BabyAGI Inside: Compiling a Bounded Agent Loop into Qwen Weights

Yohei Nakajima · Untapped Capital · Preprint, November 2026

Abstract. We demonstrate that a bounded agent loop can be compiled directly into the weights of a pretrained language model, without training. BabyAGI Inside adds a constructed controller, task-ledger protocol, in-model filesystem and halt condition to Qwen3.5-2B. The loop is a fixed-length task chain: Qwen writes the plan up front, a compiled pointer runs its tasks in order, and the controller halts after N cycles set at build time. The weights implement the loop; the ledger and files it writes live in the run's attention cache and DeltaNet state. After a single objective prefill, the host only samples tokens until a compiled HALT, with no scheduler, parser, tool executor or logits processor. Recent work compiles machines into Qwen that Qwen calls as tools. Here the compiled loop calls Qwen.

In a frozen 32-item live evaluation of two to eight cycles, the compiled controller ran every phase in order and halted in all 32 runs (432/432 phases), with exact token accounting; in a teacher-forced circuit oracle every compiled token matched its reference (1,716 of 1,716 positions). With the loop inactive, the merged model agreed with stock Qwen on 4/4 test prefixes (top-1; maximum KL 0.0019). Every emitted token is attributed to the circuit or to Qwen: in a five-cycle run, Qwen wrote 1,416 of 3,376 tokens, 1,280 of them reasoning.

Inside the loop, Qwen solved all eight sequential-arithmetic chains (36/36 transitions) and none of the 24 chains in other task families: 8/32 overall, against 5/32 for plain Qwen. Structured tasks failed in a revealing way: Qwen wrote a wrong plan and then executed every step of it correctly (8/8 against its own ledger, 0/8 against the reference). Two runs exposed a parser-isolation bug inherited from qwenFS, in which tool-call markup inside Qwen's reasoning put the filesystem into write mode. The prototype establishes that a fixed agent workflow can run inside a pretrained model's weights, and it gives a token-level account of what the compiled loop wrote and what the model wrote.

1 Introduction

An LLM agent is usually two programs: a language model, and host code that loops around it. In BabyAGI (2023), a short Python script kept a task queue, called GPT-4 to execute the next task, called it again to create new tasks from the result, called it a third time to reprioritize, and stored results in a vector database. The model only ever produced text; the loop, the queue and the memory lived outside it.

Recent work shows that machines can be compiled directly into a pretrained model's weights. Qwenputer places a RISC-V CPU inside Qwen3.5-4B, using its Gated DeltaNet state as RAM; qwenFS, qwenGPU, qwenTPU and linuxformer add a filesystem, a GPU, a matrix-multiply array and a Linux-booting CPU. In these systems the model remains the agent: Qwen decides when to call the compiled machine, as it would call a tool.

We ask the reverse question. Can the agent loop itself, the part that is usually host code, be compiled into the weights, with the pretrained model as its subroutine? If so, the host shrinks to a sampler, and the compiled controller schedules the workflow phases and delegates designated generation spans to the model, with no prompts or Python deciding what happens next.

Our contributions:

1. **BabyAGI Inside**, a training-free construction that compiles a bounded, fixed-length BabyAGI-style loop into Qwen3.5-2B: a model-written JSON task ledger, a compiled task pointer, bounded model-owned reasoning, qwenFS persistence and a compiled HALT. After one prefill, the host only samples.
2. **A demonstration and verification method** for compiled agents: forced-token circuit oracles, parity checks, exact emission-ownership accounting, and live traces of up to eight cycles that complete and halt.
3. **A capability account** of what the pretrained model does inside the loop: reliable sequential arithmetic, a reasoning-budget curve, and structured-task failures in which the loop carries out a wrong plan faithfully.
4. **A typed in-weight choice gate**, a secondary primitive: one residual coordinate restricts a model-owned decision to legal tokens with no logits processor.

2 Related work

Compiling programs into transformers. RASP and Tracr [2] compile human-readable programs into transformer weights, mainly as ground truth for interpretability; ALTA [3] adds loops and universal transformers. Looped transformers [4] show a fixed, hand-set network can emulate program counters, branches and an instruction-set computer. Percepta compiled a WebAssembly interpreter into a small purpose-built transformer [14]. These models are built from scratch; none hosts a pretrained language model. Tracr-injection [5] distills compiled algorithms into a pretrained LLM, but by fine-tuning.

Machines compiled into a pretrained Qwen. Our closest neighbours compile machines into Qwen3.5 without training and leave Qwen's own weights unchanged. Qwenputer [6] adds a RISC-V CPU and an in-model filesystem, using Gated DeltaNet state as exact RAM; qwenFS [7], on which we build, adds a filesystem to Qwen3.5-2B; qwenGPU [8] a SIMT GPU that runs CUDA; qwenTPU [9] a matrix-multiply array; and linuxformer [10] a CPU that boots Linux, with Qwen using the shell. In these systems Qwen is the controller and the machine is its tool. The nearest exception is Agrawal's compiled oversight monitor [18], which inserts a flag into Qwen's output and logs it whatever Qwen is told; it interrupts the model rather than running a task loop around it. qwenFS itself already parses tool calls and frames responses in-weight; we lift that control to the level of a whole task workflow.

Internalizing workflows and compiled modules. Dennis et al. [15] compile agentic workflows into a small model's weights by generating procedural conversations and fine-tuning, removing the external orchestrator. Neural Comprehension [16] gates compiled neural modules with a pretrained language model, token by token, for symbolic operations. Our construction is training-free: the controller is specified analytically and schedules repeated generation and storage operations across a multi-cycle task loop.

Program-controlled LLMs. ReAct-style [11] agent loops let the model decide control flow. LMQL [17] constrains generation inside an explicit program. LLM-as-Code [12] argues against model-owned control: a program should own looping and sequencing, and call the LLM only for judgment. BabyAGI Inside adopts that division of labour, but moves the program from the host into the weights.

Constrained decoding and test-time budgets. Grammar-constrained decoding masks illegal tokens in a host-side logits processor. Our typed gate implements the same restriction inside the network. Budget forcing

[13] controls test-time reasoning by ending or extending thinking at a token count; our controller compiles a fixed budget into each execute phase.

3 System

BabyAGI Inside keeps the division of labour of the original script: what Python did is compiled; what GPT-4 did, Qwen does. The workflow is simplified: one up-front plan, fixed-order prioritization, and a cycle count fixed at build.

3.1 Base model and layout

We start from Qwen3.5-2B and the qwenFS merge, which prepends compiled filesystem layers to Qwen's 24 and widens the residual stream. Our controller adds its own registers and a prompt ROM. The merged stream is 4,096 wide: Qwen's native dimensions 0–2047, qwenFS 2048–3071, agent scratch 3072–3208, and padding. Qwen's weights are unchanged apart from the norm-gain rescaling that the widening requires. No parameter is trained. The weights implement the controller and filesystem; the ledger and files a run produces live in its inference state (attention history and DeltaNet state) and persist across cycles within that run; they are gone when it ends.

Terms. *Extended embeddings*: qwenFS encodes file contents and filesystem responses with input embeddings that write to its own lanes as well as Qwen's; stock Qwen has no input for them. *Control symbols*: added tokens the controller emits (START, HALT and phase markers); besides their code in the agent lanes, they carry a view in Qwen's lanes of 0.25 × the newline embedding. *Protocols*: typed (v17), with two task files, one-token prioritization through the typed gate and no reasoning span; and ledger-pointer, with a JSON ledger, a compiled pointer and a 256-token reasoning span (b256). *Checkpoints* m2, m3, m5 and m8 are ledger-pointer builds with N = 2, 3, 5 and 8. *Suites*: held-out v1 (12 items, typed protocol) and the frozen v3 suite (32 objectives, 144 transitions), used for the budget curve, the live evaluation and Table 3.

3.2 The loop

After the objective prefill and a START symbol, the controller steps through compiled phases. In the current protocol (ledger-pointer):

1. **Create.** The ROM prints the creation prompt and opens a `write_file` call; Qwen writes a JSON task ledger. qwenFS stores it.
2. **Prioritize.** qwenFS reads the whole ledger back and the ROM names the next index (“execute ledger task k of N”); Qwen resolves that entry from the ledger it has just read. The order is fixed and owned by the circuit, which never parses the ledger.
3. **Execute.** The ROM opens a reasoning span of exactly 256 tokens, which Qwen fills; spans always run to full length, and the ROM then forces the span closed and opens a write; Qwen writes the result.
4. **Persist.** qwenFS stores `result.txt`; the next cycle reads it back token-exactly.
5. **Halt.** After N cycles the controller emits HALT.

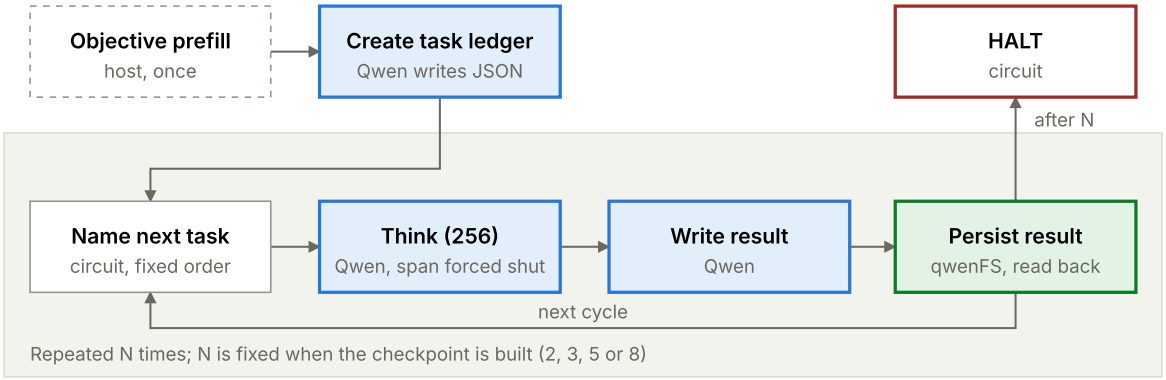


Figure 1. The ledger-pointer protocol: one cycle, repeated N times. Every transition between boxes is compiled; Qwen writes only in the three model-owned boxes (blue, solid border): Create task ledger, Think and Write result. Persist result (green) is qwenFS; Name next task and HALT are compiled controller phases; Qwen resolves the named entry inside Think; the dashed box is the host's one-time prefill.

N is fixed when the checkpoint is built; “bounded” refers to this cycle count. We build N = 2, 3, 5 and 8. An N-cycle run has 3N phases, three per cycle: create (in cycle 1 Qwen writes the ledger; in later cycles the previous result is read back), prioritize and execute, whose result write qwenFS persists. HALT follows the last execute; for N = 8 that is 24 phases.

3.3 A primitive: the typed choice gate

An earlier protocol (typed, v17) let Qwen prioritize with a single legal token. The ROM opens `read_file(task`, Qwen supplies one token, and the ROM closes `.txt)`, so the choice becomes part of a filename. One added residual coordinate is nonzero only for tokens `1` and `2`. At the decision position, a compiled layer activates it and the final norm amplifies it, adding the same large value to both legal logits. Every illegal token loses by at least 1,893 logits; how the gap between `1` and `2` fares under bf16 rounding at this offset is untested. The current protocol uses the compiled pointer instead, after two diagnostic probes in which Qwen chose Task 1 in both orders (Section 6.4).

3.4 The host

The host prefills one objective, then calls `sample()` until HALT. It holds no scheduler, parser, tool executor, retry logic or logits processor. Every prompt, tool call, file response and control token after the objective is emitted by the checkpoint itself. The complete post-prefill runtime:

```
ids = prefill(objective) + [START]
while ids[-1] != HALT:
    ids.append(argmax(model(ids)))   # KV-cached in practice
```

Decoding is greedy (temperature 0). The prefill includes Qwen's tool schema with native thinking disabled, and a run ends at HALT or at an external token limit.

4 Verification methodology

A compiled agent makes two kinds of claim, and we test them separately: that the circuit did exactly what it was built to do, and that Qwen did something useful inside it.

Circuit oracle. We script a full trace, teacher-force it token by token, and require the checkpoint to predict every compiled token exactly: prompts, tool-call wrappers, filesystem responses and control symbols. A separate check confirms that every model-owned reasoning position yields a token from Qwen's ordinary vocabulary rather than a control symbol. Live runs then check the same compiled positions under free decoding.

Parity. With the loop inactive, the merged checkpoint must match stock Qwen3.5-2B: we compare top-1 predictions and KL divergence on four native-text prefixes. This checks that the norm-gain rescaling required by the widening preserves Qwen's predictions.

Token-ownership partition. For every live trace, each emitted token is attributed to exactly one owner: ROM prompt, filesystem response, controller symbol, or model. Model tokens are further split into reasoning, payload and tool-call closure. The partition must account for every token.

Host-versus-compiled equivalence. We run the same protocol with stock Qwen and a Python host, and compare model-owned outputs. Exact equivalence is only expected at the output level: in the reasoning spans, greedy trajectories diverge. We attribute this to small numerical differences in the widened model accumulating over 256 tokens, a hypothesis not yet tested in fp32.

Host proxy, and its limits. A host-side harness with stock weights is fast enough to screen prompts and run paired baselines. It is approximate. The compiled control symbol has a nonzero view in Qwen's native dimensions (0.25 × the newline embedding), which the first harness omitted; qwenFS responses use extended embeddings that stock Qwen cannot consume, so the harness feeds ordinary-token surrogates. Live compiled traces are therefore authoritative for any claim about compiled behaviour.

Scope of the claims. Our validation is empirical and limited to the tested traces, checkpoints and backend (Apple MPS, bf16). A correctness contract over arbitrary model outputs, including a total token bound for every model-owned span, is future work. So is adversarial testing: the ledger and result files are model-written and read back by later cycles, so text imitating a wrapper or HALT is a natural injection channel (Section 5.3).

Preregistration. Benchmarks, prompts, decoding (greedy), token budgets, scorers and trace-selection rules are frozen and checksummed before each run. Superseded results are kept and labelled (Appendix A).

5 Results I: the loop runs as built

The compiled controller ran as built in every live run. The filesystem did in 30 of 32; both exceptions trace to one inherited parser rule (Section 5.3).

- **Circuit oracle:** 1,716/1,716 forced tokens exact on the five-cycle ledger-pointer checkpoint (m5); 1,280/1,280 model-owned reasoning positions yield a native-vocabulary token (id < 248,044, with extended-vocabulary rows suppressed by a compiled gate). The typed v17 checkpoint passed 4,788/4,788.

- **Parity, loop inactive:** top-1 agreement 4/4, maximum KL 0.0019 against stock Qwen3.5-2B.
- **Typed gate:** at all five decision positions of a v17 trace, the best illegal token trails the legal pair by at least 1,893 logits.
- **Host versus compiled:** model-owned ledger and result outputs matched in 6/6 stages; reasoning spans matched token-for-token in 0/5.

5.1 Live compiled evaluation

We ran all 32 items of the frozen v3 suite live on the compiled checkpoints: four task families (ordered literal, sequential arithmetic, structured operation, checkable language), two objectives each, at N = 2, 3, 5 and 8. Decoding was greedy in bf16 on an Apple M5 with a frozen runner. A run passes the controller check when its phases occur in the compiled order and HALT follows the final cycle; it passes the filesystem check when every read returns the latest prior write token-exactly; it passes the benchmark when every result it writes equals the reference value for that step. All checks are scored automatically by the frozen scorer.

Table 1. System checks on the frozen 32-item live evaluation.

Check	Passed
Controller: phases in compiled order	32/32 runs; 432/432 phases
Compiled HALT after the final cycle	32/32
Exact token accounting	32/32
Filesystem: every read exact	30/32
All system checks	30/32

5.2 Who wrote the tokens

Table 2 splits every token of the four add-first runs and the typed v17 run by owner. Ownership records who selected each emitted token, not each part's share of the run's competence. Under the ledger-pointer protocol Qwen writes 39–42% of tokens, nearly all of them reasoning, a share set by the fixed 256-token budget per cycle. Under the typed protocol, with no reasoning span, Qwen wrote 241 of 5,010 tokens (4.8%), and its five prioritization decisions were one token each. Live forced counts differ from oracle counts because qwenFS responses echo model-written content, so their length varies by run; for v17 the 19-token gap (4,788 versus 4,769) is entirely response length.

Table 2. Token ownership by checkpoint (ledger-pointer: arithmetic-add runs; typed: v17 five-cycle run).

Checkpoint	Oracle forced	Live forced: ROM / qwenFS / controller	Qwen: reasoning / payload / closure	Total	Qwen share
ledger-pointer m2	840	693 / 193 / 6 = 892	512 / 41 / 30 = 583	1,475	39.5%
ledger-pointer m3	1,132	911 / 313 / 9 = 1,233	768 / 53 / 40 = 861	2,094	41.1%
ledger-pointer m5	1,716	1,347 / 598 / 15 = 1,960	1,280 / 76 / 60 = 1,416	3,376	41.9%
ledger-pointer m8	2,592	2,001 / 1,121 / 24 = 3,146	2,048 / 112 / 90 = 2,250	5,396	41.7%
typed v17 (5 cycles)	4,788	4,163 / 582 / 24 = 4,769	0 / 91 / 150 = 241	5,010	4.8%

5.3 A parser-isolation bug inherited from qwenFS

In two eight-cycle literal runs, reads returned the latest write plus extra text: 78 bytes became 1,086 in one, 45 became 80 in the other. The cause is a rule in qwenFS's compiled parser. It recognizes the start of file text from a token pattern (`=text>` followed by a newline) without checking that a tool call is open. When Qwen's reasoning contained imitation tool-call markup, the filesystem entered write mode and copied the following tokens into the stored file until the next genuine tool-call end. The rule is identical in upstream qwenFS (`fe3f5c0`).

Across the evaluation, 52 of 608 reasoning spans contained tool-call markup; 32 spans in ten runs activated write mode; two produced corrupted reads. The controller was unaffected: both runs completed every phase and halted. Neither failure changed a benchmark verdict, since both runs were wrong from the first cycle. The exact-read check catches every corrupted read but not a dormant activation that never reaches a read. The fix gates text-start on the real call state; it requires rebuilding the checkpoints and repeating the oracle and live runs. This is the injection channel named in Section 4, observed without adversarial prompting.

5.4 Cost

On an Apple M5 (MPS, bf16), the five-cycle typed (v17) trace ran at about 10.4 tokens per second (5,010 tokens in 481 s). Building a 2B checkpoint takes about 32 s and peaks near 14.6 GB of memory; each ledger-pointer checkpoint is 8.8 GB on disk. The compiled loop pays one decode step for every forced token, where a host loop could prefill known prompt text in a single pass; packaging the workflow in the checkpoint costs throughput.

6 Results II: capability is narrow

Inside the loop, Qwen solved every sequential-arithmetic chain and no chain in any other family. Its structured-task plans were wrong, and it executed them correctly.

6.1 Reasoning budget (exploratory)

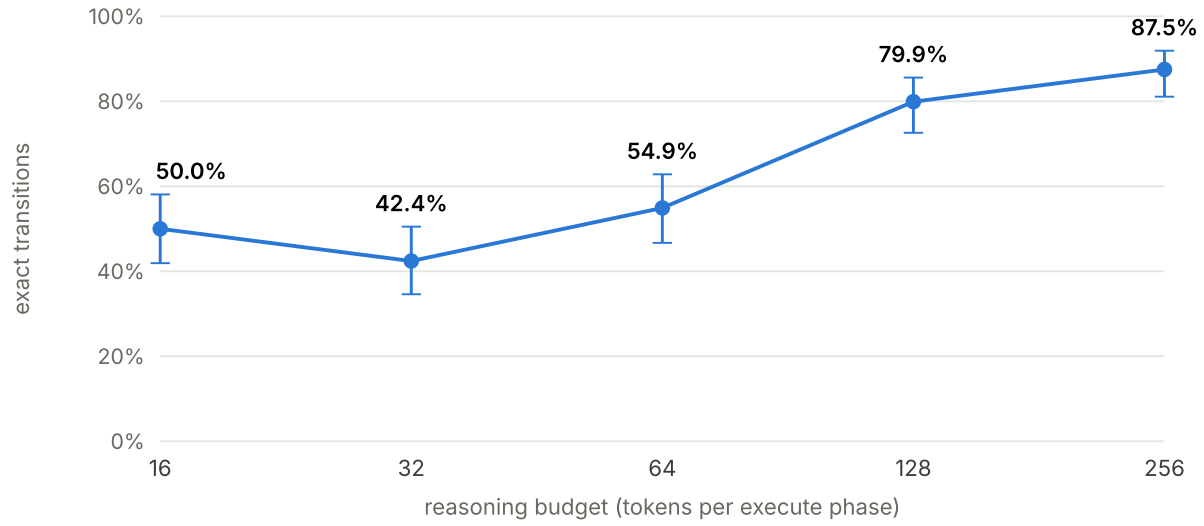


Figure 2. Transition accuracy by reasoning budget (exploratory): 144 transitions per budget from the frozen v3 suite, Qwen3.5-2B, greedy, bf16 on MPS. Budgets tested: 16, 32, 64, 128 and 256 tokens. Whiskers are 95% Wilson intervals treating transitions as independent; they are not clustered by objective and so understate uncertainty. The y-axis runs from 0 to 100%.

More reasoning lifted transition accuracy under the tested protocol, from 42–55% at 16–64 tokens to 80–88% at 128–256. Each transition starts from the correct task and incoming state, so the curve measures conditional execution, not planning. Gains are largest between 64 and 128 tokens; 128 to 256 is a modest further gain (17 improvements, 6 regressions; paired $p = 0.035$ unadjusted, which does not survive adjustment across the budget contrasts, and the objective-clustered interval touches zero). The curve is not monotonic: a fixed span changes the model's trajectory, so more tokens can make an answer worse. At 256 tokens, per-step accuracy is 91.7% on ordered literals, 94.4% on arithmetic, 86.1% on structured operations and 77.8% on constrained language edits, from 36 transitions each; the family ranking is suggestive, and per-family budget curves are future work. We froze 256 as the protocol budget.

6.2 Whole chains

Per-step accuracy does not carry over to whole chains. The compiled loop solved 8 of 32 chains (36/144 transitions), plain Qwen 5 (23/144) and a host-side proxy of the loop 7 (35/144) (Table 3). Every success in every arm is sequential arithmetic. Against plain chat the compiled loop gains three arithmetic chains with no regressions (exact McNemar $p = 0.25$): consistent with a benefit on arithmetic, not evidence of one. Transitions here are unconditioned: each step is scored on the chain as the arm actually ran it, unlike the conditioned transitions of Section 6.1. Plain Qwen ran with thinking enabled and a ceiling of $N \times 256 + 256$ new tokens, the loop's thinking budget plus one answer. The loop may also spend up to 256 tokens on each of its $N + 1$ writes, so its maximum model budget is larger (4,352 against 2,304 tokens at $N = 8$).

Table 3. Exact chains and exact transitions on the frozen v3 suite, by family (8 chains and 36 transitions per family).

Family	Compiled loop, live	Plain Qwen, one turn	Host proxy of the loop
Ordered literal	0/8 · 0/36	0/8 · 0/36	0/8 · 0/36
Sequential arithmetic	8/8 · 36/36	5/8 · 23/36	7/8 · 34/36
Structured operation	0/8 · 0/36	0/8 · 0/36	0/8 · 0/36
Checkable language	0/8 · 0/36	0/8 · 0/36	0/8 · 1/36
Total	8/32 · 36/144	5/32 · 23/144	7/32 · 35/144

The proxy tracked the live loop closely, 7/8 against 8/8 on arithmetic. Its one miss was the five-cycle multiply-first chain, where it stalled while closing its third write and the live checkpoint produced 30, 37, 33, 66, 73 exactly. A first proxy run reported 9/32; it is withdrawn (Appendix A).

6.3 A wrong plan, executed faithfully

Structured chains failed at every length (0/8), and the eight-cycle runs show why. In both, Qwen's ledger set the first task to add a combined amount (18 or 19) and every later task to add one, applied elementwise to the list. Every result Qwen then wrote correctly executed its own ledger task (8/8 in each run), and none matched the reference (0/8). The controller pointed to each task in order and persisted each result. Planning failed; execution and orchestration did not.

Plan quality is also hard to score. All 32 proxy ledgers parsed and 31/32 extracted the correct initial value, but exact task-string match was 0/32, because close paraphrases such as “multiply by 2” do not equal “multiply the result by 2”. We keep that frozen metric and will report lexical and semantic plan fidelity separately in future suites.

6.4 What did not work

- **No reasoning span:** the typed v17 protocol, with 2–8-token payloads and no thinking, solved 0/12 held-out items.
- **Few-shot exemplars:** format-only examples fixed first-task shape but leaked content; a “launch note” exemplar became an attractor for a launch-announcement objective.
- **A model-maintained ledger lifecycle:** when Qwen rewrote the whole ledger each phase, it left completed tasks pending and regenerated candidate lists instead of executing.
- **Model-owned prioritization:** with the typed gate, Qwen chose Task 1 in both orders of a counterbalanced probe (n = 2), a positional bias. The current protocol uses a compiled pointer instead.

7 Discussion and limitations

What compiling the loop buys. The workflow ships inside the checkpoint: loading the model and sampling is the whole runtime. Control flow is fixed by construction rather than by prompts or host code, and on every tested trace the loop ran its phases in order, persisted each result and halted. Every emitted token has an owner, so the model's share of a run is measurable (about 40% under the ledger-pointer protocol, 4.8% under the typed one), and failures localize: the structured runs show a wrong plan carried out step by step by a loop that ran as built, and the one fault that crossed component boundaries traced to a single parser rule.

What it does not buy. This study shows no capability gain from compilation. Exact orchestration around a 2B model still yields narrow capability: the loop solved 8/32 chains against 5/32 for plain chat, all of them arithmetic, and the difference is not significant. Compiled control is also harder to inspect than host code: its behaviour is verified by oracle and trace, not read from source.

Limitations.

- **Bounded loop.** N is fixed at build time; each length needs its own checkpoint, and the controller does not decide when to stop.
- **Compiled prioritization.** The current protocol's task order is fixed, and Qwen resolves each entry from the ledger itself; the controller never parses it. BabyAGI's prioritization agent is represented only in the earlier typed protocol, where two probes suggested a positional bias.
- **Up-front planning.** Qwen writes the whole ledger at the start. The original BabyAGI created tasks incrementally from each result, which may matter most for the structured failures.
- **Output agreement only.** Outputs agreed in the six stages tested; reasoning traces differed.
- **Oracle coverage.** The circuit oracle was run on the m5 and v17 checkpoints; m2, m3 and m8 are verified by live runs only.
- **Approximate proxy.** Host-side baselines cannot consume qwenFS's extended embeddings exactly.
- **Small samples.** 32 chains cannot detect modest effects. Malformed model output reached the filesystem in two runs (Section 5.3); empty and non-closing outputs remain untested.
- **One model, one machine.** All results are for Qwen3.5-2B on an Apple M5 (MPS, bf16). Related projects found bf16 bugs that appeared only on particular GPUs; we have not yet replicated on CUDA.

Future work. The parser fix of Section 5.3, with a regression test that places tool-call markup inside reasoning, comes first. A program-counter controller held in Gated DeltaNet state, as in Qwenputer and linuxformer, would remove the cycle bound and allow model-decided halting through the typed gate. It depends on two untested properties: the gate's margin under bf16, and counter state surviving free generation that writes to the same recurrent state. Incremental task creation and a larger, new frozen suite would test whether the loop can help planning. Fine-tuning only Qwen's dimensions, with the compiled parts frozen, would test how much of the gap is the pretrained model's. Giving both loops a correct reference plan and the same model-written plan would measure the planning bottleneck directly.

8 Conclusion

An agent loop can be compiled into a pretrained model's weights without training. In BabyAGI Inside, the controller and filesystem live in Qwen3.5-2B's weights, the ledger and files in its inference state, and the host only samples. In a 32-item live evaluation of up to eight cycles, the controller ran as built and halted in every run. Inside it, Qwen solved every arithmetic chain and wrote wrong plans for structured tasks, which it then executed step by step. Fixing orchestration by construction makes every token the model wrote, and the point where each failure arises, directly visible.

Appendix

A. Superseded results

- **Host loop 9/32 (withdrawn).** The first paired run omitted the compiled control symbol's view in Qwen's native dimensions (0.25 × the newline embedding). Corrected, the proxy scores 7/32; its two structured successes were false positives. The 9/32 run is kept as an audit artifact (commit `a5fbf2f`; corrected in `14f1c0f`).
- **Held-out v1, 0/12 (typed protocol, no reasoning span).** Superseded as a capability measure by the budget study and the v3 suite.
- **Pilot budget curve (32 transitions).** A sampling bug was found before cross-budget comparison; those outputs were excluded and the full 144-transition run replaces them.

B. Reproduction

Code, build scripts, trace logs and the frozen suite: github.com/yoheinakajima/babyagi-inside (compiler commit `48f4f45`; qwenFS baseline `fe3f5c0`; Qwen3.5-2B weights pinned by SHA-256 in `provenance/qwen3.5-2b.sha256`). Checkpoints, rebuilt locally rather than distributed: `build/babyagi-2b-ledger-pointer-b256-m{2,3,5,8}`. Results: `results/babyagi-v3-full-live-compiled/` (32-item live evaluation, runner commit `48f4f45`), `results/end-to-end-v3-b256/` (paired run and ten live traces), `results/end-to-end-v3-b256-control-surrogate/` (corrected proxy), `results/execution-budget-v3-full/` (budget curve). Frozen v3 suite SHA-256 `e880e043f7ac412aaab938036cccd5f79305aed4cd0c13a7204e09fe07c1dff4`.

```
uv run python -m qfs.end_to_end models/Qwen3.5-2B \
  benchmarks/babyagi-v3.json \
  results/end-to-end-v3-b256-control-surrogate \
  --think-tokens 256 --plain-answer-tokens 256 \
  --control-surrogate-scale 0.25 \
  --plain-from results/end-to-end-v3-b256 --device mps
```

Build (`scripts/build_checkpoints.sh`), circuit-oracle and live-decode commands for the ledger-pointer checkpoints are in the repository, with per-token ownership logs for the ten live runs in `traces/`; the five-cycle multiply-first run should reproduce the result sequence 30, 37, 33, 66, 73.

C. Selected live runs

Table C1. Selected live runs on the ledger-pointer checkpoints (greedy, bf16, Apple M5).

Item	Cycles	Controller, files, HALT	Benchmark	Result sequence
arithmetic-add-2	2	pass	pass	9, 18
arithmetic-add-3	3	pass	pass	11, 22, 19
arithmetic-add-5	5	pass	pass	15, 30, 27, 32, 96
arithmetic-add-8	8	pass	pass	21, 42, 39, 44, 132, 124, 248, 257
arithmetic-multiply-2	2	pass	pass	21, 25
arithmetic-multiply-3	3	pass	pass	24, 29, 25
arithmetic-multiply-5	5	pass	pass	30, 37, 33, 66, 73
arithmetic-multiply-8	8	pass	pass	39, 49, 45, 90, 97, 194, 185, 196
structured-even-8	8	pass	fail	wrong ledger; 0/8 transitions
structured-odd-8	8	pass	fail	wrong ledger; 0/8 transitions

These ten runs were inspected before the full evaluation; the four add-first runs were chosen by a preregistered rule and the other six afterwards, for diagnosis. All ten are among the 32 items of Table 1.

References

1. Nakajima, Y. BabyAGI (2023). github.com/yoheinakajima/babyagi
2. Lindner et al. Tracr: Compiled Transformers as a Laboratory for Interpretability. NeurIPS 2023. arXiv:2301.05062
3. Shaw et al. ALTA: Compiler-Based Analysis of Transformers. TMLR 2025. arXiv:2410.18077
4. Giannou et al. Looped Transformers as Programmable Computers. ICML 2023.
5. Tracr-Injection: Distilling Algorithms into Pre-trained Language Models. arXiv:2505.10719
6. Agrawal, D. Qwenputer (2026). github.com/divagr18/qwenputer
7. Baraiya, V. qwenFS (2026), commit fe3f5c0. github.com/vixhal-baraiya/qwenFS
8. Baraiya, V. qwenGPU (2026). github.com/vixhal-baraiya/qwenGPU
9. Bobliuuu. qwenTPU (2026). github.com/Bobliuuu/qwentpu
10. itsrealranky. linuxformer (2026). github.com/itsrealranky/linuxformer
11. Yao et al. ReAct: Synergizing Reasoning and Acting in Language Models. ICLR 2023. arXiv:2210.03629
12. LLM-as-Code: Agentic Programming for Agent Harness (2026). arXiv:2606.15874
13. Muennighoff et al. s1: Simple test-time scaling (2025). arXiv:2501.19393
14. Percepta. Can LLMs Be Computers? (2026). Blog post.
15. Dennis, S., Patil, R., Shabahang, K., Guo, H. Compiling Agentic Workflows into LLM Weights (2026). arXiv:2605.22502
16. Weng et al. Mastering Symbolic Operations: Augmenting Language Models with Compiled Neural Networks. arXiv:2304.01665

17. Beurer-Kellner et al. Prompting Is Programming: A Query Language for Large Language Models (LMQL). PLDI 2023. arXiv:2212.06094
18. Agrawal, D. Building oversight tools into a model's weights (2026). divagr.com