AAEN
Affan AI Engine Nano

TECHNICAL REPORT | VERSION 3 | PART I | PAGES 1 TO 6

Affan AI Engine Nano V3

Design, Mathematics and Training

How I designed, trained and specified a 14-million-parameter chat model with a built-in calculator tool.

13,965,696	~8 MB	~4B	87.9%
parameters	int4 model file	tokens trained in total	known-fact accuracy

Muhammad Affan bin Adnan - MABA Studios

Model, native engine, in-browser engine and website: one project, documented in two parts.

Abstract and System Overview

Abstract

In this report I describe Affan AI Engine Nano (AEN), a complete small language-model system that I designed and built. It consists of a 13,965,696-parameter decoder-only transformer that I trained from scratch, a calculator tool that the inference engine executes on the model's behalf, a native C++ engine, a WebAssembly engine that runs in the browser, and a static website that serves them. My central design constraint is that the model must stay at 14 million parameters and about 8 MB, so that the same file runs on an ESP32-S3 microcontroller and in a web browser.

Version 3 is the product of a 13.9-hour training pipeline that I ran on a single laptop GPU. On a 420-question hard probe written in wordings that the model never saw during training, known-fact accuracy rose from 28.1% to 87.9%, while the model continued to decline 67.8% of unanswerable questions. This report documents the architecture, the mathematics, the training method, the quantisation scheme, the engines, the evaluation and, importantly, the limits of the system.

My contributions

- A compact transformer specification (8 layers, width 384, grouped-query attention with 6 query and 2 key-value heads) whose exact parameter count I derive in Section 3.
- A tool-use protocol in which the model writes an arithmetic expression and the engine, not the model, computes the result.
- A fine-tuning recipe that adds casual chat and about 1,500 verified facts without forgetting earlier abilities (KL-regularised replay and an assistant-only loss).
- A 4-bit compression pipeline (GPTQ followed by distillation-based quantisation-aware training) that keeps about 99% of full-precision quality.
- The AFN single-file format and two engines (native C++ and WebAssembly SIMD) that are numerically checked against each other.
- An evaluation protocol built on held-out wordings and a quality gate that decides whether a new model may replace the old one.

System architecture



Every stage is reproducible from files in the project folder; the quality gate in Part II decides whether a new model is installed.

Contents of this report

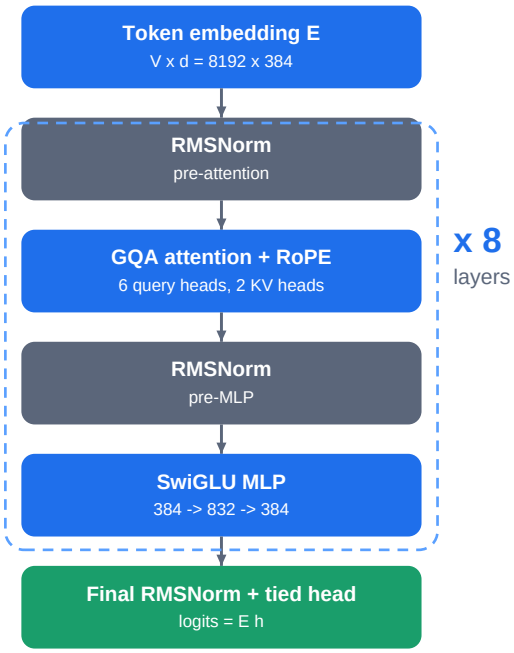
Pages	Part	Topics
1-6	Part I	Design, mathematics of the network, tool protocol, training
7-12	Part II	Quantisation, engines, speed model, evaluation, outlook

Notation used throughout

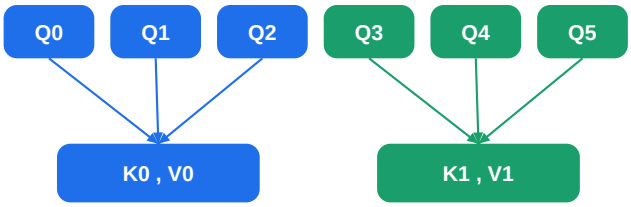
Symbol	Meaning	Value
d	width of the residual stream	384
L	number of transformer layers	8
n_h, n_kv	query heads, key-value heads	6, 2
d_h	size of one attention head	64
f	hidden width of the SwiGLU MLP	832
V	vocabulary size	8,192
N	total number of parameters	13,965,696
s_g, b	scale of a quantisation group (32 weights); bits per weight	fp16, 4.5

Model Architecture

I chose a decoder-only transformer in the pre-norm style, with SwiGLU feed-forward blocks, rotary position embeddings and RMSNorm. Grouped-query attention lets six query heads share two key-value heads, which cuts the key-value cache to one third. The input embedding and the output projection share one matrix, which saves 3.15 million parameters.



Grouped-query attention (n_h = 6, n_kv = 2)



Each group of 3 query heads reads the same key-value head, so the cache stores 2 head pairs instead of 6.

Hyper-parameter	Value
Layers L	8
Width d	384
MLP width f	832
Heads n_h / n_kv	6 / 2
Head size d_h	64
Vocabulary V	8,192
Context	1,024
RoPE base	10,000

Exact parameter count

$$N = Vd + L(2d^2 + 2\,d\,d_{kv} + 3\,d\,f + 2d) + d, \qquad d_{kv} = n_{kv}\,d_h = 128 \tag{1}$$

$$N = 3{,}145{,}728 + 8 \times 1{,}352{,}448 + 384 = 13{,}965{,}696 \tag{2}$$

Component	Parameters	Share
Tied embedding / output head	3,145,728	22.5%
Attention (8 x 393,216)	3,145,728	22.5%
SwiGLU MLP (8 x 958,464)	7,667,712	54.9%
RMSNorm weights	6,528	0.05%
Total	13,965,696	100%

Where the parameters live, and why it matters



- Tied embeddings save V x d = 3,145,728 parameters, 22.5% of the whole model.
- With GQA the key-value cache at 1,024 tokens is 2 x 8 x 2 x 64 x 1024 = 2,097,152 values (2 MiB at one byte each); with six key-value heads it would be 6 MiB.
- Every matrix row length (384, 832 and 128) is a multiple of 32, so each row splits into whole quantisation groups (12, 26 and 4 groups); d_h = 64 also fills 128-bit SIMD lanes exactly.

Mathematics of the Network

I write x for the residual-stream vector of one token (dimension d = 384). The forward pass of one layer consists of the following operations; equations (3) to (10) are exactly what both engines implement.

$$\mathrm{RMSNorm}(x) = \frac{x}{\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \epsilon}} \odot g, \qquad \epsilon = 10^{-5} \tag{3}$$

$$q_h = W_Q^{(h)} \hat{x}, \quad k_j = W_K^{(j)} \hat{x}, \quad v_j = W_V^{(j)} \hat{x}, \qquad j(h) = \lfloor h/3 \rfloor, \; h = 0, \ldots, 5, \; j = 0, 1 \tag{4}$$

Rotary embedding: every consecutive pair of components of q and k is rotated by an angle that grows with the token position m.

$$x'_{2i} = x_{2i}\cos(m\theta_i) - x_{2i+1}\sin(m\theta_i), \quad x'_{2i+1} = x_{2i}\sin(m\theta_i) + x_{2i+1}\cos(m\theta_i), \quad \theta_i = 10000^{-2i/d_h} \tag{5}$$

$$a_{h,s}(t) = \frac{\exp\left(q_h(t) \cdot k_{j(h)}(s) / \sqrt{d_h}\right)}{\sum_{u \le t} \exp\left(q_h(t) \cdot k_{j(h)}(u) / \sqrt{d_h}\right)}, \qquad o_h(t) = \sum_{s \le t} a_{h,s}(t)\, v_{j(h)}(s) \tag{6}$$

$$y = x + W_O \left[o_0; \, o_1; \, \ldots; \, o_5 \right] \tag{7}$$

$$\mathrm{FFN}(x) = W_{\mathrm{down}}(\mathrm{SiLU}(W_{\mathrm{gate}}\, x) \odot (W_{\mathrm{up}}\, x)), \qquad \mathrm{SiLU}(z) = \frac{z}{1+e^{-z}} \tag{8}$$

$$x^{(l+1)} = y + \mathrm{FFN}(\mathrm{RMSNorm}(y)), \qquad z = E\,\mathrm{RMSNorm}(x^{(L)}) \in \mathbb{R}^V \tag{9}$$

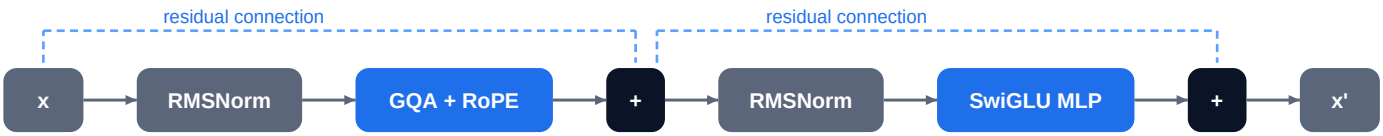
The causal mask in (6) (the sum runs over s <= t) is what makes the key-value cache exact: the keys and values of earlier tokens never depend on later tokens. Equation (9) ends with the tied output head, so the same matrix E embeds the input token and scores the next one.

Cost per generated token

$$\mathrm{MACs}(t) \approx 13.97 \times 10^6 + 2\, n_h\, L\, d_h\, t = 13.97 \times 10^6 + 6144\, t, \qquad \mathrm{MACs}(1024) \approx 20.3 \times 10^6 \tag{10}$$

At a short context the weight matrices dominate: about 14 million multiply-accumulates (MACs) per token, i.e. roughly 28 MFLOPs. At the full 1,024-token context the attention term adds about 45%. I use equation (10) in Part II to bound the speed on a microcontroller.

One layer as a data-flow diagram



Tensor shapes (input x output)

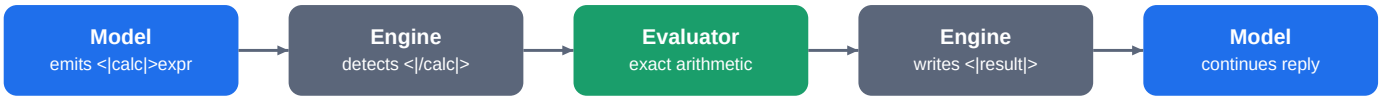
Tensor	Shape	Parameters	How many
W_Q, W_O	384 x 384	147,456 each	1 each per layer
W_K, W_V	384 x 128	49,152 each	1 each per layer
W_gate, W_up	384 x 832	319,488 each	1 each per layer
W_down	832 x 384	319,488	1 per layer
RMSNorm weights g	384	384 each	2 per layer + 1 final
E (tied)	8192 x 384	3,145,728	1 in total

Tokeniser, Chat Format and Calculator Tool

A byte-level BPE tokeniser with V = 8,192 tokens is stored inside the model file, so a single file is sufficient for any engine. Digits are split into single tokens, which makes arithmetic easier to learn and lets the calculator receive exact operands. The chat format uses special role tokens:

```
<|bos|><|system|>You are Nano, a friendly small AI chat companion made by MABA Studios.<|end|>
<|user|>What is 17 times 23?<|end|><|assistant|>
<|calc|>17*23<|/calc|><|result|>391<|/result|>It is 391.<|end|>
```

Calculator tool protocol



The model therefore only has to learn to translate a question into an expression; the value is computed by the engine and inserted as ordinary tokens. In my 40-question probe the v3.0 model produced 39 correct expressions (v2.2: 38). The user never sees the tags: the engine hides them from the displayed answer.

Sampling

$$p_i = \frac{\exp(z_i/T)}{\sum_k \exp(z_k/T)}, \qquad \text{then keep the top-}k\text{ tokens and the smallest set with cumulative mass } \geq p_{\text{top}} \tag{11}$$

Greedy decoding (T = 0) is used for tests and for the parity checks between the two engines.

Stateless chat and the exact prefix cache

Early versions kept the whole conversation in the key-value cache. I observed that identical prompts were answered well in a new chat and worse in a long one, so I removed conversation history completely: every message is answered as a new chat. The only state kept is the cache of the fixed system prompt, which is exact because of the causal property noted after equation (9).

$$\text{reply} = f(\text{system, user}) \quad \text{independent of earlier turns;} \qquad S_{KV} = 2\,L\,n_{kv}\,d_h\,t\,b = 2{,}097{,}152\,b \text{ bytes at } t = 1024 \tag{12}$$

S_KV is the size of the key-value cache for t = 1,024 tokens at b bytes per stored value: 2 MiB for b = 1, 4 MiB for b = 2, 8 MiB for b = 4.

Special tokens

Token	Role
<|bos|>	start of every sequence
<|system|> <|user|> <|assistant|>	role markers that open each part of the chat
<|end|>	closes a turn; the engine stops generating when the assistant emits it
<|calc|> ... <|/calc|>	delimits an arithmetic expression written by the model
<|result|> ... <|/result|>	delimits the value inserted by the engine (never produced by the model)

Digit splitting means that 391 is read as the three tokens '3', '9', '1', so the model learns column arithmetic patterns and the calculator always receives exact operands.

Training Objective and Pipeline

I trained in stages. The base model saw about 2 billion tokens of simple English text and stories. I then fine-tuned for casual chat (v2.1) and for a closed set of verified facts with honest abstention (v2.2). Version 3.0 is a further 13.9-hour pipeline. Earlier attempts in which I simply added data made the model worse, so every stage of the recipe is designed to prevent forgetting.

$$\mathcal{L} = \frac{1}{|\mathcal{A}|}\sum_{t \in \mathcal{A}} -\log p_\theta(x_t \mid x_{<t}) + \lambda \mathbb{E}_{x \sim \mathcal{R}} \mathrm{KL}(p_{\text{base}} \,\|\, p_\theta), \qquad \mathrm{KL}(p \,\|\, q) = \sum_v p_v \log \frac{p_v}{q_v} \tag{13}$$

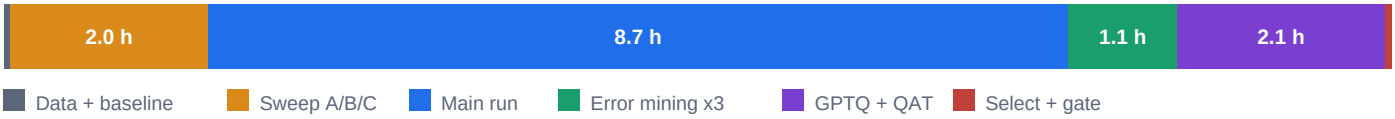
Here A is the set of assistant-reply positions (the user's and the system's tokens are masked out), R is a replay set of stories and general text, and p_base is the frozen pre-fine-tuning model. A sweep of three recipes selected lambda = 0.4 with a learning rate of 8e-5 (composite scores 126.8, 124.8 and 122.3).

Token accounting

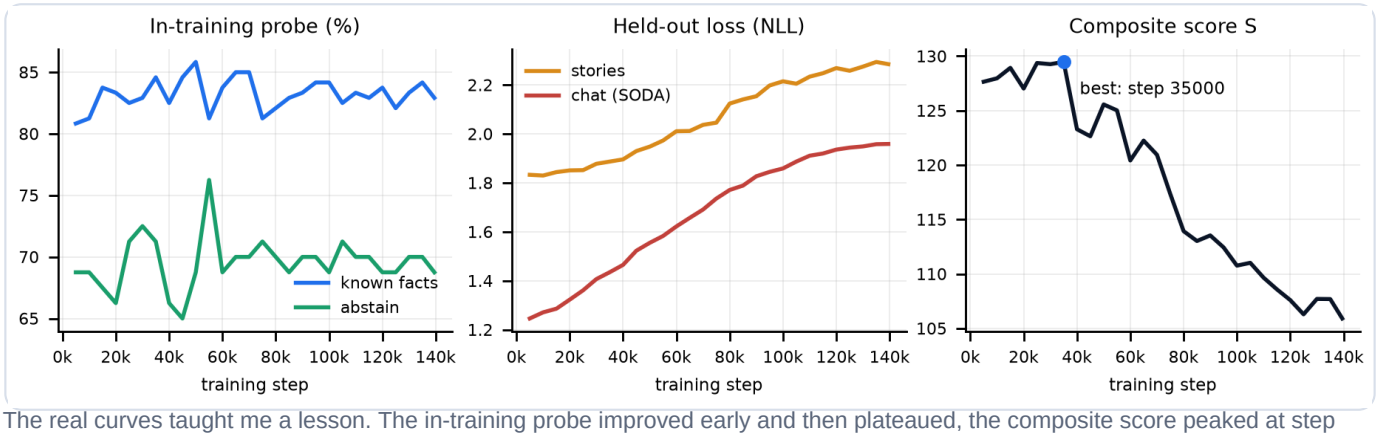
$$T = S(B + B_r)\ell = 139{,}516 \times (32 + 12) \times 256 \approx 1.57 \times 10^9 \text{ tokens (main run)} \tag{14}$$

Adding the 2.0 B base tokens, about 0.1 B of earlier tuning and 3 x 10,000 sweep steps (about 0.34 B) gives roughly 4 billion tokens in total. This counts repeated chat and fact examples; I do not claim 4 billion distinct tokens.

Pipeline timeline (13.9 hours, one 8 GB laptop GPU)



Behaviour during the main run (measured)



The real curves taught me a lesson. The in-training probe improved early and then plateaued, the composite score peaked at step 35,000, and the held-out losses rose steadily afterwards (stories from about 1.83 to 2.28, chat from about 1.25 to 1.95): a clear sign of forgetting and over-fitting. The full 9-hour main run was longer than it needed to be. Because the pipeline stores the best checkpoint by composite score, the later checkpoints could not replace it; I then applied three rounds of error mining on the questions the model still failed.

Fine-tuning data mix (v2.1 / v2.2 metadata)



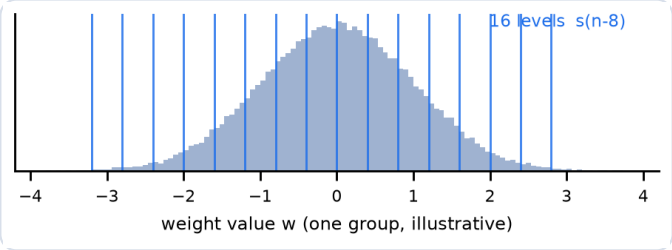
Quantisation and the AFN File Format

An fp32 copy of 13.97 million parameters occupies about 56 MB. To reach about 8 MB, I store every weight, including the embedding, as a 4-bit integer in groups of 32 with one fp16 scale per group.

$$\hat{w} = s_g\,(n-8), \qquad n = \mathrm{clip}\big(\mathrm{round}\big(\tfrac{w}{s_g}\big)+8,\ 0,\ 15\big), \qquad b = 4+\tfrac{16}{32} = 4.5 \text{ bits per weight} \tag{15}$$

$$\text{bytes} = \frac{Nb}{8} = \frac{13{,}965{,}696 \times 4.5}{8} = 7{,}855{,}704 \ \ (\approx 7.86 \text{ MB}); \qquad \text{file} = 8{,}438{,}532 \text{ bytes} \tag{16}$$

The remaining 0.58 MB of the file holds the tokeniser, the JSON header, the RMSNorm weights and the 4096-byte alignment padding.



A group of 32 weights is mapped onto the 16 levels s(n - 8). The scale s_g is chosen per group; naive rounding loses too much on a model this small: in my tests it kept only 84 to 88% of the full-precision quality. I therefore use two steps.

Step 1 - GPTQ (calibrated rounding)

$$\min_{\hat{W}} \|WX - \hat{W}X\|_F^2; \qquad \delta w = -\frac{w_q - \hat{w}_q}{[H^{-1}]_{qq}}\,[H^{-1}]_{:,q}, \qquad H = 2XX^\top \tag{17}$$

X holds calibration activations. After each weight q is rounded, the not-yet-rounded weights are corrected by delta w so that the layer output changes as little as possible.

Step 2 - Quantisation-aware training with distillation

$$\mathcal{L}_{QAT} = \mathrm{KL}(p_T \,\|\, p_S), \qquad \hat{w} = s \cdot \mathrm{clip}(\mathrm{round}(w/s)), \qquad \frac{\partial \hat{w}}{\partial w} \approx 1 \ \ (\text{straight-through estimator}) \tag{18}$$

The student S (quantised) is trained to reproduce the output distribution of the teacher T (the full-precision v3.0 model). Gradients pass through the rounding with the straight-through estimator. I ran 26,000 steps, kept every evaluated checkpoint and chose among real int4 files by testing them in the engine. Result: about 99% of full-precision quality on stories and about 100% on identity.

AFN container (AFNESP01)



data aligned to 4096 B

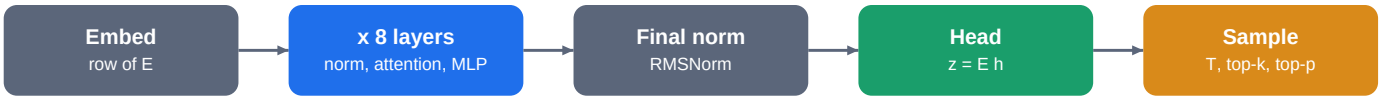
two 4-bit values per byte, low nibble first, stored as value + 8

File	Size	Use
model_int4.afn	8.44 MB	ESP32-S3, browser, native engine (default)
model_int8.afn	14.98 MB	higher-fidelity reference
model-q8_0.gguf	15.14 MB	GGUF tooling, 8-bit
model-f16.gguf	28.23 MB	GGUF half precision

Format (bits per weight)	Formula	Weights only	File on disk
fp32 (32)	N x 32 / 8	55.9 MB	not shipped
fp16 (16)	N x 16 / 8	27.9 MB	28.23 MB (GGUF)
int8 + scale (8.5)	N x 8.5 / 8	14.8 MB	14.98 MB (AFN)
int4 + scale (4.5)	N x 4.5 / 8	7.86 MB	8.44 MB (AFN)

Inference Engines

I implemented the same algorithm twice: AEN.cpp for PCs and the ESP32-S3, and a JavaScript port with hand-assembled WebAssembly SIMD128 kernels for the browser. A decode step is a fixed sequence of operations:



The int8-by-int4 kernel

Activations are quantised on the fly to int8 with a per-vector scale, so the inner loop is an integer dot product between int8 activations and 4-bit weights; one floating-point multiplication per group of 32 restores the scale.

$$y_r \approx \Delta_a \sum_g s_{r,g} \sum_{i \in g} a_i^q \, (n_{r,i} - 8), \qquad a_i^q = \mathrm{round}\left(\frac{a_i}{\Delta_a}\right), \quad \Delta_a = \frac{\max_i |a_i|}{127} \tag{19}$$

Threads in the browser

Rows of a weight matrix are split into small dynamic chunks that idle workers claim:



Threads need cross-origin isolation (COOP and COEP headers), which the site provides through its _headers file or a service worker; without it, the engine still runs single-threaded, and without SIMD it falls back to scalar JavaScript.

Measured results

Check (all measured on my PC)	Result
Generation speed, WebAssembly SIMD128, 4 threads	368 tokens/s (485 tokens/s including prompt)
Generation speed, scalar JavaScript fallback	about 32 tokens/s
Top-1 agreement with native AEN.cpp on story text	99.6%
Test-set NLL: web engine vs native engine	1.7263 vs 1.7273
4-thread logits vs 1-thread logits	identical (max difference 0)
Engine test suite / site SEO checks	36 of 36 / 103 of 103 pass

A scalar-versus-SIMD comparison differs by up to 0.19 in one logit because the two int8 paths round differently; the top-1 token is identical, so I allow a tolerance of 0.5 in that single test.

What the measured speed implies

$$\frac{\mathrm{MACs}}{\mathrm{s}} \approx 13.97 \times 10^6 \times 368 \approx 5.1 \times 10^9 \ \ \mathrm{MAC/s} \quad (\mathrm{PC},\ 4\ \mathrm{threads},\ \mathrm{SIMD}) \tag{20}$$

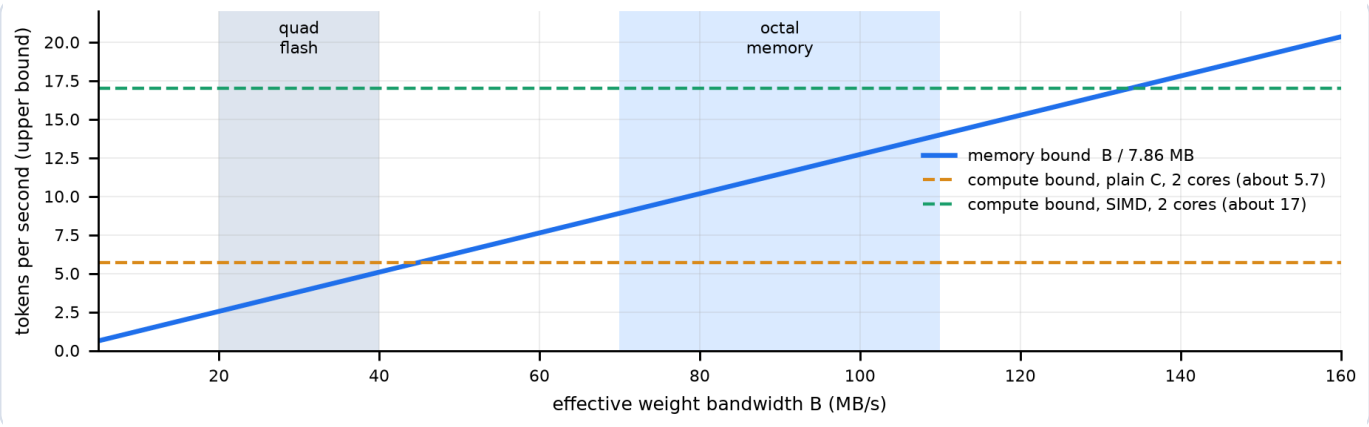
My PC therefore sustains about 5 billion MAC/s, roughly 60 times the 80 million MAC/s that I assume for a plain-C two-core ESP32-S3 on the next page. This is why I treat the microcontroller as a separate engineering problem.

Engine	Language and kernels	Threads	Status
Native CPU	C++ (AEN.cpp)	yes	tested on PC
Browser	JS + WASM SIMD128	up to 4	tested, 36/36 checks
ESP32-S3	C, to be optimised	2 cores	not yet measured
GPU (Vulkan)	-	-	not built

Speed Model for the ESP32-S3

I have measured the engine on my PC, not yet on an ESP32-S3. This page is therefore an analytical estimate. Generating one token requires reading essentially every weight once, so speed is limited either by memory bandwidth or by compute, whichever is smaller:

$$v \le \min\left(\frac{B_{\text{eff}}}{S_w}, \frac{P_{\text{eff}}}{M}\right), \qquad S_w \approx 7.86 \text{ MB}, \quad M \approx 14 \times 10^6 \text{ MAC per token} \tag{21}$$



Quad flash delivers roughly 20 to 40 MB/s, so the memory bound is 2.5 to 5 tokens/s. Octal memory raises the bound, but the compute bound of plain C (assumed 40 million effective MAC/s per core, two cores) is about 5.7 tokens/s; only SIMD kernels lift that ceiling. These assumptions are mine and unmeasured.

Configuration	Estimated speed	Limiting factor
Quad flash, plain C	2 to 4 tokens/s	memory bandwidth
Octal flash or PSRAM, plain C	4 to 6 tokens/s	compute
Octal memory + SIMD kernels	6 to 10 tokens/s	memory and overheads

Memory map of the target board

Flash 16 MB
memory-mapped, read-only
firmware + AFN file (8.44 MB)

PSRAM 8 MB
external RAM
KV cache (2 to 4 MiB), activations

SRAM 512 KB
internal, fastest
stack, norms, scales, hot loops

The 8.44 MB file does not fit an 8 MB PSRAM, so the weights stay in flash and PSRAM holds the cache. Code can approach the bandwidth ceiling (SIMD kernels, both cores, 80 MHz quad-I/O flash mode, large cache, hot loops in internal RAM, multi-token prompt processing) but cannot exceed it; only fewer bytes per token or faster memory can.

Waiting time for one answer

$$t_{\text{reply}} = \frac{n_{\text{tokens}}}{v} + t_{\text{prompt}}, \qquad t_{\text{prompt}} \approx \frac{n_{\text{user}}}{v} \quad \text{(system prompt is cached)} \tag{22}$$

Reply of 30 tokens, 15-token question	v = 3	v = 5	v = 8 tokens/s
Time to finish the answer (prompt included)	15 s	9 s	5.6 s
Time before the first word	5 s	3 s	1.9 s

Values are computed from equation (22) for the three speeds, not measured on hardware.

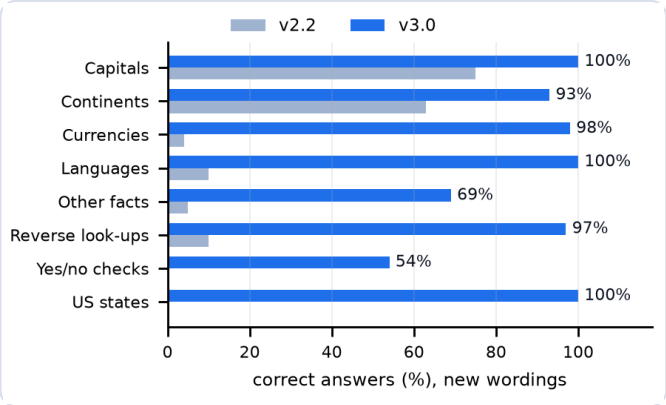
Evaluation and Results

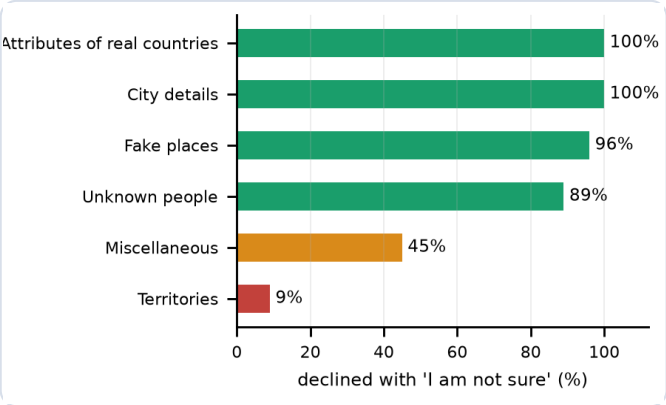
A 14M model can memorise its training questions, so I never judge it on them. My probe set contains 420 known-answer questions and 90 unanswerable ones (made-up places and people, other details of real countries, live data), all written in wordings and wrappers that are disjoint from the training templates. Candidate models are ranked by a composite score and must pass a gate against the previous version:

$$S = A_{\text{known}} + 0.7\,A_{\text{abstain}} - P_{\text{ident}} - P_{\text{stories}} - P_{\text{chat}} \tag{23}$$

Gate: known facts at least +25 points, abstention not worse than -3, casual-chat leak not worse than +3, calculator at least 35 of 40.

Measure (420 + 90 hard probe, new wordings)	v2.2	v3.0 int4
Known facts answered correctly	28.1%	87.9%
Unanswerable questions declined ('not sure')	65.6%	67.8%
Casual prompts with a facts-style reply (of 90, lower is better)	1	1
Calculator: correct expression (of 40)	38	39





Reading the results

- Capitals, languages, reverse look-ups and US-state questions are answered almost perfectly; the largest gains over v2.2 are in currencies (4% to 98%) and languages (10% to 100%).
- Yes/no checks (54%) and miscellaneous facts (69%) remain weak. Territories are almost never declined correctly (9%): the model tends to answer instead of saying it is not sure.
- Fake places (96%) and unknown people (89%) are declined reliably, which is the behaviour I wanted from the abstention training.
- Retention of the int4 file against full precision is about 99% on stories and about 100% on identity, measured on the evaluation sets used during quantisation.

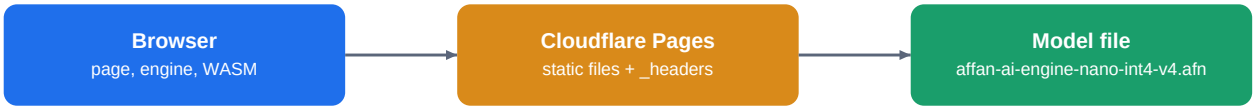
Quality gate: v3.0 against v2.2

Check	Requirement	v3.0 result	Verdict
Known-fact accuracy	at least +25 points	+59.8 points	pass
Abstention rate	not worse than -3	+2.2 points	pass
Casual-chat leak (of 90)	not worse than +3	+0	pass
Calculator correct (of 40)	at least 35	39	pass

Sweep recipe (10,000 steps)	lr	KL weight	Score S
A	8e-5	1.0	122.3
B	1.5e-4	1.0	124.8
C (selected)	8e-5	0.4	126.8

Deployment, Limits and Outlook

The website is entirely static, so it runs on Cloudflare Pages without any server: the visitor downloads the 8.44 MB model once, and every reply is computed on the visitor's own device.



No server computes anything; the model file is cached by the browser, so later visits and offline use need no download.

Honest limits

- Facts: the model only knows what I taught it. Outside its fact set it can still invent answers, especially about real places it was not taught.
- Context: 1,024 tokens, and every message is a fresh chat by design, because longer chats degrade at this size.
- Language: English only. I considered Urdu and Roman Urdu and dropped them to protect quality.
- Platforms: not yet tested on a real phone or an ESP32-S3 board, not yet deployed; a Vulkan backend is not built, and Metal and ARM paths are untested.

Lower-bit options I studied and deferred

$$\text{size} \approx \frac{N_{\text{body}}\, b_{\text{body}} + N_{\text{emb}}\, b_{\text{emb}}}{8} + 0.5\ \text{MB}, \qquad N_{\text{body}} = 10.82\,\text{M}, \quad N_{\text{emb}} = 3.15\,\text{M} \tag{24}$$

Variant (estimates)	Size	Time to build	Expected quality
Ternary 1.58-bit body	about 4.6 MB	3 to 4 days	93 to 97% (loss)
int2 body	5.3 to 6 MB	1.5 to 2 days	90 to 96% (loss)
int3	6.6 to 7 MB	about 1 day	95 to 97% (loss)
int4 (shipped)	8.44 MB	done	about 99%

None of these has been trained. I decided that the saving (1.5 to 4 MB) does not justify the cost for now, so int4 remains the product.

Next steps

When	Goal
Next	Publish to Cloudflare Pages, set the final domain, test on real phones and slow networks
Then	ESP32-S3 hardware demonstration and measurement of real tokens per second
Later	More verified facts, Vulkan backend, Metal and ARM testing (ideas, not promises)

Project layout and release checks

Folder	Contents
Model	the four model files (AFN int4 and int8, GGUF f16 and q8_0), metadata, model card
engine	AEN.cpp native engine, quantisation and export tools, tests, fine-tuning and evaluation scripts, the results log
website	static site, WASM kernels, service worker, headers, generated SEO pages, engine zip download

- Before a release I rebuild the generated pages, run the 36 engine checks and the 103 site checks, rebuild the engine zip, bump the service-worker version and give the model a new file name so that cached copies are never stale.

Small model. Big idea.

Affan AI Engine Nano V3

SUMMARY

I built a 13,965,696-parameter chat model, a hidden calculator tool, two engines and a website, and I compressed the model to an 8.44 MB int4 file. In Version 3 known-fact accuracy rose from 28.1% to 87.9% on unseen wordings, abstention stayed at 67.8%, and the calculator reached 39 of 40. I am equally clear about the limits: it is a small model that can still be wrong, it speaks English only, and it has not yet been measured on real microcontroller hardware.

14M	8.44 MB	~4B	87.9%
parameters	int4 file	tokens trained	known facts

DATA SOURCES AND LICENCES

Filtered SODA (CC BY 4.0) short exchanges; hand-written casual chat; smoltalk everyday conversations; generated identity, calculator and joke data; TinyStories replay; and a closed set of about 1,500 verified facts written by me. Model and code licence: Apache 2.0.

MABA Studios

Muhammad Affan bin Adnan

Thank you for reading.