polarjsapkota@gmail.com | Feb 2026

FUNGIBLE SOFTWARE IS ALREADY POSSIBLE, But it is NOT EVEN REMOTELY CLOSE to being FEASIBLE !

A two-page case for a deterministic program structure index [with appendices]

THE PROBLEM

Inference through LLM code-generation will burn approximately 225 GWh of energy in 2026.

SOLUTION

- Generate code once, verify it through assertion feedback loops, and index the verified computational structure into a massive **PSI** (Program-Structure Index)
- Do it for all structures that drive the world today. From Apps to Operating Systems

GOAL

Make fungible software **feasible**. Once a program structure is indexed, subsequent requests for the same become a look-up, costing approximately **2.1 Joules** to generate 10k lines of code.

That’s almost a **10^5 reduction in energy per request**.

THE BELIEF

Reading through articles of when the Perceptron came out and became popular back in the 1950s, today we face the day of reckoning in what seems to be aptly called a self-fulfilling prophecy. Software will now augment human thinking by orders of magnitude while quietly sitting in the background consuming quite literally, orders of magnitude less energy.

Polarj Sapkota (2026)
github.com/antiresonant/nlc

The argument.

Every program is a relation between an input and an output.

A Fibonacci function takes a number and returns a number. A sort takes a list and returns a list. An authentication module takes credentials and returns a session. Strip away the variable names, the syntax, the language and what remains is the relation itself: a mapping from a Domain (input space) to a Range (output space), with constraints that guarantee correctness.

*That **relation, not function**, is the program's identity.*

This is the core idea behind NLC. Each step in an algorithm is expressed as a Domain/Range assertion: "given *this* kind of input, the output *must* satisfy *these* constraints." An LLM generates code. The assertions run. If any fail, the error is fed back and the LLM regenerates only the failing step, not the whole program. This loop continues until every assertion passes. The result is code that is **deterministically correct by construction**.

Now here is the insight that changes the economics. Once a verified relation exists, say, "sort an array of integers in ascending order" with its Domain and Range constraints satisfied, it never needs to be generated again. It can be **indexed**. Not the code but the verified computational structure. The next time any user, anywhere, needs a sort with identical constraints, the index returns the structure and projects it into whatever language or style is required.

Generation cost: one time & big. Retrieval cost: near zero.
The LLM becomes a one-time compiler; the index becomes the runtime compiler.

This is what makes software **fungible**. Two programs with identical Domain/Range specifications being interchangeable no matter what underlying language and hardware the code runs on.

After this index is complete, software will become a true commodity.

The energy implications are stark (Appendices A and B). Today, every code-generation request fires an LLM from scratch. Tens of trillions of tokens per year (counting input context, output, and retries), each costing ~2 joules. An indexed lookup costs **10 microjoules per token:** five orders of magnitude less. But energy is only the near-term argument. The deeper consequence is architectural: once verified program structures are indexed and retrievable, the economics of software shift from generation to retrieval. Software can be created to satisfy a task and discarded the moment the task is done, because recreating it is instantaneous and virtually free. The world moves from prompting and iterating to "describe what you need, receive a verified program, move on." A new paradigm of software awaits us. This is essentially the Relational Database, PageRank-esque moment for software itself, and I wish to be backed by the people who most innately understand shifts of such scale.

Polarj Sapkota (2026)
github.com/antiresonant/nlc

Appendix A: Software Generation: LLM vs. Deterministic Index

The energy gap between LLM generation and indexed retrieval grows dramatically with code size. Below we compute the energy for three representative scales of software.

Assumptions and Sources

Code density – 30 output tokens per accepted line of code. This accounts for ~10 raw tokens/LOC plus comments, docstrings, imports, boilerplate, and formatting tokens that accompany real codegen output (range: 20–60 tokens/LOC in practice)

LLM output energy - 2 J/token

- ***Realistic:*** *2 J/token (figure from Epoch AI 2025 est. for frontier models with ~10% GPU utilization)*
- ***Best case:*** *0.4 J/token (H100, FP8, high batching per Lin 2025)*

NLC iteration overhead - Avg 3.0 iterations (accounts for retries from compile/test failures, user-requested changes, partial completions, and refactors — real-world range 2–10×) (*NLC 2026*)

Input/context overhead - Input/context overhead — 4× the output token count per call. This accounts for system prompts (~500–2,000 tokens), user instructions, retrieved file context, conversation history, tool outputs, and the growing prefix during decoding. GitHub Copilot Chat uses 64k–128k token context windows; agent-mode sessions routinely have input > output.

Index lookup - Hash-based structure matching at:

- *0.01J per fragment (code block) + 0.1 J base cost for query parsing and assembly*
- *~ 50 lines per fragment*

Table 1: Energy Cost of Software Generation. LOC = Lines of code

Scale	LLM (realistic)	LLM (best-case)	NLC Index	Savings Factor
100 LOC *Small utility*	90 kJ	*18 kJ*	***0.12 J***	**750,000× (real)** **150,000× (best)**
1,000 LOC *Medium application*	900 kJ	*180 kJ*	***0.3 J***	**3,000,000× (real)** **600,000× (best)**
10,000 LOC *Large System*	9,000 kJ	*1,800 kJ*	***2.1 J***	**4,286,000× (real)** **857,000× (best)**

Minimum 5-orders of magnitude improvement in energy efficiency

Computation Details

1. Small utility (100 lines — CLI tool, single-purpose script)

LLM path	= 100 LOC × 30 tokens/LOC = 3,000 output tokens = 3,000 output tokens × 5 (input/context multiplier) × 3.0 iterations × 2 J/token = 90 kJ
Index path	= 2 fragments × 0.01 J + 0.1 J base = 0.12 J

Energy reduction: 750,000×

2. Medium application (1,000 lines — REST API, web app backend)

LLM path	= 1,000 LOC × 30 tokens/LOC = 30,000 output tokens = 30,000 output tokens × 5 (input/context multiplier) × 3.0 iterations × 2 J/token = 900 kJ
Index path	= 20 fragments × 0.01 J + 0.1 J base = 0.3 J

Energy reduction: 3,000,000×

3. Large system (10,000 lines — Full-stack SaaS, enterprise module)

LLM path	= 10,000 LOC × 30 tokens/LOC = 300,000 output tokens = 300,000 output tokens × 5 (input/context multiplier) × 3.0 iterations × 2 J/token = 9,000 kJ (9 MJ)
Index path	= 200 fragments × 0.01 J + 0.1 J base = 2.1 J

Energy reduction: 4,286,000×

Appendix B: Economics

Dario Amodei, the CEO of Anthropic in March 2025 predicted 90% of code would be AI written within 6 months. Taking this figure at face value, we attempt to estimate the total energy cost of AI code generation worldwide.

Step 1: Estimating accepted lines of code per year

The best pre-AI estimate comes from Sage McEnery (2020), estimating software engineers produce about 93 billion lines of code per year. GitHub now has 100M+ developers on the platform. Greptile's State of AI Coding 2025 report shows lines per developer grew 76%, from 4,450 to 7,839 LOC/year. A reasonable 2026 estimate is 300 billion accepted lines of code per year.

Step 2: Corrected token accounting model

The naïve approach (LOC × tokens/LOC × J/token) grossly underestimates energy because it counts only output tokens. Epoch AI's detailed analysis (Feb 2025) shows that even a single GPT-4o query consumes ~0.3 Wh (~1,080 J) for ~500 output tokens, corroborated by Sam Altman. But for codegen, the real cost comes from four token categories:

(a) Output tokens: the generated code itself. At ~30 tokens per accepted LOC (accounting for comments, docstrings, imports, boilerplate, formatting — range 20–60 in practice): 300B × 30 = 9 trillion output tokens.

(b) Input/context tokens: system prompts (500–2,000 tokens), retrieved file context, conversation history, tool outputs, and KV-cache prefix during decoding. GitHub Copilot Chat uses 64k–128k token context windows. Epoch AI finds that a 10k-token input costs ~2.5 Wh per query; a 100k-token input costs ~40 Wh — a 130× increase over the base query cost. Agentic codegen sessions routinely have input > output. Conservative multiplier: 4× output = 36 trillion input tokens.

(c) Retry/regeneration tokens: compile/test failures, user-requested changes, partial completions, refactors, "continue" prompts. Conservative multiplier: 3× total (range 2–10× depending on workflow). With 45T tokens per iteration × 3 iterations = 135 trillion tokens.

(d) Non-LOC token burn: explanations, diffs, test output analysis, debugging conversations, code review, documentation generation, abandoned/discarded generations — work that never becomes "lines of code." Solovyeva and Castor (2026) found that 3 out of 10 code-generation models exhibit "babbling behavior," adding excessive content to output that unnecessarily inflates energy consumption. Suppressing babbling alone achieved 44–89% energy savings. Conservative estimate: 2× the LOC-anchored total = 270 trillion additional tokens.

Step 3: Bottom-up energy estimate (LOC-anchored)

Using only categories (a)–(c), anchored to accepted LOC:

135 trillion tokens × 2 J/token = 270 TJ = 75 GWh/year

This is a lower bound. It counts only tokens directly tied to producing accepted code.

Step 4: Full-scope estimate (including non-LOC token burn)

Adding category (d) — the non-LOC overhead of explanations, debugging, discarded generations, reviews, babbling, and documentation — doubles the token volume:

(135 + 270) trillion = 405 trillion total tokens × 2 J/token = 810 TJ = 225 GWh/year

Step 5: Prefill/decoding phase accounting

Solovyeva and Castor (2026) conducted phase-level energy analysis and found that prefill costs amplify the energy cost per token during decoding by 1.3–51.8% depending on the model. This means our 2 J/token figure, which is derived from total-query energy divided by output tokens, already partially captures prefill overhead. However, for codegen — where context windows are much larger than average chat queries — the prefill amplification effect is at the upper end. Epoch AI confirms this: at 100k input tokens, query energy reaches ~40 Wh (vs. 0.3 Wh for a short query), a 130× increase driven almost entirely by prefill. Our 4× input multiplier in Step 2(b) is therefore conservative for agentic workflows.

Step 6: Cross-validation against published data

MIT Technology Review (May 2025) reports that at 1 billion ChatGPT queries/day, that single product alone consumes ~109 GWh/year. AI-specific servers in US data centers used an estimated 53–76 TWh in 2024 (LBNL/DOE), with 80–90% going to inference. SCSP (2024) modeled US software engineers alone with 5 coding agents each and reached a theoretical ceiling of ~536 TWh — an extreme upper bound assuming 24/7 max-throughput operation.

Our 225 GWh estimate sits in a plausible range: far above the naïve 1.67 GWh, well below the SCSP extreme, and consistent with code generation being a single-digit percentage of total AI inference. A reasonable confidence interval for 2026 codegen inference energy is 75–500 GWh, with our central estimate at ~225 GWh.

Revised estimate: AI code generation will burn approximately 225 GWh in 2026.

Step 7: With NLC index (assuming ~80% cache hit rate on common patterns)

The 80% of requests served from the index consume ~10.8 MJ total. 20% still require LLM generation = 45 GWh. Net savings: ~180 GWh/year.

New effective rate of 0.4 J/token (weighted average: 20% at 2 J + 80% at near-zero)

In the limit, using the 0.01 J/fragment (50 LOC/fragment) assumption, total index energy consumption is ~25 kWh/year for all 300B LOC. Conservatively, < 30 kWh/year in the limit.

Only 30 kWh for the whole world to generate fungible software at will.

This is equivalent energy to 4 people charging their smartphones across a year.

Step 8: Second-order effects — the cascading energy dividend

The 225 GWh figure captures only the direct cost of code generation inference. But code generation creates cascading downstream energy effects that compound multiplicatively across the entire software lifecycle:

(a) Runtime energy penalty of generated code. Islam et al. (2025) benchmarked 20 LLMs against human-written solutions on 878 LeetCode problems and found that LLM-generated code is systematically less energy-efficient at runtime: the best models (DeepSeek-v3, GPT-4o) produce code consuming 1.17–1.21× more runtime energy than human-optimized solutions. The worst models (Grok-2, Gemini-1.5-Pro) consume over 2× more. For specific algorithmic categories — dynamic programming, backtracking, bit manipulation — LLM code can consume up to 450× more runtime energy than canonical solutions. This means every AI-generated application, API, or data pipeline deployed into production carries a permanent energy tax that runs 24/7. The code is written once; the runtime penalty runs forever.

(b) Testing and CI/CD multiplication. Generated code triggers automated test suites, linting, security scans, and build pipelines. Each commit from an AI agent triggers full CI/CD runs. With AI generating 76% more code per developer (Greptile 2025), the volume of CI/CD cycles scales proportionally. GREEN-CODE (Ilager et al. 2025) showed that even at the inference level, RL-based early-exit optimization can reduce codegen energy by 23–50% — but NLC eliminates the inference entirely for cached patterns.

(c) AI-reviewing-AI loops. AI-generated code must be reviewed — increasingly by other AI systems. Greptile, CodeRabbit, and Copilot all offer AI code review. This creates a second wave of inference: AI reviewing AI-generated code, each review consuming its own token budget. Solovyeva and Castor (2026) found that prefill costs from reviewing long code contexts amplify per-token decoding energy by up to 51.8%, making code review disproportionately expensive.

(d) Documentation and maintenance chains. Generated code requires generated documentation, generated tests, generated migration scripts. Each is itself a codegen task with its own token overhead, creating a multiplicative chain. The babbling behavior identified by Solovyeva and Castor

— where models add excessive, unnecessary content — means these secondary tasks may burn 44–89% more energy than strictly necessary.

(e) Jevons paradox. As code becomes cheaper to produce, more code gets produced. SCSP (2024) notes that efficiency gains in coding may paradoxically cause engineers to spend more time coding, not less, because "there is more to be coded." The 300B LOC/year estimate may itself be conservative if AI reduces the marginal cost of software to near zero.

(f) Industry convergence toward retrieval. GitHub's new Copilot embedding model (Oct 2025) demonstrates the industry trajectory: 37.6% better retrieval quality, 8× smaller index, 2× throughput, and code acceptance rates doubling for C# (+110.7%) and Java (+113.1%). GitHub trained this model using contrastive learning with hard negatives — code that looks correct but isn't — to serve verified, contextually precise snippets via semantic search. This is PSI-adjacent infrastructure: the industry is already building the retrieval layer that NLC would operate on. The difference is that Copilot still generates each response via LLM inference on retrieved context; NLC eliminates the generation step entirely for indexed patterns.

These cascading effects mean that the true energy footprint of AI code generation is not 225 GWh — it is 225 GWh of generation energy, plus a permanent runtime energy tax on every deployed application (1.17–450× overhead per Islam et al.), plus the multiplicative token burn of retries, reviews, documentation, and babbling at every stage of the cascade.

Quantifying the cascade: If even 10% of the 300B generated LOC reaches production, and the average runtime energy penalty is a conservative 1.5× over optimal, the ongoing runtime overhead alone dwarfs the one-time generation cost within months. Factor in CI/CD multiplication, AI code review, documentation generation, and Jevons-driven volume growth, and a plausible total energy footprint of AI-assisted code (generation + downstream) reaches the low single-digit TWh range — consistent with broader estimates of AI inference energy.

This is precisely the argument for NLC. By indexing verified, optimized program structures and serving them via lookup:

• Generation energy: eliminated for cached patterns (225 GWh → ~45 GWh at 80% hit rate, approaching 30 kWh in the limit)

• Runtime energy penalty: eliminated. NLC serves human-verified, algorithmically optimal code — no 1.17–450× runtime tax

• Retry/babbling overhead: eliminated. Deterministic lookup produces correct output on first retrieval

• Review overhead: eliminated. Indexed patterns are pre-verified

- Documentation: co-indexed with code fragments, served at lookup cost

The energy reduction compounds multiplicatively across the entire software lifecycle. NLC does not merely reduce generation cost — it collapses every stage of the cascade.

Step 9: Theoretical foundation — from program synthesis to program structure indexing

NLC builds on decades of program synthesis research. The formal program synthesis tradition — from Church (1957) through Manna and Waldinger (1980) to Solar-Lezama's Counterexample-Guided Inductive Synthesis (CEGIS) — seeks programs that provably satisfy specifications. Srivastava, Gulwani, and Foster (POPL 2010) showed that program verification and program synthesis are two sides of the same coin, with verification conditions guiding the synthesis search.

NLC inverts the traditional synthesis pipeline: rather than synthesizing programs from specifications on demand (which requires expensive search), NLC pre-indexes verified program structures and serves them via embedding-based retrieval. The verification has already been done. The specification matching happens through semantic similarity in embedding space — the same contrastive learning and hard-negative training that GitHub's Copilot embedding model uses, but applied to an index of formally or empirically verified program fragments rather than raw repository code.

In formal terms: traditional CEGIS loops through generate → verify → counterexample → generate until convergence. Each iteration costs O(inference). NLC pre-computes the converged solutions and serves them at O(lookup). The energy cost shifts from O(n × inference) to O(1 × indexing) + O(n × lookup), where lookup costs microjoules per fragment vs. joules per token.

This is the path from 225 GWh to 30 kWh: not optimization of inference, but elimination of inference through pre-verified retrieval.

New effective rate of 10 microjoules/token.

The merits of determinism

Appendix C: Pattern detection with Regex vs. LLM as a reference

Consider a simple task: detecting whether a string matches an email pattern. This is the kind of operation performed billions of times per day across the internet.

- **A compiled regex** executes in approximately 500 nanoseconds on modern hardware, consuming roughly 42.3 μJ of energy (65W CPU, PUE 1.3)
- **An LLM generating that same regex** requires ~50 output tokens.
  At the realistic rate of 2 J/token, that costs 100.0 J. And this is before the regex even runs

Table 2: Energy Cost of Pattern Matching

Operation	Time	Energy	Ratio to Regex	Notes
Regex match	500 ns	**42.3 μJ**	1× (baseline)	DFA-style, compiled
LLM writes regex (best)	0.5 sec	20.0 J	473,000×	H100, FP8, high batch
LLM writes regex (real)	2 sec	100.0 J	2.4M×	Typical API inference
LLM writes regex (pess)	4 sec	200.0 J	4.7M×	Older GPU, low batch

The pragmatic argument: Yes, a smart system will have the LLM write the regex once and then reuse it. This is exactly what we're trying to do with NLC's thesis. Generate all possible structures once then reuse it indefinitely.

The question is: why stop at regex?
The same principle—generate once, verify, index, reuse—applies to entire programs.

Methodology Notes

- **LLM energy figures** are derived from Samsi et al. (2023) empirical measurements on LLaMA-65B (3–4 J/token on V100/A100), Lin (2025) measurements on LLaMA-3.3-70B with H100/FP8 (~0.39 J/token), and Epoch AI's analysis of GPT-4o inference (~0.3 Wh per 500-token query, corroborated by Sam Altman). The realistic estimate of 2 J/token accounts for typical GPU utilization rates (~10%) and datacenter overhead. Token accounting uses 30 output tokens per accepted LOC (range 20–60), a 4× input/context multiplier (system prompts, retrieved files, conversation history, tool outputs — consistent with GitHub Copilot's 64k–128k context windows), and a 3× retry/iteration multiplier (real-world retries from compile failures, user changes, partial completions).
- **Index lookup energy** is estimated conservatively. A hash-based lookup on modern hardware costs ~50 ns of CPU time (~4 μJ). We use 0.01 J per fragment to account for SSD I/O, network overhead, and matching logic—roughly 2,000× more than the bare CPU cost. The 0.1 J base cost covers query parsing and result assembly.
- **Regex energy** is based on DFA-style compiled regex execution benchmarks (CTRE: 105–384 ns, libstdc++: ~1,164 ns per email validation). We use 500 ns as a middle estimate on a 65W CPU.
- **The token accounting model follows the corrected formula: (accepted_LOC/year) × (total_tokens_touched_per_accepted_LOC) × (J/token), where total_tokens_touched includes output tokens, input/context tokens (system prompts, retrieved files, conversation history), and retry/regeneration tokens. This approach avoids the common undercount of treating codegen as output-only. All estimates deliberately favor the LLM side** (using improving hardware efficiency) and are conservative on the index side (generous overhead estimates). The actual savings ratio is likely to be higher than computed.