402.17764*, 2024.

[72] Shuming Ma, Hongyu Wang, Shaohan Huang, Xingxing Zhang, Ying Hu, Ting Song, Yan Xia, and Furu Wei. BitNet b1.58 2B4T technical report. *arXiv preprint arXiv:2504.12285*, 2025.

[73] Jinheng Wang, Hansong Zhou, Ting Song, Shijie Cao, Yan Xia, Ting Cao, Jianyu Wei, Shuming Ma, Hongyu Wang, and Furu Wei. Bitnet.cpp: Efficient edge inference for ternary LLMs. *arXiv preprint arXiv:2502.11880*, 2025.

[74] Jinheng Wang, Hansong Zhou, Ting Song, Shaoguang Mao, Shuming Ma, Hongyu Wang, Yan Xia, and Furu Wei. 1-bit AI infra: Part 1.1, fast and lossless BitNet b1.58 inference on CPUs. *arXiv preprint arXiv:2410.16144*, 2024.

[75] Rui-Jie Zhu, Yu Zhang, Steven Abreu, Ethan Sifferman, Tyler Sheaves, Yiqiao Wang, Dustin Richmond, Sumit Bam Shrestha, Peng Zhou, and Jason K. Eshraghian. Scalable MatMul-free language modeling. *arXiv preprint arXiv:2406.02528*, 2024.

[76] Zhenxuan Yu, Takeshi Kojima, Yutaka Matsuo, and Yusuke Iwasawa. Slender-Mamba: Fully quantized Mamba in 1.58 bits from head to toe. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 4715–4724, 2025.

[77] Shengkun Tang, Liqun Ma, Haonan Li, Mingjie Sun, and Zhiqiang Shen. Bi-Mamba: Towards accurate 1-bit state space models. *Transactions on Machine Learning Research*, 2025. arXiv:2411.11843.

[78] Ramprasath Ganesaraja, Sahil Dilip Panse, and Swathika N. Ternary Mamba: Grouped quantization-aware training of W1.58A16 state space models. *arXiv preprint arXiv:2606.18114*, 2026.

[79] Gabriel Zo-Hasina Rasatavohary. State space models as CPU-native neural network architectures. engrXiv preprint, 2026. Not peer reviewed.

[80] Hung-Yueh Chiang, Chi-Chih Chang, Natalia Frumkin, Kai-Chiang Wu, and Diana Marculescu. Quamba: A post-training quantization recipe for selective state space models. *arXiv preprint arXiv:2410.13229*, 2024.

[81] Xin Cheng et al. Conditional memory via scalable lookup: A new axis of sparsity for large language models, 2026.

[82] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

[83] Ayush Kaushal, Tejas Vaidhya, Arnab Kumar Mondal, Tejas Pandey, Aaryan Bhagat, and Irina Rish. Spectra: Surprising effectiveness of pretraining ternary language models at scale. *arXiv preprint arXiv:2407.12327*, 2024.

[84] Zechun Liu, Changsheng Zhao, Hanxian Huang, Sijia Chen, Jing Zhang, Jiawei Zhao, Scott Roy, Lisa Jin, Yunyang Xiong, Yangyang Shi, et al. ParetoQ: Improving scaling laws in extremely low-bit LLM quantization. In *Advances in Neural Information Processing Systems 38 (NeurIPS 2025)*, pages 101395–101420, 2025. doi: 10.52202/085713-3055.

[85] Gautam Veldanda. Baseline shape decides the verdict: A controlled re-examination of ternary language models at 60K parameters. *arXiv preprint arXiv:2609.29397*, 2026.

[86] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, Pete Warden, and Rocky Rhodes. TensorFlow Lite Micro: Embedded machine learning for TinyML systems. In *Proceedings of Machine Learning and Systems 3 (MLSys 2021)*, 2021. arXiv:2010.08678.

[87] Liangzhen Lai, Naveen Suda, and Vikas Chandra. CMSIS-NN: Efficient neural network kernels for Arm Cortex-M CPUs. *arXiv preprint arXiv:1801.06601*, 2018.

[88] Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. MCUNet: Tiny deep learning on IoT devices. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020. arXiv:2007.10319.

[89] Ji Lin, Wei-Ming Chen, Han Cai, Chuang Gan, and Song Han. MCUNetV2: Memory-efficient patch-based inference for tiny deep learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021. arXiv:2110.15352.

[90] Colby Banbury, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, David Kanter, Sebastian Ahmed, Danilo Pau, et al. MLPerf Tiny benchmark. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021. arXiv:2106.07597.

[91] MLCommons. MLPerf Tiny inference v1.2 results. https://github.com/mlcommons/tiny_results_v1.2, 2024. Keyword-spotting energy files under closed/STMicroelectronics/results/*/kws/energy/results.txt; accessed 2026-09-25.

[92] Moritz Scherer, Luka Macan, Victor Jung, Philip Wiese, Luca Bompani, Alessio Burrello, Francesco Conti, and Luca Benini. Deeploy: Enabling energy-efficient deployment of small language models on heterogeneous microcontrollers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43:4009–4020, 2024. doi: 10.1109/TCAD.2024.3443718.

[93] Nordic Semiconductor. nRF52840. https://www.nordicsemi.com/Products/nRF52840, 2026. Product page; accessed 2026-09-25.

[94] Raspberry Pi Ltd. RP-series microcontrollers (RP2040, RP2350). https://www.raspberrypi.com/documentation/microcontrollers/silicon.html, 2026. Documentation page; accessed 2026-09-25.

[95] Espressif Systems. ESP32-S3. https://www.espressif.com/en/products/socs/esp32-s3, 2026. Product page; accessed 2026-09-25.

[96] NXP Semiconductors. i.MX RT1060 crossover MCU with Arm Cortex-M7 core. https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/i-mx-rt-crossover-mcus/i-mx-rt1060-crossover-mcu-with-arm-cortex-m7-core:i.MX-RT1060, 2026. Product page; accessed 2026-09-25.

[97] Alif Semiconductor. Ensemble E3 series. https://alifsemi.com/ensemble-e3-series/, 2026. Product page; accessed 2026-09-25.

[98] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. In *Network and Distributed System Security Symposium (NDSS)*, 2018. doi: 10.14722/ndss.2018.23204.

[99] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Gianni Antichi, Paolo Costa, Hamed Haddadi, and Roberto Bifulco. Re-architecting traffic analysis with neural network interface cards. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022. URL https://www.usenix.org/conference/nsdi22/presentation/siracusano.

[100] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. Taurus: A data plane architecture for per-packet ML. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1099–1114, 2022. doi: 10.1145/3503222.3507726.

[101] Daniele Palossi, Antonio Loquercio, Francesco Conti, Eric Flamand, Davide Scaramuzza, and Luca Benini. A 64-mW DNN-based visual navigation engine for autonomous nano-drones. *IEEE Internet of Things Journal*, 6(5):8357–8371, 2019. doi: 10.1109/JIOT.2019.2917066.

[102] Bardienus P. Duisterhof, Srivatsan Krishnan, Jonathan J. Cruz, Colby R. Banbury, William Fu, Aleksandra Faust, Guido C. H. E. de Croon, and Vijay Janapa Reddi. Learning to seek: Autonomous source seeking with deep reinforcement learning onboard a nano drone microcontroller. *arXiv preprint arXiv:1909.11236*, 2019.

[103] Sabrina M. Neuman, Brian Plancher, Bardienus P. Duisterhof, Srivatsan Krishnan, Colby Banbury, Mark Mazumder, Shvetank Prakash, Jason Jabbour, Aleksandra Faust, Guido C. H. E. de Croon, and Vijay Janapa Reddi. Tiny robot learning: Challenges and directions for machine learning in resource-constrained robots. In *IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 2022. arXiv:2205.05748.

[104] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 3645–3650, 2019. doi: 10.18653/v1/P19-1355.

[105] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. Green AI. *Communications of the ACM*, 63:54–63, 2020. doi: 10.1145/3381831.

[106] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*, 2021.

[107] Cooper Elsworth, Keguo Huang, David Patterson, Ian Schneider, Robert Sedivy, Savannah Goodman, Ben Townsend, Parthasarathy Ranganathan, Jeff Dean, Amin Vahdat, Ben Gomes, and James Manyika. Measuring the environmental impact of delivering AI at Google scale. *arXiv preprint arXiv:2508.15734*, 2025.

[108] Sam Altman. The gentle singularity. https://blog.samaltman.com/the-gentle-singularity, 2025. 10 June 2025; accessed 2026-09-25.

[109] Josh You. How much energy does ChatGPT use? Epoch AI, https://epoch.ai/gradient-updates/how-much-energy-does-chatgpt-use, 2025. 7 February 2025; accessed 2026-09-25.

[110] International Energy Agency. Energy and AI. https://www.iea.org/reports/energy-and-ai, 2025. April 2025; accessed 2026-09-25.

[111] Ben Cottier, Ben Snodin, David Owen, and Tom Adamczewski. LLM inference prices have fallen rapidly but unequally across tasks. Epoch AI, https://epoch.ai/data-insights/llm-inference-price-trends, 2025. 12 March 2025; accessed 2026-09-25.

[112] Guido Appenzeller. Welcome to LLMflation – LLM inference cost is going down fast. a16z, https://a16z.com/llmflation-llm-inference-cost/, 2024. 12 November 2024; accessed 2026-09-25.

[113] Nidhal Jegham, Marwan Abdelatti, Chan Young Koh, Lassad Elmoubarki, and Abdeltawab Hendawi. How hungry is AI? benchmarking energy, water, and carbon footprint of LLM inference. *arXiv preprint arXiv:2505.09598*, 2025.

[114] Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. MLAgentBench: Evaluating language agents on machine learning experimentation. In *International Conference on Machine Learning (ICML)*, pages 20271–20309, 2024. arXiv:2310.03302.

[115] Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. AIDE: AI-driven exploration in the space of code. *arXiv preprint arXiv:2502.13138*, 2025.

[116] Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Mądry. MLE-bench: Evaluating machine learning agents on machine learning engineering. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2410.07095.

[117] Ziming Li, Qianbo Zang, David Ma, Jiawei Guo, Tuney Zheng, Minghao Liu, Xinyao Niu, Yue Wang, Jian Yang, Jiaheng Liu, et al. AutoKaggle: A multi-agent framework for autonomous data science competitions. *arXiv preprint arXiv:2410.20424*, 2024.

[118] Patara Trirat, Wonyong Jeong, and Sung Ju Hwang. AutoML-Agent: A multi-agent LLM framework for full-pipeline AutoML. In *International Conference on Machine Learning (ICML)*, 2025. arXiv:2410.02958.

[119] Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The AI scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*, 2024.

[120] Andrej Karpathy. autoresearch: AI agents running research on single-GPU nanochat training automatically. https://github.com/karpathy/autoresearch, 2026. Repository created 6 March 2026; accessed 2026-09-25.

[121] Hjalmar Wijk, Tao Lin, Joel Becker, Sami Jawhar, Neev Parikh, Thomas Broadley, Lawrence Chan, Michael Chen, Josh Clymer, Jai Dhyani, et al. RE-Bench: Evaluating frontier AI R&D capabilities of language model agents against human experts. *arXiv preprint arXiv:2411.15114*, 2024.

[122] Deepak Nathani, Lovish Madaan, Nicholas Roberts, Nikolay Bashlykov, Ajay Menon, Vincent Moens, Amar Budhiraja, Despoina Magka, Vladislav Vorotilov, Gaurav Chaurasia, et al. MLGym: A new framework and benchmark for advancing AI research agents. *arXiv preprint arXiv:2502.14499*, 2025.

[123] Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, et al. PaperBench: Evaluating AI's ability to replicate AI research. In *International Conference on Machine Learning (ICML)*, 2025. arXiv:2504.01848.

[124] Wentao Zhang, Liliana Hotsko, Woojeong Kim, Pengyu Nie, Stuart Shieber, and Yuntian Deng. Program-as-weights: A programming paradigm for fuzzy functions. *arXiv preprint arXiv:2607.02512*, 2026. URL https://arxiv.org/abs/2607.02512.

[125] Dhruv Atreja, Julia White, Nikhil Nayak, Kelton Zhang, Henrijs Princis, George Hurn-Maloney, Ash Lewis, and Urchade Zaratiana. Pioneer agent: Continual improvement of small language models in production. *arXiv preprint arXiv:2604.09791*, 2026.

[126] Adel ElZemity, Joshua Sylvester, Budi Arief, and Rogério De Lemos. Agentic knowledge distillation: Autonomous training of small language models for SMS threat detection. *arXiv preprint arXiv:2602.10869*, 2026.

[127] Penglin Dai, Zijie Zhou, Xincao Xu, Junhua Wang, Xiao Wu, and Lixin Duan. AutoMCU: Feasibility-first MCU neural network customization via LLM-based multi-agent systems. *arXiv preprint arXiv:2605.21560*, 2026.

[128] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors. *Automated Machine Learning: Methods, Systems, Challenges.* The Springer Series on Challenges in Machine Learning. Springer, 2019. doi: 10.1007/978-3-030-05318-5.

[129] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20:1–21, 2019.

[130] Alexander Tornede, Difan Deng, Theresa Eimer, Joseph Giovanelli, Aditya Mohan, Tim Ruhkopf, Sarah Segel, Daphne Theodorakopoulos, Tanja Tornede, Henning Wachsmuth, and Marius Lindauer. AutoML in the age of large language models: Current challenges, future opportunities and risks. *Transactions on Machine Learning Research*, 2024. arXiv:2306.08107.

[131] Alexander Ratner, Christopher De Sa, Sen Wu, Daniel Selsam, and Christopher Ré. Data programming: Creating large training sets, quickly. In *Advances in Neural Information Processing Systems 29 (NeurIPS 2016)*, pages 3567–3575, 2016. arXiv:1605.07723.

[132] Jiacheng Ye, Jiahui Gao, Qintong Li, Hang Xu, Jiangtao Feng, Zhiyong Wu, Tao Yu, and Lingpeng Kong. ZeroGen: Efficient zero-shot learning via dataset generation. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 11653–11669, 2022. doi: 10.18653/v1/2022.emnlp-main.801.

[133] Andrej Karpathy. Software 2.0. Medium, https://karpathy.medium.com/software-2-0-a64152b37c35, 2017. 11 November 2017; accessed 2026-09-25.

[134] Yuyao Ge, Lingrui Mei, Zenghao Duan, Tianhao Li, Yujia Zheng, Yiwei Wang, Lexin Wang, Jiayu Yao, Tianyu Liu, Yujun Cai, et al. A survey of vibe coding with large language models. *arXiv preprint arXiv:2510.12399*, 2025.

[135] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*, pages 68539–68551, 2023. doi: 10.52202/075280-2997.

[136] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive APIs. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, pages 126544–126565, 2024. doi: 10.52202/079017-4020.

[137] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. HuggingGPT: Solving AI tasks with ChatGPT and its friends in Hugging Face. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. arXiv:2303.17580.

[138] Anthropic. Introducing the Model Context Protocol. https://www.anthropic.com/news/model-context-protocol, 2024. 25 November 2024; accessed 2026-09-25.

[139] Wei Chen and Zhiyuan Li. Octopus v2: On-device language model for super agent. *arXiv preprint arXiv:2404.01744*, 2024.

[140] Lutfi Eren Erdogan, Nicholas Lee, Siddharth Jha, Sehoon Kim, Ryan Tabrizi, Suhong Moon, Coleman Hooper, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. TinyAgent: Function calling at the edge. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 80–88, 2024. doi: 10.18653/v1/2024.emnlp-demo.9.

[141] Henry Ndubuaku, Karen Mosoyan, Jakub Mroz, Noah Cylich, Satyajit Kumar, Parkirat Sandhu, Roman Shemet, and Justin H. Lee. Needle: Automation foundation model for tiny devices. https://github.com/cactus-compute/needle, 2026. Apache-2.0; accessed 2026-09-25.

[142] Michael Meli, Matthew R. McNiece, and Bradley Reaves. How bad can it git? Characterizing secret leakage in public GitHub repositories. In *Network and Distributed System Security Symposium (NDSS)*, 2019. doi: 10.14722/ndss.2019.23418.

[143] GitGuardian. The state of secrets sprawl 2026. https://www.gitguardian.com/state-of-secrets-sprawl-report-2026, 2026. 5th edition; accessed 2026-09-25.

[144] Markus Dahlmanns, Constantin Sander, Robin Decker, and Klaus Wehrle. Secrets revealed in container images: An internet-wide study on occurrence and impact. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security (ASIA CCS)*, pages 797–811, 2023. doi: 10.1145/3579856.3590329.

[145] Gitleaks contributors. Gitleaks, version 8.30.1. https://github.com/gitleaks/gitleaks, 2026. Accessed 2026-09-25.

[146] Truffle Security. TruffleHog, version 3.97.6. https://github.com/trufflesecurity/trufflehog, 2026. Accessed 2026-09-25.

[147] Yelp. detect-secrets, version 1.5.0. https://github.com/Yelp/detect-secrets, 2024. Accessed 2026-09-25.

[148] Samsung. CredSweeper, version 1.18.5. https://github.com/Samsung/CredSweeper, 2026. Accessed 2026-09-25.

[149] Setu Kumar Basak, Jamison Cox, Bradley Reaves, and Laurie Williams. A comparative study of software secrets reporting by secret detection tools. In *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, 2023. doi: 10.1109/ESEM56168.2023.10304853.

[150] JaeKu Yun, ShinHyung Choi, YuJeong Lee, Oleksandra Sokol, WooChul Shim, Arkadiy Melkonyan, and Dmytro Kuzmenko. Project CredData: A dataset of credentials for research. https://github.com/Samsung/CredData, 2021. Samsung; accessed 2026-09-25.

[151] Aakanksha Saha, Tamara Denning, Vivek Srikumar, and Sneha Kumar Kasera. Secrets in source code: Reducing false positives using machine learning. In *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*, pages 168–175, 2020. doi: 10.1109/COMSNETS48256.2020.9027350.

[152] Md Nafiu Rahman, Sadif Ahmed, Zahin Wahab, S M Sohan, and Rifat Shahriyar. Secret breach detection in source code with large language models. In *2025 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 207–217, 2025. doi: 10.1109/ESEM64174.2025.00021.

[153] Sadif Ahmed, Md Nafiu Rahman, Zahin Wahab, Gias Uddin, and Rifat Shahriyar. Secret leak detection in software issue reports using LLMs: A comprehensive evaluation. In *Proceedings of the 23rd International Conference on Mining Software Repositories (MSR)*, pages 169–180, 2026. doi: 10.1145/3793302.3793348.

[154] Zhengdong Huang, Kevin Li, Jinqiu Yang, Yepang Liu, and Lili Wei. Checked-in secret detection: Strings are all you need. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2026. arXiv:2608.04523; DOI 10.1145/3832207 listed by arXiv, not yet resolvable at Crossref on 2026-09-25.

[155] Setu Kumar Basak, Lorenzo Neil, Bradley Reaves, and Laurie Williams. SecretBench: A dataset of software secrets. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 347–351, 2023. doi: 10.1109/MSR59073.2023.00053.

[156] Microsoft Presidio contributors. Presidio: data protection and de-identification SDK, version 2.2.364. https://github.com/microsoft/presidio (redirects to https://github.com/data-privacy-stack/presidio), 2026. Accessed 2026-09-25.

[157] Urchade Zaratiana, Nadi Tomeh, Pierre Holat, and Thierry Charnois. GLiNER: Generalist model for named entity recognition using bidirectional transformer. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 5364–5376, 2024. doi: 10.18653/v1/2024.naacl-long.300.

[158] Urchade Zaratiana, Gil Pasternak, Oliver Boyd, George Hurn-Maloney, and Ash Lewis. GLiNER2: An efficient multi-task information extraction system with schema-driven interface. *arXiv preprint arXiv:2507.18546*, 2025.

[159] Urchade Zaratiana, Ash Lewis, and George Hurn-Maloney. GLiNER2-PII: A multilingual model for personally identifiable information extraction. *arXiv preprint arXiv:2605.09973*, 2026.

[160] OpenAI. OpenAI Privacy Filter. https://huggingface.co/openai/privacy-filter, 2026. Apache-2.0; accessed 2026-09-25.

[161] PII Masking Benchmark authors. PII Masking Benchmark (PIIMB). https://huggingface.co/datasets/piimb/pii-masking-benchmark, 2026. CC BY-NC 4.0; accessed 2026-09-25.

[162] Nils Lukas, Ahmed Salem, Robert Sim, Shruti Tople, Lukas Wutschitz, and Santiago Zanella-Béguelin. Analyzing leakage of personally identifiable information in language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 346–363, 2023. doi: 10.1109/SP46215.2023.10179300.

[163] Jose D. Posada, David Love, Somalee Datta, and Priya Desai. SHIELD: A diverse clinical note dataset and distilled small language models for enterprise-scale de-identification. *arXiv preprint arXiv:2605.03301*, 2026.

[164] Arm Holdings plc. Annual report on Form 20-F for the fiscal year ended march 31, 2026. U.S. Securities and Exchange Commission, https://www.sec.gov/Archives/edgar/data/1973239/000197323926000097/arm-20260331.htm, 2026. Filed 26 May 2026; accessed 2026-09-25.

[165] Steve Furber. Microprocessors: the engines of the digital age. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 473(2199):20160893, 2017. doi: 10.1098/rspa.2016.0893.

[166] Arm Holdings plc. Prospectus (Form 424B4). U.S. Securities and Exchange Commission, https://www.sec.gov/Archives/edgar/data/1973239/000119312523235320/d550931d424b4.htm, 2023. Filed 14 September 2023; accessed 2026-09-25.

[167] Krste Asanović and David A. Patterson. Instruction sets should be free: The case for RISC-V. Technical Report UCB/EECS-2014-146, EECS Department, University of California, Berkeley, 2014. URL https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html.

[168] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62:48–60, 2019. doi: 10.1145/3282307.

[169] Joshua Saxe and Konstantin Berlin. eXpose: A character-level convolutional neural network with embeddings for detecting malicious URLs, file paths and registry keys, 2017.

[170] Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *Advances in Neural Information Processing Systems 28*, 2015.

[171] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, pages 328–339, 2018.

[172] Dan Hendrycks, Xiaoyuan Liu, Eric Wallace, Adam Dziedzic, Rishabh Krishnan, and Dawn Song. Pretrained transformers improve out-of-distribution robustness. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020.

[173] Pengcheng He, Jianfeng Gao, and Weizhu Chen. DeBERTaV3: Improving DeBERTa using ELECTRA-style pre-training with gradient-disentangled embedding sharing. In *International Conference on Learning Representations*, 2023.

[174] OpenMed. OpenMed-PII-SuperClinical-Small-44M-v1 (model card). https://huggingface.co/OpenMed/OpenMed-PII-SuperClinical-Small-44M-v1, 2025. Fine-tuned from DeBERTa-v3-small on Nemotron-PII; accessed 25 September 2026.

[175] Orion Weller, Kathryn Ricci, Marc Marone, Antoine Chaffin, Dawn Lawrie, and Benjamin Van Durme. Seq vs seq: An open suite of paired encoders and decoders. In *International Conference on Learning Representations*, 2026.

[176] kalyan-ks. ettin-68m-nemotron-pii (model card). https://huggingface.co/kalyan-ks/ettin-68m-nemotron-pii, 2025. Fine-tuned from Ettin-encoder-68M on Nemotron-PII; accessed 25 September 2026.

[177] Jared Fernandez, Clara Na, Vashisth Tiwari, Yonatan Bisk, Sasha Luccioni, and Emma Strubell. Energy considerations of large language model inference and efficiency optimizations. *arXiv preprint arXiv:2504.17674*, 2025.

[178] NVIDIA. NVIDIA H100 tensor core GPU. https://www.nvidia.com/en-us/data-center/h100/, 2023. H100 SXM: 1,979 BF16 teraFLOPS with sparsity (about 990 dense), up to 700 W; accessed 25 September 2026.

[179] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020. doi: 10.1038/s41586-020-2649-2.

[180] IEEE. IEEE standard for floating-point arithmetic. IEEE Std 754-2019, 2019.

[181] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th International Conference on Machine Learning*, pages 1113–1120, 2009.

[182] Dan Svenstrup, Jonas Meinertz Hansen, and Ole Winther. Hash embeddings for efficient word representations. In *Advances in Neural Information Processing Systems*, 2017.

[183] Biao Zhang and Rico Sennrich. Root mean square layer normalization. In *Advances in Neural Information Processing Systems*, 2019.

[184] Lianghui Zhu, Bencheng Liao, Qian Zhang, Xinlong Wang, Wenyu Liu, and Xinggang Wang. Vision Mamba: Efficient visual representation learning with bidirectional state space model. In *Proceedings of the 41st International Conference on Machine Learning*, 2024.

[185] Sukjun Hwang, Aakash Lahoti, Tri Dao, and Albert Gu. Hydra: Bidirectional state space models through generalized matrix mixers. In *Advances in Neural Information Processing Systems 37*, 2024.

[186] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017.

[187] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. RoFormer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024. doi: 10.1016/j.neucom.2023.127063.

[188] Wenzhi Cao, Vahid Mirjalili, and Sebastian Raschka. Rank consistent ordinal regression for neural networks with application to age estimation. *Pattern Recognition Letters*, 140:325–331, 2020. doi: 10.1016/j.patrec.2020.11.008.

[189] Lev Ratinov and Dan Roth. Design challenges and misconceptions in named entity recognition. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning (CoNLL)*, pages 147–155, 2009.

[190] Andrew J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, 1967. doi: 10.1109/TIT.1967.1054010.

[191] ggml contributors. GGUF: the file format of the ggml project. https://github.com/ggml-org/ggml/blob/master/docs/gguf.md, 2023. Accessed 2026-09-25.

[192] Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving Mamba2 with delta rule. In *International Conference on Learning Representations*, 2025.

[193] Sylvie Boldo and Guillaume Melquiond. Emulation of FMA and correctly rounded sums: Proved algorithms using rounding to odd. *IEEE Transactions on Computers*, 57(4):462–471, 2008. doi: 10.1109/TC.2007.70819.

[194] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, MA, third edition, 1997.

[195] Stephen L. B. Moshier. *Methods and Programs for Mathematical Functions*. Ellis Horwood, Chichester, 1989.

[196] Convai Innovations. Laya: a non-generative decision model (model card). https://huggingface.co/convaiinnovations/laya, 2026. Package `laya` 0.3.6 measured; model card accessed 25 September 2026.

[197] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, 2019.

[198] Sayash Kapoor and Arvind Narayanan. Leakage and the reproducibility crisis in machine-learning-based science. *Patterns*, 4(9):100804, 2023. doi: 10.1016/j.patter.2023.100804.

[199] Udi Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 1–10, 1994.

[200] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019.

[201] Shengding Hu, Yuge Tu, Xu Han, Chaoqun He, Ganqu Cui, Xiang Long, Zhi Zheng, Yewei Fang, Yuxiang Huang, Weilin Zhao, et al. MiniCPM: Unveiling the potential of small language models with scalable training strategies. *arXiv preprint arXiv:2404.06395*, 2024. URL https://arxiv.org/abs/2404.06395.

[202] Jesse Dodge, Gabriel Ilharco, Roy Schwartz, Ali Farhadi, Hannaneh Hajishirzi, and Noah Smith. Fine-tuning pretrained language models: Weight initializations, data orders, and early stopping, 2020.

[203] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. On calibration of modern neural networks. In *Proceedings of the 34th International Conference on Machine Learning*, pages 1321–1330, 2017.

[204] Brian A. Nosek, Charles R. Ebersole, Alexander C. DeHaven, and David T. Mellor. The preregistration revolution. *Proceedings of the National Academy of Sciences*, 115(11):2600–2606, 2018. doi: 10.1073/pnas.1708274114.

[205] Margaret Mitchell, Simone Wu, Andrew Zaldivar, Parker Barnes, Lucy Vasserman, Ben Hutchinson, Elena Spitzer, Inioluwa Deborah Raji, and Timnit Gebru. Model cards for model reporting. In *Proceedings of the Conference on Fairness, Accountability, and Transparency (FAT*)*, pages 220–229, 2019. doi: 10.1145/3287560.3287596.

[206] Tom Gunter et al. Apple intelligence foundation language models, 2024.

[207] Horace He and Thinking Machines Lab. Defeating nondeterminism in LLM inference. Thinking Machines Lab: Connectionism, https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/, 2025. 10 September 2025.

[208] Cynthia Dwork, Vitaly Feldman, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Aaron Roth. The reusable holdout: Preserving validity in adaptive data analysis. *Science*, 349(6248):636–638, 2015. doi: 10.1126/science.aaa9375.

[209] Ai4Privacy. OpenPII 1M: Multilingual PII masking dataset (19 labels, 23 languages). https://huggingface.co/datasets/ai4privacy/pii-masking-openpii-1m, 2026. Ai4Privacy / Ai Suisse SA; CC BY 4.0; accessed 2026-09-25.

[210] NVIDIA. Nemotron-PII: Synthesized data for privacy-preserving AI. https://huggingface.co/datasets/nvidia/Nemotron-PII, 2025. CC BY 4.0; accessed 2026-09-25.

[211] Gretel.ai. Gretel PII masking EN v1. https://huggingface.co/datasets/gretelai/gretel-pii-masking-en-v1, 2024. Apache-2.0; accessed 2026-09-25.

[212] Gretel.ai. Synthetic PII finance multilingual. https://huggingface.co/datasets/gretelai/synthetic_pii_finance_multilingual, 2024. Apache-2.0; accessed 2026-09-25.

[213] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with Gumbel-Softmax, 2016.

[214] Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. The Concrete distribution: A continuous relaxation of discrete random variables, 2016.

[215] Weizhi Wang, Li Dong, Hao Cheng, Xiaodong Liu, Xifeng Yan, Jianfeng Gao, and Furu Wei. Augmenting language models with long-term memory. In *Advances in Neural Information Processing Systems*, 2023.

[216] Yi-Lin Sung, Jaemin Cho, and Mohit Bansal. LST: Ladder side-tuning for parameter and memory efficient transfer learning. In *Advances in Neural Information Processing Systems*, 2022.

[217] Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Generalization through memorization: Nearest neighbor language models. In *International Conference on Learning Representations*, 2020.

[218] Yuhuai Wu, Markus N. Rabe, DeLesley Hutchins, and Christian Szegedy. Memorizing transformers. In *International Conference on Learning Representations*, 2022.

[219] Jiacheng Liu, Sewon Min, Luke Zettlemoyer, Yejin Choi, and Hannaneh Hajishirzi. Infini-gram: Scaling unbounded *n*-gram language models to a trillion tokens. In *Conference on Language Modeling*, 2024.

[220] Ekin Akyürek, Bailin Wang, Yoon Kim, and Jacob Andreas. In-context language learning: Architectures and algorithms. In *Proceedings of the 41st International Conference on Machine Learning*, 2024.

[221] Catherine Olsson, Nelson Elhage, Neel Nanda, Nicholas Joseph, Nova DasSarma, et al. In-context learning and induction heads, 2022.

[222] Matthew V. Mahoney. Adaptive weighing of context models for lossless data compression. Technical Report CS-2005-16, Florida Institute of Technology, 2005.

[223] Szymon Tworkowski, Konrad Staniszewski, Mikołaj Pacek, Yuhuai Wu, Henryk Michalewski, and Piotr Miłoś. Focused transformer: Contrastive training for context scaling. In *Advances in Neural Information Processing Systems*, 2023.

[224] Jon Saad-Falcon, Avanika Narayan, Hakki Orhun Akengin, J. Wes Griffin, Herumb Shandilya, Adrian Gamarra Lafuente, Medhya Goel, Rebecca Joseph, Shlok Natarajan, Etash Kumar Guha, Shang Zhu, Ben Athiwaratkun, John Hennessy, Azalia Mirhoseini, and Christopher Ré. Intelligence per watt: Measuring intelligence efficiency of local AI. *arXiv preprint arXiv:2511.07885*, 2025.

[225] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In *Advances in Neural Information Processing Systems 28*, 2015.

[226] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics*, 2017.

[227] Instituto Nacional de Estadística. Nombres y apellidos: frecuencias. https://www.ine.es/daco/daco42/nombyapel/, 2026. CC BY 4.0; accessed September 2026.

[228] Tatoeba contributors. Tatoeba: a collection of sentences and translations. https://tatoeba.org, 2026. CC BY 2.0 FR; accessed September 2026.

Table 17: Three ways to automate a byte-level job. Measured values are from the runtime's published comparisons (same desktop CPU unless stated; conditions in Sections 5 and 8; CredData figures: its test half without the four repositories that overlap the training corpus, gitleaks with its default rules); qualitative rows are properties of the approach. The measured values of column (a) are those of gitleaks' default rules, a mature community rule set, not of rules written by an agent for the job, which we did not measure.

	(a) Rules	(b) A general model at run time	(c) A trained AI Specialist
Cost to build	low, grows with every exception	a prompt, and an examination to know its quality	data, criteria, one training run (25 minutes on one consumer GPU for three seeds of `secrets-code`)
Time per decision	microseconds (gitleaks reads bytes roughly 80 times faster than `secrets-code`)	measured: Laya, a 421 M-parameter decision model, 373 ms on the same CPU; quoted by Laya's model card: 33–40 ms on a T4 GPU, and a hosted decision API 236–276 ms (third-party median)	0.82 ms for a 64-byte input on eight threads of a desktop CPU
Where the data go	nowhere	to the provider if hosted; nowhere if local	nowhere
Same input, same answer	yes	not guaranteed by hosted APIs; reproducible locally with greedy decoding and batch-invariant kernels [207]	yes, bit for bit on a platform (Section 5)
Patterns without a fixed form	weak: gitleaks recall 0.207 overall on CredData (0.201 on its original values)	strong in principle; not measured here	strong: `secrets-code` recall 0.729 overall at precision 0.879 (0.583 on the original values)
Patterns with a fixed form	strong for private keys (recall 0.84), weak for provider tokens with default rules (0.28, a gap not explained, Section 8.2)	strong in principle; not measured here	strong (0.90 and 0.85)
A new question	a new program	a new prompt	a new training run

Table 18: The three released AI Specialists. Parameters without heads: the network (byte embedding, blocks and final normalization) plus the *n*-gram tables and their projection where there are any; the two heads add 34,439 to each (Table 6).

	`secrets-code` 1.0.0	`pii` 1.0.0	`secrets-bin` 1.0.0
Job	credentials in code, configuration and other text	personal data in any text, masked as one type	credentials in executables, libraries, bytecode, firmware
Input	text files up to 4,000,000 bytes; under 24 bytes not analyzed	any file up to 64 MiB, of any length	any file up to 64 MiB; archives opened; under 24 bytes not analyzed
Preset; blocks of width 256	`nano`; 4 `ssm_v2`	`nano_n15`; 4 `ssm_v2`	`tiny_n18`; 8 `ssm_v2`
n-gram tables (4-bit)	none	$3 \times 2^{15} \times 64$	$3 \times 2^{18} \times 64$
Parameters: without heads; with heads	1,927,520; 1,961,959	8,268,128; 8,302,567	54,170,048; 54,204,487
Model file	1,028,704 bytes (0.98 MiB)	4,764,864 bytes (4.5 MiB)	30,163,488 bytes (28.8 MiB)
Windows and heads	512 bytes every 384; a two-class window head with max pooling gates a span head with BIOES labels of one type (`secret`, `PII`, `secret`)		
Stored bias; profile	0.0; `secrets-code`	−0.5; `redact`	0.0; `secrets-binary`
Seeds + controls; released seed	3 + 1; fewest findings on `code-fp-b`, CredData TEST in view	6 + 2; best F1 on *select*	6 + 1; best F1 on *select*
Headline	CredData F1 0.797 [0.719–0.860] on 164 of 168 TEST repositories, obfuscated values (0.694 on the original values)	PIIMB mean F2 0.7689; four of six tasks from its training sources	false alarms on 193 clean files: 1, 9, 12, 4 and 2; synthetic probe recall 1.000 (360 of 360)

Table 19: Exams that also informed development decisions. Their figures in this section are partly in-sample.

Exam	What it also decided during development
CredData TEST (complete half)	the data steps of the `secrets-code` recipe, scored at each of its last four versions against per-family recall targets (short passwords, `Authorization` headers, URL credentials, values with no name); the rejection of a recalibration of confidences (Section 6.7); the released seed, chosen with TEST in view
`code-fp-b`	the learning rate and the rejection of a pretrained body (Section 6.5); the rejection of a late attention layer and of larger bodies (Section 4.11); the released seed of `secrets-code`
`bin-fp-a`	the *n*-gram memory and the cost of ternary weights (Section 4.11); the rate of the output projections, against tail seeds (Section 6.5)
`bin-fp-b`, Windows part	the share of real negatives in each batch (Section 6.5)
`bin-fp-b`, `bin-fp-c`	chosen after earlier models failed on them; the recipe's negatives were built to fix those failures, and the profile's rule for JAR digests was written after `bin-fp-c` showed them
Binary probe	the recall of development models, among them those of the *n*-gram ablation; a seed chosen on it was the worst on real libraries (Section 8.5)
PIIMB	the direction of the body, the preset and the size of the public pool (Section 8.5); the data mix, compared on it (Section 8.3)

Table 20: `secrets-code` on CredData TEST (commit `c09c0c5`, obfuscated values): the published scope without the four repositories that overlap the training corpus, and the complete TEST half for reference. Files up to 4 MB; the CredData line rule, on labeled lines only; F1 with its 95 % bootstrap interval over repositories, which covers the sampling of repositories, not the variation between training runs (see the three seeds' row). All tools run by us on the same machine (AMD Ryzen 9 5900X, Linux under WSL2, 6 threads per tool) and scored by the same code: the model at its stored operating point and the ensemble with two of three votes, on regions of the files around the labels that give them the same windows as the whole files (checked on a sample of files); gitleaks with its default rules and TruffleHog without verification, on the whole files.

Tool	Found (TP)	False pos.	Missed	Precision	Recall	F1 [95 % interval]
Published scope: 164 repositories, 4,737 files, 4,219 positive line units						
`secrets-code` 1.0.0	3,077	422	1,142	0.879	0.729	**0.797** [0.719–0.860]
`secrets-code`, ensemble of three	3,128	429	1,091	0.879	0.741	0.805 [0.731–0.863]
`secrets-code`, the recipe's three seeds	–	–	–	–	–	range 0.788–0.799
gitleaks 8.30.1	875	92	3,344	0.905	0.207	0.337 [0.263–0.421]
TruffleHog 3.97.6, no verification	104	76	4,115	0.578	0.025	0.047 [0.017–0.092]
Complete TEST half, for reference: 168 repositories, 5,262 files, 5,295 positive line units						
`secrets-code` 1.0.0	3,491	492	1,804	0.876	0.659	0.753 [0.665–0.841]
`secrets-code`, ensemble of three	3,564	545	1,731	0.867	0.673	0.758 [0.675–0.844]
gitleaks 8.30.1	1,119	100	4,176	0.918	0.211	0.344 [0.263–0.413]
TruffleHog 3.97.6, no verification	106	86	5,189	0.552	0.020	0.039 [0.014–0.078]

Table 21: `secrets-code` beyond the headline. (a) Operating points, chosen on the 51 DEV repositories that fed no training window, before the released model was scored on TEST; TEST is the published scope. (b) False alarms on public repositories without live credentials, pinned by commit, scanned by the release's CLI with default options (one model, 6 threads): every finding counts; warnings are findings in test, example and documentation paths; errors are the rest. The indented rows leave out the nine repositories that were in the training data (see the text). `code-fp-b` also chose the learning rate and the released seed, so its counts are development figures; gitleaks and TruffleHog were not run on these sets.

(a) Operating points

Point	Rule on DEV	DEV P / R	TEST P / R / F1 [95 % interval]
`--bias 0` (default)	the best F1	0.876 / 0.728	0.879 / 0.729 / 0.797 [0.719–0.860]
`--bias -2`	the highest recall with precision ≥ 0.90	0.920 / 0.662	0.932 / 0.663 / 0.775 [0.683–0.848]

(b) False alarms on repositories without live credentials

Set	Repositories	Files scanned	Findings	Warnings	Errors
`code-fp-a`	6	1,219	93	90	3
without tornado	5	931	73	70	3
`code-fp-b`	18	4,823	408	392	16
without starlette, got and devise	15	4,322	339	323	16
`code-fp-c`	12	4,778	138	49	89
without the five overlapping repositories	7	2,046	51	27	24
All three	36	10,820	639	531	108
without the nine overlapping repositories	27	7,299	463	420	43

Table 22: CredData TEST, published scope (164 repositories, 4,219 labeled credential lines), scanned as distributed (obfuscated values) and with the original values restored. Paired 95 % interval of the change in recall, bootstrap over repositories.

Tool	Recall, obfuscated → original [change]	Found	F1
`secrets-code` 1.0.0	0.729 → 0.583 [−0.189, −0.114]	3,077 → 2,460	0.797 → 0.694
Ensemble of three	0.741 → 0.597 [−0.187, −0.115]	3,128 → 2,517	0.805 → 0.704
gitleaks 8.30.1	0.207 → 0.201 [−0.027, +0.019]	875 → 850	0.337 → 0.329
trufflehog 3.97.6	0.025 → 0.046 [+0.004, +0.036]	104 → 195	0.047 → 0.087

Table 23: `pii` on PIIMB (revision `7d797b9f`, `sentences` subset, 150,022 sentences): F2 per task and the means over the six tasks of F2, precision and recall. `pii` measured by us with the `purebyte` CLI 1.0.0 (AMD Ryzen 9 5900X, Windows 10, 3 threads) at the operating point stored in its file, together with a file of each of the recipe's six seeds; "PIIMB mode" (the released seed at bias 2.0, gate off) is computed in PyTorch; the criterion as first written was read there, and since its move it is reported, not judged. The last column leaves out the 1,018 sentences that share a 64-byte block with the training pool (recall not reported). openai/privacy-filter and the leader (the leaderboard's first, OpenMed/privacy-filter-multilingual-v2, a fine-tune of it), both of 1.4 B parameters with about 50 M active per token, are quoted from the public leaderboard (retrieved 2026-09-23).

Task	`pii` 1.0.0 (released)	six seeds mean	six seeds min–max	PIIMB mode	openai/ privacy-filter	leader	masking everything	`pii` 1.0.0, no overlap
ai4privacy-en	0.9569	0.9624	0.9569–0.9643	0.9716	0.8426	0.9754	0.5883	0.9568
ai4privacy-multi	0.9498	0.9569	0.9498–0.9602	0.9686	0.8052	0.9701	0.5867	0.9495
nemotron-pii	0.8918	0.9003	0.8918–0.9050	0.9334	0.6068	0.9070	0.4300	0.8902
gretel	0.9463	0.9522	0.9463–0.9563	0.9622	0.8687	0.9727	0.6443	0.9464
privy	0.5360	0.5183	0.4834–0.5462	0.6694	0.5362	0.8176	0.2679	0.5360
mapa-eur-lex	0.3325	0.3744	0.3325–0.4002	0.4069	0.3129	0.3728	0.1364	0.3325
Mean F2	**0.7689**	0.7774	0.7689–0.7853	0.8187	0.6621	0.8359	0.4423	0.7686
Mean precision	0.773	0.765	0.759–0.773	0.680	0.751	0.832	0.157	0.773
Mean recall	0.770	0.782	0.769–0.794	0.892	0.655	0.839	1.000	–

Table 24: `secrets-bin` on binaries that hold no credential (every finding is a false alarm) and on the probe. Left: the released file with the `purebyte` CLI 1.0.0, default options (containers opened; findings must touch a printable string of at least 16 characters) and with `--all-bytes`. Right: the recipe's six seeds with the research code (the same weights with the *n*-gram tables at 32 bits; every span reported; containers scanned as they are), without the files that overlap the binary training corpus. The exams are partly not redistributable; the probe is synthetic, made with the training generator (see the text).

		Released file, CLI		Six seeds, research code		
Exam	Files	Default	`--all-bytes`	Mean	Worst	Released
bin-fp-a: extension modules of scipy 1.18.1 and scikit-learn 1.9.1 (Linux x86-64 wheels)	24 (38 MiB)	1	1	2.7	6	2
bin-fp-b (Linux part): libraries and executables (Ubuntu 26.04 packages, PyTorch and NumPy wheels)	25 (93 MiB)	9[a]	9	3.5[b]	8[b]	0[b]
bin-fp-b (Windows part): DLLs (system, DirectX, MFC, a graphics driver)	15 (80 MiB)	12	12	13.2	27	12
bin-fp-c: 14 formats absent from the training corpus of the first binary models	69 (109 MiB)	4[c]	7[c]	13.0[b]	32[b]	0[b]
Held-out extension modules: 60 modules of 18 Python packages, set apart before training	60 (34 MiB)	2	2	0.8	2	2
Probe: recall on the 360 decidable credentials	2,000 windows	1.000	1.000	0.995	0.989	1.000

[a] All in 4 of the 8 files that overlap the training corpus; 0 in the other 17. [b] Without the overlapping files: 17 Linux files and 67 files of bin-fp-c. [c] With `--no-archives`: 0 by default and 1 with `--all-bytes`.

Table 25: What a synthetic or development signal said, and what the real exam said, in the development of the three recipes.

Signal during development	The real exam
An earlier credential model: recall 0.98 on generated windows	recall 0.31 on CredData
A seed with a validation F1 of 0.867	catastrophic on real data
Released `secrets-code` seed: lowest validation F1 of three (0.883)	CredData F1 0.797; the three seeds 0.788–0.799
Bidirectional `pii` body (an earlier recipe): validation F1 0.913 against 0.84–0.85	tied on PIIMB at the "PIIMB mode" point (−0.005 in mean F2)
Released `pii` seed: the best *select* F1 of six	the lowest of the six on PIIMB (span 0.0164)
The binary seed with the best F1 on the planted-credential probe	the worst on real shared libraries
An ensemble chosen on one false-alarm set: zero false alarms there	148 on the next set

Table 26: I1: recall by distance with a role, eight streams of 1 MiB (trained on 16 KiB), seed 0. Balanced accuracy: minimum over the seven buckets unless noted; chance is 0.5.

Model	What it is	Balanced accuracy
No memory	the body and a trained head only	0.43 (mean)
Frozen body in stream	the body's state carried over the whole document	0.43 (mean)
Body retrained in stream	full backpropagation through 16 KiB	0.57–0.63 to 32 KiB, then chance
Sliding window	attention over the last 1 KiB; did not train in 3,000 steps (chance even inside its window)	0.47 (mean)
Learned key-value memory	learned keys, 4,096 entries (65,536: the same)	0.91 at 16 KiB, 0.66 (mean) at 1 MiB
h2	exact 8-byte address, gated writes, copy length; near-copy false alarms 0.53–0.66	0.86 (mean 0.88)
h6	*h2* + causal mixer + copy breaks	0.92 (mean 0.95)
h9	*h6* + exact presence + token coverage + hard negatives	**0.966** (mean 0.977)
h10	*h9* with bounded state (0.49 MB after 1 MiB)	**0.961** (mean 0.978)
h10, body in stream	the same memory, body state carried; twice as slow	0.958 (mean 0.974)

A Notation

Symbol	Meaning
$\mathcal{B}$, $\mathcal{B}^*$	byte values $\{0, \ldots, 255\}$; finite byte strings
$J = (\mathcal{Y}, P_J, \ell)$	a byte job: output space, data distribution, loss (Definition 1)
f_θ	a specialist with parameters θ; $\lvert\theta_{\text{net}}\rvert$ its network (byte embedding, blocks and final normalization), without lookup tables and heads
T, d	window length in bytes; model width (`d_model`)
h_t	hidden state of the body at byte t
D_s, M_s, c_s	build cost, maintenance cost and cost per decision of approach s (Equation (31))
m_s, f_s, e_s	misses and false alarms per decision, and their cost $e_s = m_s C_{\text{miss}} + f_s C_{\text{fa}}$
$T_s(N)$, N^*	total cost over N decisions; break-even number of decisions (Equation (32))

B Configurations of the released specialists

Table 27 lists what the three released model files declare (read from their metadata) and Table 28 the settings of the recipes that trained them. All three are causal (they read left to right), use the same heads and read windows of 512 bytes every 384.

Table 27: The released model files. Parameter counts: network = embedding, blocks and final normalization; tables = hashed n-gram tables at 4 bits and their projection; heads = the `tag` and `choice` heads (34,439 parameters). Quantization scales (one per group of 128 ternary weights, one per row of a 4-bit table) are derived and not counted. Of the table parameters of `secrets-bin`, 12,582,912 are never read (Section 4.10).

	`secrets-code` 1.0.0	`pii` 1.0.0	`secrets-bin` 1.0.0
Size preset	`nano`	`nano_n15`	`tiny_n18`
Blocks (`ssm_v2`)	4	4	8
Width d	256	256	256
SSM heads × head width	8×64	8×64	8×64
State size, groups	64, 2	64, 2	64, 2
Convolution width	4	4	4
Ternary group	128	128	128
n-gram orders	none	2, 3, 4	2, 3, 4
Rows per order × columns	–	$32{,}768 \times 64$	$262{,}144 \times 64$
Heads	`tag`: BIOES, one type, gated by the `choice` head; `choice`: 2 classes, max pooling, hidden width 128		
Window, stride	512, 384	512, 384 (minimum 1)	512, 384
Post-processing profile	`secrets-code`	`redact`	`secrets-binary`
Network parameters	1,927,520	1,927,520	3,789,248
Table parameters	0	6,340,608	50,380,800
Head parameters	34,439	34,439	34,439
Total	1,961,959	8,302,567	54,204,487
File size (bytes)	1,028,704	4,764,864	30,163,488

Table 28: Training settings of the released recipes. Each step draws `batch` generated windows and appends a share (`real_frac`) of real windows: labeled windows of CredData's development half for `secrets-code` (spans), real negative windows of public binaries for `secrets-bin` (neg). All use AdamW ($\beta_1 = 0.9$, $\beta_2 = 0.95$), weight decay 0.05 on matrices only, a warm-up of 30 steps, a plateau and a linear decay to zero over the last 20 % of the steps, gradient-norm clipping at 1.0, and the loss $\mathrm{CE}_{\mathrm{tag}} + 0.5\,\mathrm{CE}_{\mathrm{choice}}$.

	`secrets-code`	`pii`	`secrets-bin`
Steps	12,000	12,000	6,000
Generated windows per step	16	16	16
Real windows per step	4 (`real_frac` 0.25)	0	7 (`real_frac` 0.45)
Peak learning rate	3×10^{-4}	3×10^{-4}	1×10^{-3}
Output-projection learning-rate multiplier	1	1	0.25
Seeds	3	6	6

C Parameters of the cost model

Figure 13 evaluates Equation (31) with $e_s = M_s = 0$, in hours of waiting time over N decisions, for three approaches; Figure 14 adds the error terms against rules. Parameters we measured are marked (m), figures published by others and quoted (q), and assumed ones (a), which are ranges chosen for illustration.

Table 29: Parameters of Figures 13 and 14.

Approach	Build cost D_s	Cost per decision c_s and errors e_s
(a) rules written by an agent	(a) 1 to 8 hours of an agent's and a person's time	(m) a pattern scanner reads bytes roughly 80 times faster than the specialist: 5.9 µs per 64-byte input, about 0.04 ms per window of 384 bytes (busy machine, approximate); (m) on CredData, 3.5×10^{-3} missed credentials and 9.7×10^{-5} false alarms per window
(b) a general model at run time	(a) zero (a prompt), plus an examination to know its quality, not counted	(q) 236–276 ms for a hosted decision API (third-party median, quoted by Laya's model card); (a) 2 s, a separate scenario for a generative model that reasons before answering; errors not measured
(c) an AI Specialist trained by an agent	(a) 4 to 24 hours of an agent's and a person's time, plus (m) 25 minutes to 1 hour of one consumer GPU	(m) 0.82 ms per short decision on eight threads of a desktop CPU, 2.9 ms per window of 384 bytes on the whole desktop; (m) on CredData, 1.2×10^{-3} missed credentials and 4.4×10^{-4} false alarms per window

The error rates are those of the published scope of CredData's test half (4,219 labeled credential lines in about 9.5×10^5 windows) as the benchmark distributes it; on its original values the specialist misses more, which moves the threshold of Figure 14 from about ten to about twelve seconds (Section 7.2). Their false alarms are those on labeled lines, as CredData's precision counts them, for both tools. A false alarm is assumed to cost 60 s of a person's triage, and the cost of a miss is the variable of Figure 14. The specialist's build cost assumes that the training stack, the data sources and an examination already exist: it is at the low end of what this project observed while building them (Section 7.5). Money and energy follow Equation (31) with other units: a provider's price per call for c_{llm}, a few milliseconds of CPU cores for c_{spec}. Prices of language models are falling fast [111], which lowers c_{llm} and raises N^* against run-time reasoning; the latency, the privacy and the determinism of Section 7.4 do not change with price.

D Data and licenses

Table 30 lists every dataset behind the released specialists and their examinations: its pin, its license, what it was used for and whether it can be redistributed. Datasets are never stored in the repositories: the data scripts download them from the pinned sources and check their SHA-256.

Table 30: Data behind the released specialists and their examinations.

Data	Pin	License	Use
OpenPII-1M, TRAIN split [209]	revision `ecfdc547`	CC BY 4.0	training of `pii`
Nemotron-PII, TRAIN split [210]	revision `b70ffaf5`	CC BY 4.0	training of `pii`
Gretel PII masking EN v1, TRAIN [211]	revision `e06eb149`	Apache-2.0	training of `pii`
Gretel synthetic PII finance, TRAIN [212]	revision `7b844d16`	Apache-2.0	training of `pii`
INE name and surname tables [227]; Tatoeba sentences [228]	SHA-256	CC BY 4.0; CC BY 2.0 FR	generators of `pii`
gitleaks v8.30.1 and trufflehog v3.97.6 catalogs [145, 146]	commits of the tags	MIT; AGPL-3.0-only	formats and example strings for the credential generators (never their code); built locally, never committed
Public code repositories	cloned on 19 September 2026 for the release; the scripts pin the heads of 23 September	permissive licenses	context windows of `secrets-code` and code windows of `pii`; not redistributed
Binaries	the training machine's system folders for the release; the scripts pin public packages by SHA-256	per package	context and negative windows of `secrets-bin`; the release corpus cannot be redistributed
CredData [150]	commit `c09c0c5`	Apache-2.0; each file under its project's license	examination (test half) and training (labeled lines of the development half, with CredData's obfuscated values)
PII Masking Benchmark [161]	revision `7d797b9f`	CC BY-NC 4.0	examination, and during development a comparison of variants (Table 19); never training material; scores published, texts never
Clean code repositories (code-fp)	commits	their licenses	false-alarm examinations of `secrets-code`
Clean binaries and the probe	packages by SHA-256, partly internal	per package	examinations of `secrets-bin`; partly not redistributable

The attributions that the licenses require: "Ai4Privacy / Ai Suisse SA OpenPII-1M, CC BY 4.0"; "NVIDIA Nemotron-PII, CC BY 4.0"; "Gretel.ai datasets, Apache 2.0"; "Contains data from the Instituto Nacional de Estadística (INE), www.ine.es, CC BY 4.0"; "Sentences from Tatoeba (tatoeba.org), CC BY 2.0 FR". The model cards of the released specialists carry the same lines. The code of the runtime and of the training stack is Apache-2.0; the weights of the three released specialists are under the PureByte Model License 1.0, which allows their use, commercial use included, but not their redistribution, models derived from them or a hosted service whose main value is the model itself.

E Reproducing the numbers

The runtime and its tests, the evaluations of `secrets-code` and `pii` on public data, and the training recipes of `secrets-code` and `pii` can be rerun from the two public repositories and public data (that of `secrets-bin` cannot: no script builds its negative windows). The released weights cannot be rebuilt bit for bit (they were trained on a GPU, and a retraining is a new draw from the recipe's seeds); the negative binaries and the examinations of `secrets-bin` are not public; and the `data` stage refuses the release data of `secrets-code` and `pii` unless they are rebuilt without their overlap with the examinations (Section 6). The design ablations of Section 4.11, the six seeds of `secrets-bin`, the "PIIMB mode" figures and the document memory of Section 9 were computed with research code that is not released, and are reported as such. Only the released file of each model is published (with the two extra members of the `secrets-code` ensemble), so the six-seed figures of `pii` and `secrets-bin`, and the criteria read on them, cannot be rerun on the released seeds. The runtime and the evaluations live in purebyte-ai/purebyte, the training stack in purebyte-ai/purebyte-train; their documentation gives every option.

The runtime and its exactness tests.

```
git clone https://github.com/purebyte-ai/purebyte && cd purebyte
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build --config Release --parallel
ctest --test-dir build --build-config Release --output-on-failure
```

The suite compares the engine with the NumPy reference on random models generated from the specification, checks bit-identity across kernels and thread counts, and refuses malformed model files (Section 5).

The released models. `purebyte models pull NAME` downloads a released model (`pii`, `secrets-code`, `secrets-code-ensemble` or `secrets-bin`) and verifies its SHA-256 against the catalog compiled into the runtime; `purebyte bench` measures a machine's latency and throughput.

CredData (Section 8.2). The snapshot we measured, prepared by CredData's own script:

```
git clone https://github.com/Samsung/CredData.git && cd CredData
git checkout c09c0c52fc6dae4ae5438ae69ba486f9f8059f0d
python3 -m pip install -r requirements.txt
python3 download_data.py --data_dir data --jobs 8
python3 /path/to/purebyte/benchmarks/creddata/score.py split --creddata .
bash /path/to/purebyte/benchmarks/creddata/run.sh /path/to/CredData test
MODEL=secrets-code-ensemble bash /path/to/purebyte/benchmarks/creddata/run.sh \
    /path/to/CredData test
```

The script prints both published tables: without the four repositories that overlap the training corpus (164 repositories, the headline figures; `--exclude-repo training-overlap` in `score.py`) and for the complete test half of 168 repositories. It runs gitleaks and trufflehog when they are installed; we used gitleaks 8.30.1 with its default rules and trufflehog 3.97.6 without verification. The paired interval of the overlap's effect uses `score.py score --paired --drop-lines` with a list of lines that is not published. The figures on the original values (Table 22) come from a script that fetches third-party credential values and is not published.

Clean repositories (Section 8.2). In a purebyte-train checkout, `python data/fetch_exams.py --set code-fp-a --set code-fp-b --set code-fp-c` fetches the three sets at their pinned commits, and `specialists/secrets-code/eval/count_findings.py` counts what the CLI reports on each (`--models DIR --exam PATH --out FILE`).

PII Masking Benchmark (Section 8.3).

```
mkdir piimb && cd piimb
curl -LO "https://huggingface.co/datasets/piimb/pii-masking-benchmark/resolve/\
7d797b9fc8dc1942cef60fbe532e7d1a0e31b655/data/test_sentences.jsonl"
bash /path/to/purebyte/benchmarks/piimb/run.sh /path/to/piimb pii
```

The benchmark is licensed CC BY-NC 4.0: it is used to measure, never to train. The other models' figures are those of the benchmark's public leaderboard. With `EXCLUDE=PIIMB_OVERLAP.json` before `run.sh`, the score leaves out the sentences that share a 64-byte block with the training pool; `python data/build_pii_pool.py --piimb-overlap` writes that list in a purebyte-train checkout, and it is kept local because it is derived from the benchmark.

Training a released recipe (Section 6).

```
git clone https://github.com/purebyte-ai/purebyte-train && cd purebyte-train
# first install PyTorch for your hardware (pytorch.org/get-started/locally)
pip install -e "training[test]"
python -m pbtrain doctor
python -m pytest -q training/tests    # the training contract, the factory on a CPU
# data scripts (data/README.md): examinations first, then corpora
python data/build_creddata_stream.py --exam-overlap drop
python -m pbtrain run specialists/secrets-code/recipes/secrets-code-1.0.0.yaml \
    --device auto --parallel 3
```

The data scripts of the training repository must run first, examinations before corpora; the repository's documentation lists them in order. The result is a new draw of the recipe, not the released weights: expect results within the spread of the recipe's seeds, not the released figures (Section 6). No retraining with the public code has yet been scored on the examinations.

The document memory (Section 9). The memory is research code that is not part of the release; the measurements of Section 9 were taken with it on the same machine and are reported as research results.

F Glossary

AI Specialist
A small model trained for one job, which reads bytes and answers with decisions, scores, labels or byte ranges (Definition 2).

Vibe Training
Training an AI Specialist by describing the job to an AI coding agent, through a factory that blocks leaks, silent options and unverified exports and records failed criteria; some of its rules are written rather than enforced (Definition 10).

Body
The embedding, the n-gram tables and the stack of blocks that turn bytes into hidden states.

Head
A layer that turns hidden states into one kind of answer (Section 3.2).

Profile
Post-processing of the runtime that merges, filters, masks or redacts the spans of a model.

Window
The unit the body reads at once (512 bytes every 384 in the examples); a stream context reads inputs of any length.

Examination, exam
A held-out real evaluation set, fingerprinted so that the `data` stage refuses training windows that share a 64-byte block with an examination of role `test`; the released examples predate part of this check (Section 6).

Control
The same recipe trained on labels shuffled between windows; it must score about zero.

Rival
A classic rule-based baseline for the same job, tuned on held-out data; the gate requires the mean of the seeds to beat it by more than the spread between seeds, a sanity check rather than a significance test.

Operating point
The label bias chosen on held-out data that trades precision for recall.